ПРОГРАММАЛАУ

C++ тілін пайдалану қағидалары

мен тәжірибесі

2-том

Бьярне Страуструп

Programming

Principles and Practice Using C++

Bjarne Stroustrup

ҚАЗАҚСТАН РЕСПУБЛИКАСЫ
БІЛІМ ЖӘНЕ ҒЫЛЫМ МИНИСТРЛІГІ

Бьярне Страуструп

ПРОГРАММАЛАУ

C++ тілін пайдалану қағидалары

мен тәжірибесі

2-том

Оқулық

Алматы, 2014

ӘОЖ
КБЖ
С

Қазақстан Республикасы Білім және ғылым министрлігінің «Оқулық» республикалық ғылыми-практикалық орталығы бекіткен

Қазақ тіліне аударғандар:
Бөрібаев Б., Адилгажинова С.

Страуструп Б.

С **Программалау. С++ тілін пайдалану қағидалары мен тәжірибесі. 2-том:** Оқулық / Ауд. Б. Бөрібаев, С. Адилгажинова. – Алматы, 2014. – 764 бет.

ISBN

Кітаптың құрылымы мен мақсатына байланысты, бұл оқулық жоғары оқу орындарындағы жаратылыстану саласында немесе техникалық бағытта білім алып жатқан студенттерге «Алгоритмдеу және программалау тілдері», «Алгоритмдер және мәліметтер құрылымы»,«Объектіге бағытталған программалау» және «Программалау технологиялары» тәрізді пәндерді оқу кезінде программалау тілі негіздерін үйрену, есептер шығаруды игеру мен зертханалық жұмыстар орындауды жеңілдетуге арналған.

Мұнда С/С++ тілдерін негізге ала отырып, программалау тәсілдерін үйретуден практикалық мағлұматтар беріліп, әрбір тақырып бойынша жинақталған көптеген есептердің программалау тіліндегі шығарылу жолдары толық көрсетілген. Әр тақырып соңында студенттердің өз беттерімен орындауларына арналған тапсырмалар келтірілген.

Ұсынылып отырған оқулық программалауды өз бетінше оқып үйренгісі келетін оқырмандардың да қажетіне жарайды деп саналады.

ӘОЖ
КБЖ

ISBN

Мазмұны

III бөлім

Мәліметтер және алгоритмдер

17

Векторлар және бос жады

"Келісім бойынша берілген vector мүмкіндіктерін пайдаланыңыз".

– Алекс Степанов (Alex Stepanov)

Осы тарау мен келесі 4-тарауда көбінесе STL деп аталатын C++ тілінің стандартты кітапханасының контейнерлері мен алгоритмдері сипатталады. Біз STL кітапханасының негізгі мүмкіндіктерін қарастырып, олардың қолданылу жолдарын көрсетеміз. Бұдан басқа, STL кітапханасын жасау барысында пайдаланылған жобалау және программалаудың негізгі тәсілдерін, сондай-ақ, мұнда қолданылған тілдің кейбір төменгі деңгейдегі қасиеттерін баяндаймыз. Мұндай қасиеттерге нұсқауыштар, жиымдар (массивтер) және компьютер жадының бос көлемін анықтау ісі жатады. Осы тарау мен келесі екі тарауда басты назар **STL** кітапханасындағы барынша танымал және пайдалы болып табылатын **vector** контейнерін жобалау мен оны жүзеге асыру ісіне арналады.

17.1 Кіріспе

C++ тілінің стандартты кітапханасында сипатталған ең пайдалы контейнер `vector` класы болып табылады. Векторда тек бір типтегі элементтер тізбегі сақталады. Біз вектор элементін оның индексі арқылы пайдалана аламыз, `push_back()` функциясының көмегімен оны кеңейте аламыз, `size()` функциясын пайдалана отырып, вектор элементтерінің санын анықтай аламыз, сондай-ақ, берілген диапазон шегінен шығып кетпеуді де жүзеге асырамыз. Стандартты вектор – статикалық типтер тұрғысынан алғанда, ыңғайлы, икемді, тиімді (жадыны пайдалану уақыты мен оның көлемі бойынша) және қауіпсіз контейнер. `string` стандартты класында осындай қасиеттермен қатар 20-тарауда сипатталатын мұнан өзге `list` және `map` сияқты стандартты контейнерлік типтердің де пайдалы қасиеттері бар.

Дегенмен, компьютер жады мұндай пайдалы типтерді тікелей сүйемелдеуді қамтамасыз ете алмайды. Аппараттық жабдықтамалар тек қана биттер тізбегінің жұмысын тікелей түрде сүйемелдей алады. Мысалы, `vector<double>` класындағы `v.push_back(2.3)` операциясы `double` типіндегі сандар тізбегіне `2.3` санын қосады да, `v` векторы элементтерінің санауышын бірге арттырады (`v.size()` функциясының көмегімен). Компьютер ең төменгі деңгейдегі `push_buck()` сияқты күрделі функциялар туралы ешнәрсе білмейді, оның бар білетіні – бір команданы орындау барысында бірнеше байтты қалай оқып, жазу керек екендігі ғана.

Біз осы және мұнан кейінгі екі тарауда кез келген программалаушы қол жеткізе алатын тілдің негізгі мүмкіндіктерін пайдалана отырып, **vector** класын қалай құруды көрсетеміз. Бұл программалаудың пайдалы әдістері мен тұжырымдамаларын көрсете отырып, оларды C++ тілі құралдарының көмегімен қалай бейнелеуге болатынын білдіру үшін жасалған. **vector** класын жүзеге асыру кезінде пайдаланылған программалау тілінің мүмкіндіктері мен әдістері өте тиімді болып табылады және олар кеңінен қолданылады.

Біз **vector** класының қолданылуын, жүзеге асырылуын және жобалауын анықтап алғаннан кейін, **map** сияқты басқа да стандартты контейнерлердің қалай жасалатынын түсініп, оларды пайдаланудың C++ (бұл туралы толық ақпарат 20-21-тарауларда беріледі) тілінің стандартты кітапханасы қамтамасыз ете алатын элементтік және тиімді әдістерін сынап көруімізге болады. Алгоритмдер деп аталатын мұндай әдістер мәліметті өңдеу программаларының типтік тапсырмаларын шешуге ғана мүмкіндік береді. Әркім өз бетінше қарапайым бір құрал жасауды ойлап тапқаннан гөрі, C++ тілінің кітапханасы арқылы программалар жазуды және оларды тесттен өткізуді жеңілдетуге көшуімізге болады. Біз стандартты кітапханадан алынған өте тиімді алгоритмдердің бірі – **sort()** мүмкіндіктерімен танысып, оны пайдаланып та көрдік.

Біз стандартты кітапхананың **vector** класына оны жүзеге асырудың біртіндеп күрделінетін нұсқаларын жасау арқылы жақындай түсеміз. Ең алдымен, біз **vector** қарапайым класын жасап аламыз. Содан соң оның кемшіліктерін анықтап, оларды түзетеміз. Бұны бірнеше рет қайталағаннан кейін, C++ тілінің компиляторымен бірге берілетін **vector** стандартты кітапхана класына өте жақын келетін, соған эквивалентті **vector** класын жасау жолдарын жүзеге асырамыз. Біртіндеп нақтыланатын бұл процесс программалау есептерін шешетін қарапайым жолды бейнелейді. Осының барысында біз мәліметтер құрылымы мен компьютер жадын пайдалануға байланысты көптеген классикалық мәселелерді анықтап, зерттеп шығамыз. Біздің негізгі жоспарымыз төменде көрсетілген:

- *17-тарау.* Жадының әртүрлі көлемімен қалай жұмыс істеуге болады? Негізінде, элементтерінің саны әртүрлі болып келген әртүрлі векторларды қалай жасауға болады және уақыттың әртүрлі сәттерінде жеке бір вектор элементтерінің саны қалай әртүрлі бола алады? Бұл бізді бос жадының көлемін (үйінді көлемін) тексеруге, нұсқауышқа, типтерді келтіруге (типтерді тікелей көрсету операторларына) және сілтемелерге алып келеді.

- *18-тарау.* Векторды қалай көшіруге болады? Элементтерді индекстері бойынша пайдалану операторын қалай жүзеге асыруға болады? Бұған қоса, біз жиымдарды қарастырып, олардың нұсқауыштармен байланысын зерттеу мәселелерін де енгіземіз.

- *19-тарау.* Сақталатын ішкі элементтерінің типтері әртүрлі болып келетін векторларды қалай жасаймыз? Берілген диапазон аймағынан тысқары

шығып кету қателерін қалай өңдейміз? Бұл сұрақтарға жауап беру үшін C++ тілінің шаблоны (template) мен аластамаларын (exception facilities) қарастырамыз.

Вектор типтері тұрғысынан алғанда, икемділік, тиімділік және қауіпсіздік үшін жасалған тілдің жаңа қасиеттері мен программалау әдістерінен бөлек, бұған дейін айтылған мүмкіндіктерді де пайдаланатын (қажетті кезінде, қайталап та) боламыз. Кей кездерде тым формальды түрдегі анықтауларды да келтіре аламыз.

Сонымен, барлығы да компьютер жадымен тікелей қатынас құруға (оны пайдалануға, қолдануға) келіп тіреледі. Бұл бізге не үшін керек? Біздің **vector** және **string** кластарымыз өте пайдалы және ыңғайлы; оларды жай ғана пайдалана беруге болады. Тереңірек қарастыратын болсақ, **vector** және **string** сияқты контейнерлер бізді нақты компьютер жадымен жұмыс істеудің келеңсіз аспектілерінен босату үшін жасап шығарылған. Егер біз ғажайыпқа сенбейтін болсақ, жадыны басқарудың ең төменгі деңгейін игеруіміз керек. Ал, неге ғажайыптарға, яғни **vector** класын жасаушылардың істеп шығарған жаңалықтарына сенбеске? Анығын айтсақ, біз компьютер жадының жұмысын қамтамасыз ететін физикалық құрылғыларды талдап жатқан жоқпыз ғой.

Мәселе біздің физик емес, программалаушы (компьютер ғылымының мамандары, программалық жабдықтама жасаушылар) болғанымызда. Егер біз физиканы оқығанымызда, онда біз компьютер жадының қызметі мен құрылғы бөлшектерін талдауға міндетті болар едік. Бірақ біз программалауды оқып үйренетін болғандықтан, программа құрылымы бөліктерінің жұмысын түсінуіміз керек. Теориялық тұрғыдан алғанда, компьютер жадымен төменгі деңгейде қатынас құру мен оны іске асыруды басқару құралдарын физикалық құрылғылар сияқты қарастыруға болар еді. Бірақ мұндай жағдайда біз ғажайыпқа сенуге ғана мәжбүр болмай, сондай-ақ жаңа контейнерлерді де (бізге ғана қажетті және стандартты кітапханада жоқ) жасай алмаған болар едік. Оның үстіне, жадыны тікелей пайдалану үшін C және C++ тілдерінде жазылған программалық кодтың өте көлемді топтамаларын түсіне де алмас едік. Келесі тарауларда көрсетілгендей, нұсқауыштар (объектіге сілтеме жасаудың тікелей және төмен деңгейлі тәсілі) тек компьютер жадын басқару мақсатында ғана пайдалы емес. Нұсқауыштардың қалай жұмыс істейтінін білмей, C++ тілін игеру де мүмкін емес.

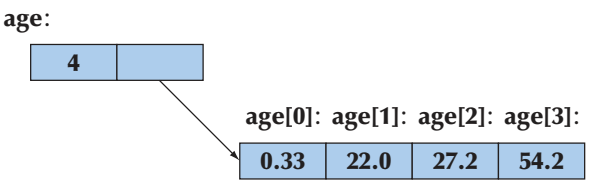
Абстрактілі түрде айтқанда, мен компьютер жадымен жұмыс істеуде теориялық және практикалық білімдері болмаған жандар үшін мәліметтер құрылымдарын өңдеу, алгоритмдер құру және операциялық жүйелер жасау сияқты жоғары деңгейлі тапсырмаларды шешу кезінде көптеген қиындықтар туындайды деп санайтын (компьютер ғылымы саласындағы) кәсіпқой мамандар тобына кіремін.

17.2 Vector негіздері

vector класын жасаудағы атқарылатын жұмыстар тізбегін қарапайым мысал қарастырудан бастаймыз.

```
vector<double> age(4); // double типіндегі төрт
                       // элементі бар вектор
age[0]=0.33;
age[1]=22.0;
age[2]=27.2;
age[3]=54.2;
```

Әрине, бұл код double типіндегі төрт элементі бар vector класының бір объектісін жасайды да, элементтерге 0.33, 22.0, 27.2, 54.2 мәндерін меншіктейді. Бұл төрт элементтің нөмірлері 0, 1, 2, 3 сандары болады. C++ тілінің стандартты контейнерлеріндегі элементтерді нөмірлеу әрқашанда нөлден басталады. Нөмірлерді нөлден бастау жиі пайдаланылады және бұл C++ тілінде программа жазатын барлық программалаушылар ұстанатын әмбебап келісім болып табылады. vector класының объектісіндегі элементтер саны оның өлшемі деп аталады. Сонымен, age векторының өлшемі төртке тең. Вектор элементтері 0-ден size-1 санына дейін нөмірленеді (индекстеледі). Мысалы, age векторының элементтері 0-ден age.size()-1 санына дейін нөмірленеді. age векторын келесі түрде көрсетуге болады:



Бұл кестені компьютер жадында қалай орналастыруға болады? Мәндерді қалай есте сақтауға және оларды қалай оқып пайдалануға болады? Әлбетте, біз класты анықтап, оның vector деп атауымыз керек. Ары қарай вектор өлшемін сақтауға кластың бір мүшесі және оның элементтерін сақтауға тағы бір мүше керек. Сандары өзгеріп отыратын элементтерді есте сақтауды қалай елестетуге болады? Бұл үшін vector стандартты класын пайдалануға болар еді, бірақ біздің мысалымызда бұл дұрыс болмас еді: біз осы класты жасау жолдарын қарастырып жатырмыз ғой.

Сонымен, суретте көрсетілген бағыттауыш тілсызықты қалай бейнелейміз? Оны жоқ деп есептейік. Біз берілген тұрақты өлшемдегі мәліметтер құрылымын анықтай аламыз.

```
class vector {
int size, age0, age1, age2, age3;
// . . .
};
```

Белгілеулерге байланысты кейбір нақтылықтарды есепке алмасақ, келесі суретке ұқсас құрылымды аламыз:

size:	age: age[0]:	age[1]:	age[2]:	age[3]:
4	0.33	22.0	27.2	54.2

Бұл қарапайым және әдемі көрінеді, бірақ біз `push_back()` функциясының көмегімен тағы бір элемент қосуға талпынсақ, қиын жағдайға кез боламыз: өйткені элементтер саны төртке тең деп белгіленгендіктен, басқа элемент қоса алмаймыз. Бізге элементтердің белгіленген санын ғана сақтайтын мәліметтер құрылымынан гөрі кеңірек, ауқымды мүмкіндік керек. Егер `vector` класындағы элементтер саны тұрақты болып белгіленген болса, онда `vector` класының объектісіндегі элементтер санын өзгертетін `push_back()` операциясын жүзеге асыру мүмкін емес. Негізінде, бізге көптеген элементтерге сілтеме жасай алатын класс мүшесі керек, ол жады көлемін кеңейту барысында басқа да элементтерге сілтеме жасай алуы тиіс. Бізге бірінші элементтің адресі ғана қажет болуы керек. C++ тілінде адресті есте сақтай алатын мәліметтер типін *нұсқауыштар (pointer)* деп атайды. Синтаксистік тұрғыдан алғанда, ол `*` *жалғауымен* (суффиксімен) белгіленеді, сондықтан `double*` тіркесі `double` типіндегі объектіге нұсқауышты білдіреді. Енді біз осы нұсқауыш ұғымын негізге ала отырып, `vector` класының бірінші нұсқасын анықтай аламыз.

```
// double типіндегі қарапайым элементтер векторы
// (vector <double> сияқты)
class vector {
 int sz;              // өлшемі
 double* elem;        // бірінші элементке нұсқауыш
                      //(double типінде)
public:
 vector (int s);      // конструктор: жадыға double типіндегі
                      // s санды орналастырады
                      // оған elem нұсқауышын орналастырады
                      // s санын sz мүшесінде сақтау
 int size() const {return sz; }  // ағымдағы өлшем
```

`Vector` класын жобалауды жалғастырудан бұрын "нұсқауыш" түсінігін түбегейлі түрде қарастырып шығайық. "Жиым" түсінігімен тығыз байланыстағы "нұсқауыш" түсінігі – бұл C++ тіліндегі "жады" түсінігін ұғудың негізгі кілті.

17.3 Компьютер жады, адрестер және нұсқауыштар

Компьютер жады – бұл байттардың тізбегі. Мұндағы байттар нөлден басталып, соңғысына дейін тізбекті түрде нөмірленеді. *Адрес* (address) деп компьютер жадындағы байттар сақталатын ұяшықтарды анықтау тәсілін айтады. Адресті бүтін сандардың реттелген тізбегі деп санауға болады. Компьютер жадының бірінші байты 0 деген адреспен, екіншісі 1-мен және т.с.с. түрде тізбектеліп (2, 3, 4, ...) жалғасып кете береді. Жадының бір мегабайтын келесідей түрде бейнелеп көрсетуге болады:

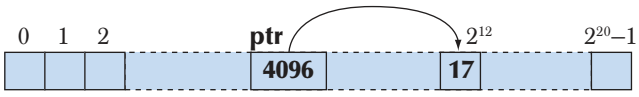
Компьютер жадында орналасқанның барлығының да адресі бар. Мысал қарайық:

```
int var=17;
```

Бұл нұсқау **var** айнымалысын сақтауға арналған мөлшері **int** типінің өлшемімен анықталатын компьютер жады көлемін бөліп береді де, оған 17 санын жазады. Бұған қоса, адрестердің өзін де есте сақтауға және оған да операциялар қолдануға болады. Адресті сақтайтын объектіні *нұсқауыш* деп атайды. Мысалы, **int** типіндегі объектіні сақтауға қажетті тип **int** типіне нұсқауыш деп аталады және **int*** түрінде белгіленеді.

```
int* ptr = &var;     // ptr нұсқауышы var айнымалысының
                     // адресін сақтайды
```

Объектінің адресін анықтау үшін адрес алу операторы – унарлы **&** қолданылады. Сонымен, егер **var** айнымалысы компьютер жадының адресі **4096** (немесе 2^{12}) нөміріне сәйкес ұяшығынан басталып сақталатын болса, онда **ptr** нұсқауышы **4096** санын сақтайтын болады.



Шын мәнісінде, компьютер жадын **0**-ден **size-1**-ге дейін нөмірленген байттардың тізбегі деп айтуға болады. Кейбір машиналар үшін мұндай пікір өте қарапайым түрде айтылған болар, бірақ біздің моделіміз үшін әзірше осылайша түсінік беру жеткілікті болып табылады.

Әрбір типтің өзіне сәйкес келетін нұсқауышының типі болады. Мысал қарастырайық:

```
char ch = 'c';
```

```
char* pc = &ch; // char типіне нұсқауыш
int ii = 17;
int* pi =&ii;   // int типіне нұсқауыш
```

Егер біз өзіміз сілтеме жасап отырған объектінің мәнін көргіміз келсе, онда нұсқауышқа унарлы * операторын, яғни "атаусыз ету" операторын қолдануға болады.

```
cout << "pc==" << pc << "; ішкі мәні pc==" << *pc << "\n";
cout << "pi==" << pi << "; ішкі мәні pi==" << *pi << "\n";
```

*pc мәні c символы, ал *pi мәні 17 бүтін саны болып табылады. pc және pi айнымалы мәндері компилятордың компьютер жадына ch пен ii айнымалыларын орналастыруына тәуелді болып келеді. Нұсқауыш (адрес) мәні үшін қолданылатын белгілеу де жүйеде қандай келісімдер қабылданғанына байланысты өзгеруі мүмкін; нұсқауыштар мәнін көрсету үшін көбінесе он алтылық сандар қолданылады (A.2.1.1 бөлімін қ.). Атаусыз ету операторы меншіктеу операторының сол жағында да тұруы мүмкін.

```
*pc ='x'; // OK: pc нұсқауышы сілтеме жасап тұрған
          // char айнымалысына
          // 'x' символын меншіктеуге болады
*pi = 27;  // OK: int* нұсқауышы int типіне сілтеме жасайды,
          // сондықтан *pi – бұл int типі болады
*pi=*pc;  // OK: (ps) символын int типіндегі айнымалыға
          // (pi) меншіктеуге болады
```

Мынаған назар аударыңыз! Нұсқауыштың мәні бүтін сан болғанымен, нұсқауыштың өзі бүтін сан болып табылмайды. "Int неге сілтеме жасайды?" деген сұрақты қою – дұрыс емес. Бүтін сан емес, нұсқауыштар ғана сілтеме жасайды. Int типі бүтін сандармен операцияларды (арифметикалық және логикалық) орындауға мүмкіндік беріп жатқанда, нұсқауыш типі адрестермен операциялар орындауға мүмкіндік береді. Сонымен, нұсқауыштар мен бүтін сандарды шатастыруға болмайды.

```
int i = pi;   //қате: int* типті объектіні int типті объектіге
              // меншіктеуге болмайды
pi = 7;       //қате: int типті объектіні int* типті объектіге
              // меншіктеуге болмайды
```

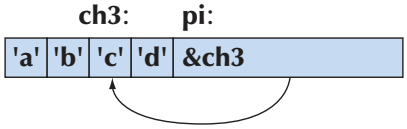
Осылар сияқты char типіне (яғни char*) нұсқауыш int типіне (яғни, int*) нұсқауыш болып табылмайды. Мысал қарастырайық:

```
pc = pi; // қате: int* типті объектіні char* типті объектіге
         // меншіктеуге болмайды
pi = pc; // қате: char* типті объектіні int* типті объектіге
         // меншіктеуге болмайды
```

pi айнымалысына **pc** айнымалысын неге меншіктеуге болмайды? Жауаптардың бірі – **char** символы **int** типінен әлдеқайда кіші.

```
char ch1 = 'a';
char ch2 = 'b';
char ch3 = 'c';
char ch4 = 'd';
int* pi = &ch3; // char типінің өлшеміндегі айнымалыға
                // сілтеме жасайды
               // қате: char* типті объектіге int* типті
               // объектіні меншіктеуге болмайды
// бір сәтке біз осылай меншіктей аламыз деп ойлап көрейік
*pi = 12345;  // жадының char типіндегі мәлімет сақтайтын
              // жеріне жазуға талпыну
*pi = 67890;
```

Компилятордың айнымалыларды компьютер жадында қалай орналастыратыны оның іске асуына (орындалуына) байланысты, бірақ ол көбінесе былай болып шығуы мүмкін:



Егерде компилятор осындай кодты өткізіп жіберген болса, онда компьютер жадының **&ch3** адресінен басталатын ұяшығына **12345** санын жазып қояр едік. Ол осы жады ұясының алдындағы мәліметтер мәнін, яғни **ch2** және **ch4** ұяларындағы айнымалылар мәнін өзгертер еді. Болмаған жағдайдың өзінде (нақты жағдайда), **pi** айнымалысының тек бір бөлігін ғана қайта жазып алар едік. Мұндайда келесі ***pi = 67890** меншіктелуі **67890** санын компьютер жадының мүлде басқа аймағына орналастыруға әкеліп соғар еді. Осындай меншіктелуге тыйым салынғаны өте дұрыс болған, бірақ программалаудың төменгі деңгейіндегі мәліметтерді қорғау механизмдері өте аз мөлшерде орындалады.

Бірен-саран жағдайларда, бізге **int** типіндегі айнымалыны нұсқауышқа түрлендіру немесе бір типті екінші бір типке ауыстыру қажет болғанда, **reinterpret_cast** операторын пайдалану керек (бұл туралы толығырақ 17.8-тарауда).

Сонымен, біз аппараттық жабдықтамаға өте жақын келдік. Программалаушыға бұл онша ыңғайлы емес. Мұнда біздің қолымызда тек бірнеше қарапайым

операциялар ғана бар және оны кітапханалық сүйемелдеу де жоқтың қасы. Дегенмен, бізге `vector` класы сияқты жоғары деңгейлі құралдар қалай жасалғанын білу керек. Біз төменгі деңгейде қандай код жазу керек екенін білуіміз керек, өйткені кез келген код жоғары деңгейлі бола бермейді (25-тарауды қ.). Мұнымен қоса, жоғары деңгейлі программалаудың ыңғайлылығы мен салыстырмалы түрдегі сенімділігін бағалау үшін төменгі деңгейдегі программмалаудың күрделілігін сезіну керек. Біздің мақсатымыз – әрдайым қойылған мақсат пен тұжырымдалған шектеулер мүмкіндік беретін абстракцияның ең жоғарғы деңгейінде жұмыс істеу. Осы тарауда, сондай-ақ, 18-19-тарауларда біз `vector` класын жүзеге асыру арқылы абстракцияның барынша ыңғайлы болып саналатын деңгейіне қалай оралуға болатынын көрсетеміз.

17.3.1 sizeof операторы

Сонымен, `int` типін сақтау үшін қандай жады көлемі керек? Ал нұсқауышты ше? Бұл сауалдарға `sizeof` операторы жауап береді.

```
cout << "char типінің өлшемі" << sizeof(char) << ' '
     << sizeof('a') << '\n';
cout << "int типінің өлшемі"  << sizeof (int) << ' '
     << sizeof(2+2) << '\n';
int* p = 0;
cout << "int* типінің өлшемі" << sizeof(int*) << ' '
     << sizeof(p) << '\n';
```

Көріп отырғанымыздай, `sizeof` операторын тип атына да, өрнекке де қолдануға болады; тип үшін `sizeof` операторы көрсетілген типтегі объектінің өлшемін, ал өрнекке оның нәтижесінің типі өлшемін қайтарады. `sizeof` операторының нәтижесі – оң бүтін сан, ал жады көлемін өлшеу бірлігі, анықтама бойынша `1`-ге тең `sizeof (char)` мәні болып табылады. Көбінесе `char` типі `1` байт орын алады, сондықтан `sizeof` операторы байттар санын қайтарады.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Жоғарыда келтірілген кодты орындап, нәтижесіне қараңыз. Содан соң осы мысалды `bool`, `double` және өзге де бірнеше типтердің өлшемін анықтау үшін кеңейтіп орындаңыз.

C++ тілінің әртүрлі жүзеге асқан орталарында (компиляторларда) бір типтің өлшемі бір-біріне сәйкес келуі міндетті емес. Қазіргі кезде `sizeof(int)` өрнегінің мәні үстелге қоятын компьютерлер мен ноутбуктерде көбінесе төртке тең. Бір байт

8 биттен тұратын болғандықтан, **int** типі 32 биттен тұрады. Дегенмен, кіріктірілген жүйелердегі процессорда **int** типі 16 бит орын алады, ал жоғары өнімді архитектурада **int** типінің өлшемі әдетте 64 битке тең болып келеді.

Vector класының объектісі компьютер жадынан қанша орын алады? Анықтауға тырысайық.

```
vector<int> v(1000);
cout << "vector<int>(1000) типіндегі объектінің өлшемі"
                          = " << sizeof(v) << '\n';
```

Нәтиже мынадай болуы мүмкін:

```
the size of vector<int>(1000) is 20
```

Мұндай көріністің себебі осы және келесі тарауды (сондай-ақ 19.2.1 бөлімде) оқығаннан кейін анық болады, бірақ **sizeof** операторының элементтерді жай ғана санап шықпайтыны қазірден-ақ белгілі болып отыр.

17.4 Бос жады және нұсқауыштар

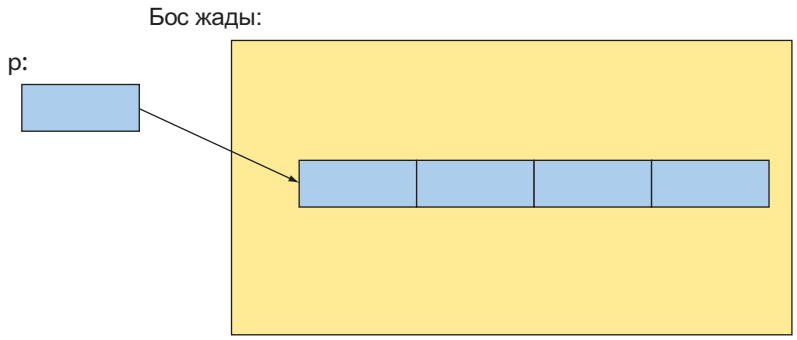
17.2 бөлімнің соңында келтірілген **vector** класының жүзеге асырылуын қарастырып көрейік. **vector** класы өзінің элементтерін сақтауға орынды қайдан табады? Оларға сілтеме жасауы үшін **elem** нұсқауышын қалай орнатамыз? C++ тілінде жазылған программаның орындалуы қашан басталады, компилятор қашан оның коды (кейде мұндай жадыны *кодтық (code storage)* деп немесе *мәтіндік (text storage)* деп атайды) және ауқымды (*глобальды*, оны *статикалық (static storage)* деп те атайды) айнымалылары үшін компьютер жадын бөледі?

Компилятор, бұдан басқа, функцияларды шақырғанда оның аргументтері мен жергілікті айнымалыларын (бұл *стек жады (stack storage)* немесе *автоматтық (automatic storage) жады* деп те аталады) сақтауға арналған жады көлемін бөледі. Компьютер жадының қалған бөлігі басқа мақсаттарға пайдаланылуы мүмкін; ол *бос жады (free)* деп аталады. Жады көлемін осылай бөлуді мынадай түрде бейнелеуге болады.

C++ тілі бұл бос жадыны (сондай-ақ, *үйінді* (куча – heap) деп те атайды) **new** операторы арқылы пайдалану мүмкіндігін береді. Мысал қарастырайық:

```
double* p = new double[4]; // double типіндегі 4 санды
                           //бос жады аймағына орналастырамыз
```

Жоғарыда көрсетілген нұсқау программаны орындайтын жүйеден **double** типіндегі 4 санды бос жады аймағына орналастырып, нұсқауышты солардың біріншісіне қайтаруды сұрайды. Бұл нұсқауыш **p** айнымалысын инициалдау үшін қолданылады. Сұлбалық тұрғыдан алғанда, ол мынадай болып көрінеді:



new операторы өзі жасаған объектіге нұсқауышты қайтарады. Егер **new** операторы бірнеше объектілер (жиым) жасап шыққан болса, онда ол нұсқауышты осы жиым элементтерінің алғашқысына (біріншісіне) қайтарады. Егер бұл объект **X** типінде болса, онда **new** операторы қайтаратын нұсқауыш **X*** типінде болады. Мысал қарастырайық:

```
char* q = new double[4];    // қате: double* нұсқауышы
                            // char*-ға меншіктеледі
```

Осы берілген **new** операторы нұсқауышты **double** типіндегі айнымалыға қайтарады, бірақ **double** типі **char** типінен басқаша болады, сондықтан біз **double** типіндегі айнымалыға нұсқауышты **char** типіндегі айнымалыға нұсқауышқа меншіктей алмаймыз (және ол болмайды да).

17.4.1 Компьютердің бос жадына орналастыру

new операторы компьютердің *бос жадын* (free store) *бөлу* (allocation) ісін атқарады:

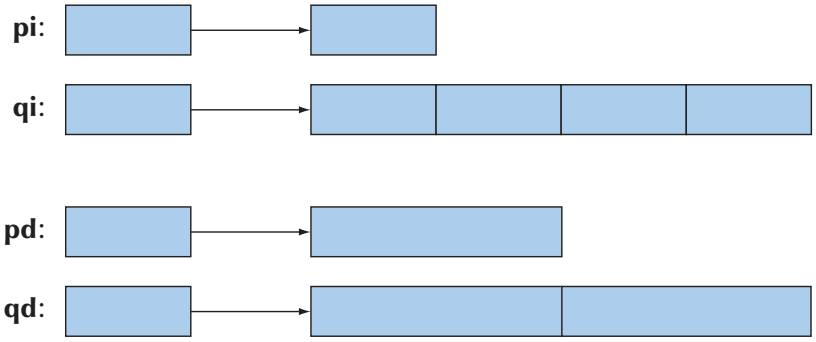
- **new** операторы нұсқауышты бөлінген жады аймағына қайтарады;
- Нұсқауыштың мәні бөлінген жадының бірінші байтының адресі болып табылады;

- Нұсқауыш көрсетілген типтің объектісіне сілтеме жасайды;
- Нұсқауыш элементтің қанша санына сілтеме жасағанын *білмейді*.

new операторы жеке элементтерге де, сондай-ақ элементтер тізбегіне де жады бөліп бере алады. Мысал қарастырайық:

```
int* pi = new int;           // бір int айнымалысына
                             // жады бөлеміз
int* qi = new int [4];       // төрт int айнымалысына
                             // жады   бөлеміз (жиым)
double* pd = new double;     // бір айнымалыға жады бөлеміз
                             // ол double типінде
double* qd = new double[n];// n айнымалыға жады бөлеміз
                             // олар double типінде
```

Объектілердің саны айнымалы түрінде беріле алатынына назар аударыңыздар. Бұл бізге программаның орындалу барысында бөлінетін жады аймағына қанша жиым орналастыра алатынымызға таңдау жасауға мүмкіндік беретіндіктен, өте маңызды болып табылады. Егер **n** екіге тең болса, онда келесідей нәрселер орын алады.



Әртүрлі типтердегі объектілерге нұсқауыштар да әртүрлі типке жатады. Мысал қарастырайық:

```
pi = pd; // қате: double* нұсқауышын int* нұсқауышына
                             // меншіктеуге болмайды
pd = pi; // қате: int* нұсқауышын double* нұсқауышына
                             // меншіктеуге болмайды
```

Неге болмайды? Негізінде, біз **int** типіндегі айнымалыны **double** типіндегі айнымалыға және де керісінше меншіктей алмаймыз ғой. Оның себебі **[]** операторында болып тұр. Элементті табу үшін оның типінің өлшемі туралы мәлімет

пайдаланылады. Мысалы, `qi[2]` элементі `qi[0]` элементінен екі `int` өлшеміне тең қашықтықта орналасады, ал `qd[2]` элементі (`double` типінде) `qd[0]` элементінен екі `double` өлшеміне тең қашықтықта тұр. Егер `int` типінің өлшемі (яғни ені) `double` типінің өлшемінен басқаша болса (ол барлық компьютерлерде басқаша), онда `qi` нұсқауышына `qd` нұсқауышы үшін (адрестелуі үшін) бөлінген жадыға сілтеме жасауға рұқсат етіп, біз біраз келеңсіз нәтижелер алған болар едік.

Бұл тәжірибелік тұрғыдан берілген түсініктеме, ал теориялық тұрғыдан алғанда, мұның жауабы мынадай: әртүрлі типке берілген нұсқауыштарды бір-біріне меншіктеу *типтік қателердің* (type errors) туындауына себепші болады.

17.4.2 Нұсқауыштар арқылы қатынас құру

Нұсқауышқа `*` атаусыз ету операторынан басқа `[]` индекстеу операторын қолдануға болады. Мысал қарастырайық:

```
double* p=new double[4]; // бос жады аймағынан double типіндегі
                         // төрт айнымалыға жады бөлеміз
double x = *p;  // p сілтеме жасайтын (бірінші объектіні оқимыз
double y = p[2];  // p сілтеме жасайтын үшінші объектіні оқимыз
```

`vector` класындағыдай индекстеу операторы санауды нөлден бастайды. Бұл `p[2]` өрнегі үшінші элементке сілтеме жасайды деген сөз; `p[0]` – бірінші элемент, сондықтан `p[0]` және `*p` мағыналары бірдей болып табылады. `[]` және `*` операторларын жазу үшін де пайдалануға болады.

```
*p = 7.7;   // санды p сілтеме жасап тұрған (бірінші)
                                  // объектіге  жазамыз
p[2] = 9.9; // санды p сілтеме жасап тұрған үшінші
                                  // объектіге жазамыз
```

Нұсқауыш компьютер жадында орналасқан объектіге сілтеме жасайды. Атаусыз ету ("*contents of*") операторы ("*the dereference operator*" деп те аталады) `p` нұсқауышы сілтеме жасап тұрған объектіні оқуға және оны жазуға да мүмкіндік береді.

```
double x = *p; // p нұсқауышы сілтеме жасап тұрған
                                  // объектіні  оқимыз
*p= 8.8;       // p нұсқауышы сілтеме жасап тұрған
                                  // объектіні жазамыз
```

`p` нұсқауышына `[]` операторы қолданылған кезде, ол компьютер жадын объектілер тізбегі (нұсқауышты жариялағанда көрсетілген типтегі тізбек) ретінде

қарастырады. **p** нұсқауышы сол тізбектің алғашқысының адресіне сілтеме жасап тұрады.

```
double x = p[3];  // p сілтеме жасап тұрған төртінші
                                    // объектіні оқимыз
p[3] = 4.4;      // p сілтеме жасап тұрған төртінші
                                    // объектіні жазамыз
double y = p[0]; // p[0] және *p бірдей болып саналады
```

Міне, осымен бітті. Мұнда ешқандай тексеру, күрделі әрекеттер жоқ, тек компьютер жадымен тікелей қатынас құру әрекеті ғана орындалады.

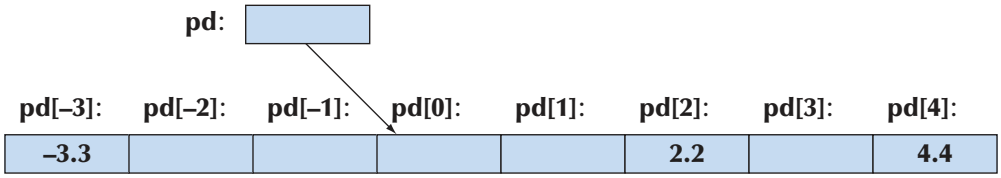
Дәл осындай жадымен тікелей қатынас құруға (оны пайдалануға) мүмкіндік беретін қарапайым, оңтайлы, әрі тиімді механизм **vector** класын жүзеге асыру үшін қажет болды.

17.4.3 Диапазондар

Нұсқауышпен байланысты негізгі мәселенің бірі оның өзі сілтеме жасайтын элементтер санын білмейтіндігінде. Мысал қарастырайық:

```
double* pd = new double[3];
pd[2] = 2.2;
pd[4] = 4.4;
pd[-3] = -3.3;
```

pd нұсқауышы үшінші **pd[2]** элементіне сілтеме жасай ала ма? Ол бесінші **pd[4]** элементіне сілтеме жасай ала ма? Егер біз pd нұсқауышының анықтамасына қарасақ, сәйкесінше "иә" және "жоқ" деп жауап бере аламыз. Бірақ компилятор бұл туралы білмейді, ол нұсқауыш мәнін бақыламайды. Біздің кодымыз компьютер жадын ол дұрыс бөлініп берілген сияқты түрде пайдалануға тырысады. Компилятор тіпті **pd[-3]** өрнегіне де қарсы шықпайды, ол **pd** нұсқауышы сілтеме жасап тұрған элементтің алдына **double** типіндегі 3 сан қоюға болмайтынын да есепке алмайды.



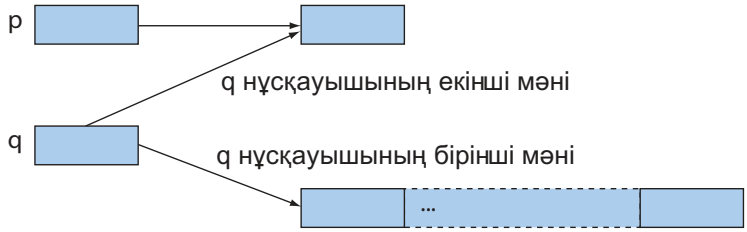
`pd[-3]` және `pd[4]` өрнектері сілтеме жасап тұрған жады ұяшығының өзінің де не ұсынатыны бізге беймәлім. Дегенмен, олардың `pd` нұсқауышы сілтеме жасап тұрған `double` типіндегі 3 сан сақталған біздің жиымымыздың бөлшегі ретінде қолданыла алмайтыны анық. Болжауымызша, олар басқа бір объектінің бөлшегі болып табылады, біз жай ғана қателесіп отырмыз. Бұл жаман. Бұл өте апатты жағдай. Бұл жерде "апатты" сөзі не "менің программам апатты түрде аяқталды", не "менің программам дұрыс жауап бермей тұр" дегенді білдіреді. Мұны дауыстап айтуға тырысып көріңіз; құлаққа түрпідей естіледі. Мұндай жағдайлардан аулақ болу үшін көп нәрсені үйрену керек. Қолдануға рұқсат берілген (шектелген) аймақтан, яғни диапазоннан шығып кету аса қауіпті, ол біздің программамызға қатысы жоқ басқа да мәліметтерді де зақымдауы мүмкін. Шектелген диапазоннан тысқары жатқан жады ұяшығының ішкі мәліметін оқи отырып, мүлде басқа есептеулердің нәтижесі болып шығатын кездейсоқ сандар аламыз. Шектелген диапазоннан тысқары тұрған жады ұяшығына мәлімет жазу арқылы кез келген бір объектіні "мүмкін болмайтын" келеңсіз жағдайға душар етуіміз ықтимал немесе мүлде күтілмеген қатесі бар мән алуымыз мүмкін. Мұндай іс-әрекеттер көбінесе ұзақ уақытта дейін көзге түспей елеусіз болып қала береді, сондықтан да оларды анықтау өте қиын. Бұдан да жаманы: шектелген диапазоннан шығып кеткен программаның берілген мәліметтерін аз ғана өзгертіп, қатарынан екі рет орындауда мүлде үйлеспейтін нәтижелерге тап болуымыз мүмкін. Осындай қателерді (тұрақсыз қателер) анықтап табу бәрінен де қиын мәселе болып саналады.

Біз шектелген диапазоннан шығу болмайтынына кепілдік беруіміз керек. Біздің компьютер жадын `new` операторының көмегімен тікелей бөлмей, `vector` класын қолдануымыздың бір себебі – ол өзінің өлшемін біледі, сондықтан шектелген диапазоннан шығып кетудің алдын алуға болады.

Шектелген диапазоннан шығып кетудің алдын алу көптеген себептермен қиын болып саналады. Оның бір себебі – нұсқауыш сілтеме жасап тұрған элементтер санына қарамастан, біз бір `double*` нұсқауышын басқа бір `double*` нұсқауышына меншіктей алатынымызда болып тұр. Нұсқауыш, шынында да, қанша элементке сілтеме жасайтынын білмейді. Мысал қарастырайық:

```
double* p = new double;       // double типіндегі
                              // айнымалыны орналастыру
double* q = new double[1000]; // double айнымалысына double
                    // типіндегі мың айнымалыны орналастыру
q[700]=7.7;         // өте жақсы
q = p;              // q нұсқауышы мен p нұсқауышы
                    // бір аймаққа сілтеме жасайды
double d = q[700]; // шектелген диапазоннан шығып кету!
```

Осы үш қатар кодта `q[700]` өрнегі екі түрлі жады ұяшығына сілтеме жасап тұр, оның үстіне, екінші жағдайда берілген диапазоннан шығып кету орын алған.



Ал енді сіздер "Нұсқауыш неліктен жадының өлшемін есте сақтай алмайды?" деген сұрақ қояды ғой деп ойлаймыз. Қанша элементке сілтеме жасап тұрған есте сақтайтын нұсқауыш ойлап табуға болар еді, – **vector** класында бұл осыған өте жақын жасалған десек те болады. Егер де C++ тіліне арналған кітаптарды оқып, оның кітапханаларын қарап шықсаңыздар, құрамындағы төменгі деңгейлі нұсқауыштардың бұл кемшіліктерінің орнын толтыратын көптеген "зерделі нұсқауыштармен" танысасыздар. Алайда кейбір жағдайларда бізге төменгі деңгейде қатынас құру әрекеттерін және объектіні адрестеу механизмін түсіну керек болады, ал машина нені адрестеп жатқанын білмейді. Оның үстіне нұсқауыштар жұмысының механизмдерін білу бүгінгі күнге дейін жазылған көптеген программаларды түсіну үшін өте маңызды болып саналады.

17.4.4 Инициалдау

Әдеттегідей, біз пайдалануға кіріскенше объектінің қандай да бір мәні болғандығы дұрыс болар еді, басқаша айтқанда, біз нұсқауыштар мен олар сілтеме жасап тұрған объектілердің инициалданғандығын қалар едік. Мысал қарастырайық:

```
double* p0;                    // инициалдаусыз жариялау:
                               // мәселе туындауы мүмкін
double* p1=new double;         // инициалдаусыз double типті
                               // айнымалы үшін жады бөлу
double* p2=new double(5.5); // double типіндегі айнымалыны
                               //  5.5 санымен инициалдау
double* p3=new double[5];      // инициалдаусыз double типті
                               // бес саннан тұратын жиым
                               // үшін жадының бөлінуі
```

Әрине, **p0** нұсқауышының инициалдаусыз жариялануы мәселе туындатуы мүмкін. Мысал келтірейік:

```
*p0=7.0;
```

Бұл нұсқау **7.0** санын компьютер жадының белгілі бір ұяшығына жазады.

Біз бұл ұяшықтың компьютер жадының қай бөлігінде орналасқандығын білмейміз. Бұл қауіпсіз болуы мүмкін, бірақ оған сенім артудың қажеті жоқ. Ерте ме, кеш пе біз берілген диапазоннан шығып кеткен кездегідей нәтиже аламыз: программа жұмысын апатты түрде тоқтатады немесе қате нәтижелер береді. C тілінің ескі стилінде жазылған программалардың көп мәселелері инициалданбаған нұсқауыштардың қолданылуынан және қолдануға рұқсат етілген диапазоннан шығып кетуінен болған. Осындай мәселелерге тап болмас үшін және жиі қайталанатын типтік қателерді іздеуге уақыт жібермеу үшін қолымыздан келгеннің бәрін істеуіміз керек, өйткені біздің мақсатымыз – жоғары кәсіби деңгейде программа құра білу.

Мұндай қателерді табу және жөндеп түзету – жалықтыратын әрі жағымсыз іс. Осындай қателіктердің алдын алу оларды табудан анағұрлым жағымды әрі өнімді жұмыс болып табылады.

Құрамдас типтер үшін **new** операторы арқылы бөлінген жады инициалданбайды. Егер нұсқауышты инициалдағыңыз келсе, **p2** нұсқауышын жариялағандағы сияқты нақты мән беріңіз: ***p2** нұсқауышының мәні **5.5** санына тең. Инициалдау кезінде қолданылатын дөңгелек жақшаларға **()** назар аударыңыз. Оларды жиымдарды (массивтерді) индекстеу үшін пайдаланылатын **[]** квадрат жақшалармен шатастырмаңыз.

C++ тілінде **new** операторы арқылы компьютер жады бөлінген, құрамдас типтегі объектілер жиымын инициалдауға арналған құралдар жоқ. Жиымдар үшін мұндай жұмысты жекелеген түрде орындау қажет болады. Мысал қарастырайық:

```
double* p4=new double[5];
for (int i=0; i<5; ++i) p4[i]=i;
```

Енді **p4** нұсқауышы құрамында **0.0**, **1.0**, **2.0**, **3.0** және **4.0** сандары бар **double** типіндегі объектілерге сілтеме жасайды.

Әдеттегідей, инициалданбаған объектілерді болдырмау үшін олар пайдаланылғанға дейін нақты бір мәнге ие болуын қадағалауымыз керек. Компиляторларда айнымалыны болжамды шамамен (әдетте нөлмен) инициалдайтын жөндеп түзетуші режим болады. Мынадай жайттар да кездеседі: программа жазылып болған соң, оны тапсырыс берушіге жіберу үшін компилятордың түзету режимін тоқтатып, оптимизаторды іске қосамыз немесе программаны басқа бір компьютерде компиляциядан өткіземіз, сондай кездерде инициалданбаған айнымалылары бар программа аяқасты дұрыс жұмыс істеуін тоқтатуы мүмкін. Инициалданбаған айнымалыларды қолданбаңыз. Егер **X** класының келісім бойынша конструкторы болатын болса, онда мынадай нәтиже аламыз:

```
X* px1 = new X;       // келісім бойынша инициалданған
                      // X класының бір объектісі
X* px2 = new X[17];   // келісім бойынша инициалданған
                      // X класының 17 объектісі
```

Егер **Y** класының конструкторы бар болғанмен, бірақ ол келісім бойынша орнатылмаған болса, біз оны тікелей түрде инициалдауымыз керек.

```
Y* py1 = new Y;       // қате:келісім бойынша конструктор жоқ
Y* py2 = new Y[17];   // қате:келісім бойынша конструктор жоқ
Y* py3 = new Y(13);// ОК: Y(13) объектісінің
                      // адресімен инициалданған
```

17.4.5 Нөлдік нұсқауыш

Егер сіздің нұсқауышыңызды инициалдайтын басқа нұсқауыш болмаса, онда **0** мәнін пайдаланыңыз.

```
double* p0=0 // нөлдік нұсқауыш
```

Нөл мәні берілген нұсқауыш *нөлдік* (null pointer) деп аталады. Нұсқауыштың дұрыстығы (яғни, оның басқа бір нәрсеге сілтеме жасап тұрғандығы) оны нөлмен салыстыру арқылы тексеріледі. Мысал қарастырайық:

```
if(p0!=0) // p0 нұсқауышының дұрыстығын тексеру.
```

Бұл тест әмбебап емес, өйткені **p0** нұсқауышында кездейсоқ түрде берілген нөлге тең емес мән болуы немесе құрамында **delete** операторының көмегімен жойылып кеткен объект адресі болып қалуы мүмкін. Дегенмен, көбінесе мұндай тексеру – орындауға болатын әрекеттердің ішіндегі ең жақсысы болып табылады. Біздің оған 0 мәнін тікелей көрсетуіміз міндетті емес, өйткені **if** нұсқауы келісім бойынша шарттың нөлдік емес екенін ғана бақылайды.

```
if (p0)  // p0 нұсқауышының дұрыстығын тексеру;
         // p0!=0 мәніне эквивалентті
```

Біз "**p0** дұрыс" деген өрнектің мағынасын дәлірек береді деген оймен, тексерудің анағұрлым қысқа формасын таңдаймыз, дегенмен, ол әркімнің талғамына байланысты шешіледі.

Нөлдік нұсқауышты белгілі бір нұсқауыш бір объектіге бірде сілтеме жасап, ал кейде сілтеме жасамай тұрған жағдайда қолданған дұрыс. Мұндай жағдай біз ойлағаннан гөрі сирегірек кездеседі. Өзіңіз ойлап көріңізші, егер сіздің нұсқауыш орнатуға болатын объектіңіз болмаса, онда нұсқауыштың өзін анықтаудың қажеті не? Объектінің құрылғанын неге күтпеске?

17.4.6 Компьютердің бос жадын босату

new операторы компьютердің бос жады аймағын бөліп беру ісін атқарады. Компьютер жады шектеулі болғандықтан, оның қажетсіз болған аймағын босатып кері қайтару жаман болмас еді. Мұндай кезде босаған жады көлемін басқа объектілерді сақтау үшін қолдануға болар еді. Көлемді және ұзақ жұмыс істейтін программалар үшін осылай компьютер жадын босату ісі маңызды рөл атқарады. Мысал келтірейік:

```
double* calc(int res_size, int max) //жадының азаюы
{
  double* p = new double[max];
  double* res = new double[res_size];
  //p нұсқауышын нәтижелерді есептеу және
  //оларды res жиымына жазу үшін қолданамыз
  return res;
}
double* r = calc(100,1000);
```

Осы программаға сәйкес **calc()** функциясын әрбір шақыру кезінде бос жады аймағынан мөлшері **double** типінің өлшеміне тең орын алынады да, оның адресі **p** нұсқауышына меншіктеледі. Мысалы, **calc(100, 1000)** өрнегін шақыру **double** типті жүз айнымалы орналасатын жады аймағын программаның қалған бөлігі үшін қолжетімсіз етеді.

Мәліметтерден босатылған жады аймағын басқа программаларға қолданысқа беру үшін пайдаланылатын (қайтарушы) оператор **delete** деп аталады. Алдағы уақытта қолданысқа кіретін компьютер жадын босату үшін, **delete** операторын **new** операторы арқылы қайтарылған нұсқауышқа пайдалану керек. Мысал келтірейік:

```
double* calc(int res_size, int max)
  //res жиымына арналып бөлінген компьютер жадын
  //қолданғаны үшін шақырушы модуль жауапты
{
  double* p = new double[max];
  double* res = new double[res_size];
  //p нұсқауышын нәтижелерді есептеп, оларды
  //res жиымына жазу үшін қолданамыз
  delete[ ] p; //бұл жады енді қажет емес; оны босатамыз
  return res;
}
  double* r = calc(100,1000); //r нұсқауышын қолданамыз
  delete[] r; //бұл жады енді қажет емес: оны босатамыз
```

Былайша айтқанда, бұл мысал бос жады аймағын қолданудың негізгі себептерінің бірін көрсетеді: біз функцияларда объектілер құрып, оларды шақырушы модульге кері жібере аламыз.

Delete операторының екі түрі бар:

- **delete p** жеке объект үшін **new** операторының көмегімен бөлінген компьютер жадын босатады;
- **delete []p** объектілер жиымы үшін **new** операторының көмегімен бөлінген компьютер жадын босатады.

Дұрыс нұсқаны программалаушы таңдауы тиіс.

Объектіні екі рет жою – өрескел қате болып саналады. Мысал келтірейік:

```
int* p = new int(5);
delete p;   // өте жақсы: p new операторы арқылы жасалған
            // объектіге сілтеме жасап тұр
            //... нұсқауыш енді мұнда қолданылмайды...
delete p;   // қате: p бос жады диспетчеріне тиесілі
            // жады аймағына сілтеме жасап тұр
```

Delete p өрнегінің екінші нұсқасы екі мәселе туғызады:

- Сіз енді объектіге сілтеме жасамайсыз, сондықтан бос жады диспетчері мәліметтердің ішкі құрылымын бұдан былай **delete p** нұсқауын екінші рет дұрыс орындау мүмкін болмайтындай етіп өзгерте алады;
- Бос жады диспетчері **p** нұсқауышы бұрын сілтеме жасаған жады аймағын қайта пайдалануы мүмкін, өйткені енді **p** нұсқауышы басқа объектіге сілтеме жасайды; бұл объектіні (программаның басқа бөлігіне тиесілі) жою қателік туғызуы мүмкін.

Осы екі мәселе де нақты программаларда кездеседі, демек бұл тек теориялық мүмкіндіктер ғана емес.

Нөлдік нұсқауышты жоюдың ешқандай салдары болмайтындықтан (нөлдік нұсқауыш бір де бір объектіге сілтеме жасамайды), бұл операцияның зияны жоқ. Мысал келтірейік:

```
int* p = 0;
delete p;// өте жақсы: ешқандай әрекеттер жасау қажет емес
deletep; // бұл да жақсы (бұрынғыдай ештеңе істеуді қажет етпейді)
```

Осылай компьютер жадын босату үшін әуре болудың қажеті қанша? Жады бізге қажет болмай қалған кезде компилятор оны өзі түсініп, адамның қатысынсыз-ақ оны босата алмай ма? Босата алады. Мұндай механизм қоқысты *автоматты*

түрде жинау (automatic garbage collection) немесе жай ғана *қоқыс жинау* (garbage collection) деп аталады. Өкінішке орай, қоқысты автоматты түрде жинау арзан нәрсе емес және ол барлық қосымша программаларға сәйкес келе бермейді. Егер сізге қоқысты автоматты түрде жинау механизмі қажет болса, сол механизмді өз программаңызға енгізіп орната аласыз. Қоқыс жинаушы жақсы программаларды *www.research.att.com/~bs/c++.htm* адресі бойынша табуға болады. Дегенмен, бұл кітапта оқырмандар өз программаларындағы "қоқыстарын" өздері табады деп есептейміз, біз тек оны қалайша тиімді әрі ыңғайлы істеуге болатынын көрсетеміз.

Жады көлемін босату неге маңызды болып табылады? Тоқталмай жұмыс істеуге арналған программа жадының босауына ешқашан жол бермеуі тиіс. Мұндай программаларға мысал ретінде операциялық жүйені, сонымен қатар, компьютерлік жүйемен құрамдас кіріктірме жүйелердің көбін айтуымызға болады (олар туралы 25-тарауда айтылады). Кітапханалар да компьютер жадының кемуіне жол бермеуі тиіс, себебі біреулер бұл кітапханаларды шексіз жұмыс істейтін жүйенің бір бөлігі ретінде пайдалануы мүмкін. Жалпы айтқанда, қажетті жады көлемінің азаюын болдырмау керек. Көптеген программалаушылар жады көлемінің азаюын салғырттықтан болады деп есептейді. Алайда, бұл көзқарасты біз біржақты деп есептейміз. Егер программа операциялық жүйенің (Unix, Windows немесе басқа) басқаруымен орындалатын болса, программа жұмысы аяқталған соң барлық жады көлемі жүйеге автоматты түрде қайтарылады. Бұдан шығатыны, егер сіздің программаңыз оған бөлінген жады көлемінен артық мөлшерді пайдаланбайтыны белгілі болса, онда операциялық жүйе өзі тәртіп орнатқанша, жады мөлшерінің азаюына (босауына) жол беруге болады. Дегенмен, егер сіз осындай жағдаймен келісетін болсаңыз, онда пайдаланылатын жады көлемін бағалауыңыздың дұрыс екендігіне сенімді болыңыз, әйтпесе сіз ұқыпсыздар санатына қосыласыз.

17.5 Деструкторлар

Енді біз элементтерді векторда қалай сақтау керек екендігін білеміз. Біз жай ғана бос жады көлемінің қажетті мөлшерін бөлеміз де, онымен нұсқауыш арқылы жұмыс істейтін боламыз.

```
// double типті сандардан тұратын өте қарапайым вектор
class vector {
    int sz;                 // өлшемі
    double* elem;           // элементтерге нұсқауыш
public:
    vector(int s)           // конструктор
    :sz(s),                 // sz мүшесін инициалдау
    elem (new double[s]) // elem мүшесін инициалдау
    {
    forinti=0;i<s;++i)elem[i]=0; //элементтерді инициалдау
    }
```

```
    int size() const { return sz; } // ағымдағы өлшем
    // . . .
};
```

Сонымен **sz** мүшесі элементтер санын сақтайды. Біз оны конструкторда инициалдаймыз, ал **vector** класын қолданушы **size()** функциясын шақыра отырып, элементтер санын анықтай алады. Элементтерге арналған жады конструкторда **new** операторының көмегімен бөлінеді, ал **new** операторымен қайтарылған нұсқауыш **elem** мүшесінде сақталады.

Біздің элементтерді өздерінің келісім бойынша қойылған мәндерімен **(0.0)** инициалдайтынымызға назар аударыңыз. Стандартты кітапханадағы **vector** класы дәл осындай әрекетті орындайтындықтан, біз басынан бастап осылай істеуге кірістік.

Өкінішке орай, біз пайдаланатын қарапайым **vector** класында жады көлемін босату орын алады. Ол конструкторда **new** операторының көмегімен элементтер үшін жады көлемін бөледі. 17.4 бөлімде жасалған ережеге сәйкес біз **delete** операторының көмегімен бұл жады көлемін босатуымыз керек. Мысал қарастырайық:

```
void f(int n)
{
    vector  v( n ); // double типіндегі n санға жады бөлу
    // . . .
}
```

f() функциясынан шыққан соң, бос жады аймағында құрылған **v** векторының элементтері өшірілмейді. Біз **vector** класының мүшесі – **clean_up()** функциясын анықтап, оны келесідей түрде шақыра аламыз.

```
void f2(int n)
{
  vector v(n);  // int типіндегі
                // n айнымалылар үшін жады бөлетін
                // векторды анықтаймыз
                // ...v векторын пайдаланамыз . . .
  v.clean_up(); // clean_up() функциясы elem мүшесін жояды
}
```

Мәселеге бұл тұрғыдан келу істі оңтайлы шешуі тиіс еді. Дегенмен бос жадымен байланысты мәселелердің бірі адамдардың **delete** операторын ұмытып кететіндігінде болып отыр. Осындай мәселе **clean_up** функциясымен де туындауы мүмкін, өйткені адамдар мұны да пайдалануды ұмытып кетіп жатады. Біз мұнан гөрі тиімдірек шешім ұсына аламыз. Мұндағы негізгі идея компилятордың

конструктор туралы ғана емес, оған қарама-қарсы бағытта маңызды рөл атқаратын функция туралы да білетіндігінде болып отыр. Мұндай функцияларды *деструктор* **(destructor)** деп атау қисынды болып табылады. Класс объектісін құруда конструктордың шақырылуы анық көрінбейтіндігі сияқты, деструктор да объектінің көріну шеңберінен шығып тұрған кезде деструктор да анық шақырылмайды. Конструктор объектінің дұрыс құрылып, инициалданатындығына кепілдік бере алады. Деструктор оған керісінше объектінің жойылғанша дұрыс тазартылатындығына кепілдік береді. Мысал қарастырайық:

```
// double типті сандарды қамтитын қарапайым вектор
class vector {
    int sz;                 // өлшемі
    double* elem;           // элементтерге нұсқауыш

public:
    vector (int s)                      // конструктор
    :sz(s), elem (new double [s])       // жады бөледі
    {
    for(int i=0;i<s;++i) elem [i]=0;
                          // элементтерді инициалдау
    }
~vector( )                  // деструктор
    { delete [] elem }      // компьютер жадын босату
    . . .
};
```

Сонымен, енді келесі кодты жазуға болады:

```
void  f3(int n)
{
    int* p = new int [n]; // int типті n санға жады бөлеміз
    vector v(n);          // векторды анықтаймыз ( басқа да
                          // int типті n санға жады бөлеміз
                          // ...p және v-ны қолданамыз
    delete [] p ;         // бүтін сандар жиымы жазылған
                          // компьютер жадын босатамыз
}   // vector класы v объектісі жазылған компьютер жадын
    // автоматты түрде босатады
```

delete операторын пайдалану онша ыңғайлы емес, ол әрі қателерге де төтеп бере алмайды! **vector** класы бар болатын болса, **new** операторы арқылы жады бөлу де, функциядан шығу кезінде **delete []** операторының көмегімен компьютер жадын босату да қажет болмай қалады. Мұның бәрін де **vector** класы

өте жақсы атқарады. **vector** класы өз элементтерінен компьютер жадын босату үшін деструкторды шақыруды ешқашан да ұмытпайды.

Мұнда біз деструктордың қолданылуы туралы толық мәлімет бергелі отырған жоқпыз. Тек олардың ресурстармен жұмыс істеу кезінде маңызды рөл атқаратынын айтып өткіміз келеді, алдымен оларды резервке қойып, кейін файлдар, ағындар, бұғаттау тәсілдері және т.с.с. арқылы кері қайтаруға тура келеді. **iostream** ағынының қалайша тазартылғаны естеріңізде ме? Олар буферлерді тазартып, файлдарды жабатын, компьютер жадын босататын және т.б. әрекеттер орындайтын еді. Бұлардың барлығын да олардың деструкторлары жүзеге асыратын. Кез келген ресурстары бар кластың деструкторы болуы тиіс.

17.5.1 Жалпылама нұсқауыштар

Егер класс мүшесінің деструкторы бар болса, онда құрамында осы мүше бар объектіні өшіру кезінде деструктор шақырылады. Мысал қарастырайық:

```
struct Customer {
    string name;
    vector <string> addresses;
    // . . .
};
void some_fct()
{
    Customer fred;
    //fred объектісін инициалдау
    //fred объектісін қолдану
}
```

Біз **some_fct()** функциясынан шығып, **fred** объектісі өзінің көріну аймағынан кеткен кезде ол жойылады; басқаша айтқанда, **name** тіркесі мен **addresses** векторы үшін деструкторлар шақырылады. Бұл өте қажет жағдай, себебі керісінше болса, мәселелер туындауы мүмкін. Кейде бұл былайша көрініс табады: компилятор **Customer** класы үшін мүшелердің деструкторларын шақыратын деструктор жасап шығарады. Компилятор мұндай әрекетті (генерация) жиі орындайды, ол әрі класс мүшелерінің деструкторларын күмәнсіз түрде шақыруға мүмкіндік береді.

Мүшелерге және базалық кластарға арналған деструкторлар туынды класқа жататын (қолданушы анықтаған немесе генерацияланған) деструктордан қосалқы түрде шақырылады. Негізінде, барлық ережелер бір нәрсеге келіп тіреледі: деструкторлар объектіні жою (өшіру) кезінде шақырылады (көріну аймағынан шыққанда, **delete** операторы орындалғанда және т.с.с.).

17.5.2 Деструкторлар және бос жады

Деструкторлар қарапайым боп келеді, алайда олар C++ тілінде программалауда пайдаланылатын көптеген тиімді әдістер үшін негіз бола алады. Мұндағы негізгі идея келесі жағдайларға байланысты:

- Егер функцияға ресурс ретінде қандай да бір объект қажет болса, ол конструкторға жүгінеді;
- Объект өзінің қызмет ету кезеңінде ресурстарды босатып, жаңаларын сұрай алады;
- Объекттің қызмет ету кезеңінің соңында деструктор объект иелік еткен барлық ресурстарды босатады.

Бұның жиі кездесетін мысалы ретінде бос жады көлемін басқаратын `vector` класындағы "конструктор-деструктор" жұбын айтуға болады. Бұл тақырыпқа біз 19.5 бөлімде қайта оралатын боламыз. Ал әзірге бос жады көлемін басқару механизмі мен кластар иерархиялары үйлесімін қарастырайық.

```
Shape* fct()
{
   Text tt(Point(200,200),"Annemarie");
   // . . .
   Shape* p = new Text(Point(100,100),"Nicholas");
   return p;
}

void f()
{
   Shape* q = fct();
   // . . .
   delete q;
}
```

Бұл код логикаға сыйымды боп көрінеді және ол солай да. Бәрі жұмыс істейді, бірақ оның қалай жұмыс істейтініне қараңыз, ол код, шынында да, икемді, маңызды және қарапайым тәсілдің мысалы болып саналады. `fct()` функциясынан шығарда, оның құрамындағы `text()` класына жататын `tt` объектісі жойылады. `Text` класының міндетті түрде деструктор шақыруға арналған `string` типті мүшесі болады. `string` класының компьютер жадында орналасуы мен оны босатуы `vector` класына ұқсас. `tt` объектісі үшін бұл жеңіл орындалады, компилятор `text` класы үшін 17.5.1 бөлімінде сипатталғандай түрде құрылған (генерацияланған) деструкторды шақырады. Ал, `Text` класындағы `fct()` функциясы қайтаратын объект туралы не айтуға болады? Шақырушы `f()` функциясы `q` нұсқауышының

Text класындағы объектіге сілтеме жасайтындығынан хабарсыз, оған нұсқауыштың тек **Shape** класындағы объектіге сілтейтіндігі ғана белгілі. **delete p** нұсқауы **Text** класындағы деструкторды қалай шақыра алады?

14.2.1 бөлімінде біз **Shape** класының деструкторы болатындығы туралы ескерткен едік. Негізінде, **Shape** класында виртуалды деструктор бар. Бар мәселе осында. Біз **delete p** нұсқауын орындағанда, деструкторды шақырудың қаншалықты қажет екендігін білу үшін **delete** операторы **p** нұсқауышының типіне талдау жасайды да, керек деп тапса, оны шақырады. Сонымен, **delete p** нұсқауы **Shape** класына жататын **~Shape()** деструкторын шақырады. Дегенмен, **~Shape()** деструкторы виртуалды болғандықтан, виртуалды функцияны шақыру механизмінің көмегімен (17.3.1. тарауын қара.) ол **Shape** класының туындысы саналатын класс деструкторын шақырады, мұнда ол – **~Text()** деструкторы. Егер **Shape::~Shape()** деструкторы виртуалды болмаса, **Text::~Text()** деструкторы шақырылмас еді және **string** типті **Text** класының мүшесі де дұрыс жойылмас еді.

Мынадай ережені есте сақтаңыз: егер кластың **виртуалды** функциясы болса, онда оның құрамында **виртуалды** деструктор да болуы тиіс. Оның себептері мынада:

1. Егер кластың **виртуалды** функциясы болса, ол базалық функция ретінде қолданылуы тиіс;
2. Егер класс базалық болса, оның оның туынды класы **new** операторын қолдануы тиіс;
3. Егер туынды кластың объектісі компьютер жадында **new** операторының көмегімен орналасатын болса, ал онымен істелетін жұмыс базалық класқа сілтеме арқылы жүзеге асатын болса, онда ол базалық класс объектісіне сілтеме жасайтын нұсқауыш арқылы **жойылады**.

Деструкторлардың **delete** операторының көмегімен тікелей емес немесе жанамалы түрде шақырылатынына назар аударыңыз. Олар ешқашанда тікелей шақырылмайды. Бұл көп еңбек етуді қажет ететін жұмыстан құтылуға мүмкіндік береді.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Өздерінің шақырылғандығы туралы ақпарат бере алатын конструкторлары мен деструкторлары анықталған базалық кластар мен мүшелерді пайдалана отырып кішігірім программа жазыңыз. Содан кейін бірнеше объект құрыңыз да, конструкторлар мен деструкторлардың қалай шақырылатынын қараңыз.

17.6 Элементтермен қатынас құру

Бізге **vector** класымен жұмыс істеу ыңғайлы болуы үшін, элементтерді жазуға және оқуға тиіспіз. Алдымен қарапайым **get()** және **set()** функция-мүшелерін қарастырайық:

```
// double типіндегі сандардың өте қарапайым векторы
class vector {
    int sz;         // өлшемі
    double* elem;   // элементтерге нұсқауыш
public:
    vector(int s) :sz(s), elem(new double[s]) { } // конструктор
    ~vector() { delete[] elem; }                  // деструктор
int size() const { return sz; }                 // ағымдағы өлшем

    double get(int n) { return elem[n];}   // қатынасу: оқу
    void set(int n, double v) { elem[n]=v; } // қатынасу: жазу
};
```

get() және **set()** функциялары **[]** операторын **elem** нұсқауышына қолдану арқылы оның элементтерімен **elem[n]** қатынасуды (оларды пайдалануды) қамтамасыз етеді.

Енді біз **double** типті сандардан тұратын вектор жасап, оны пайдалана аламыз:

```
vector v(5);
for(int i=0; i<v.size(); ++i) {
     v.set(i,1.1*i);
     cout << "v[" << i << "]==" << v.get(i) << '\n';
}
```

Мұның нәтижесі мынадай болады:

```
v[0]==0
v[1]==1.1
v[2]==2.2
v[3]==3.3
v[4]==4.4
```

vector класының бұл нұсқасы өте қарапайым, ал **get()** және **set()** функцияларын пайдаланатын код тік жақшалар негізінде әдеттегі элементтерді пайдалану тәсілімен салыстырғанда, онша әдемі емес. Дегенмен, біздің мақсатымыз шағын әрі қарапайым нұсқадан бастап, оны әр кезеңде тесттен өткізе отырып, біртіндеп дамытып жетілдіру болып табылады. Осындай біртіндеп кеңіту мен

тұрақты тесттен өткізу стратегиясы жіберілетін қателер саны мен программаны жөндеп түзету мерзімін де азайтады.

17.7 Класс объектілеріне нұсқауыштар

Нұсқауыш ұғымы әмбебап сипатта болғандықтан, біз оны компьютер жадының кез келген ұясына орната аламыз. Мысалы, біз **vector** класының объектісіне нұсқауыштарды **char** типіндегі айнымалылар нұсқауыштары сияқты қолдана аламыз:

```
vector*  f(int s)
 {
   vector* p = new vector (s);
   //векторды бос жадыға орналастырамыз, *p толтырамыз
   return p;
 }
 void ff()
 {
   vector* g = f(4); // *g  нұсқауышын қолданамыз
   delete g;          // бос жады аймағынан векторды өшіреміз
 }
```

Егер біз vector класының объектісін **delete** операторының көмегімен өшіретін болсақ, онда оның деструкторы шақырылатынына назар аударыңыз. Мысал қарастырайық:

```
vector* p = new vector(s);  // бос жады аймағына
                            // векторды орналастырамыз
delete p;          // бос жады аймағынан векторды өшіреміз
```

Бос жады аймағында **vector** класының объектісін құрған кезде, **new** операторы келесі әрекеттерді орындайды:

- алдымен **vector** класының объектісі үшін жады бөледі;
- сонан соң **vector** класының конструкторын шақырады, ол оның объектісін инициалдау үшін қажет; бұл конструктор **vector** класы объектісінің элементтері үшін компьютер жадын бөліп, оларды инициалдайды.

vector класының объектілерін өшіре отырып, **delete** операторы келесі әрекеттерді орындайды:

- алдымен **vector** класының деструкторын шақырады, бұл деструктор, өз

кезегінде, элементтер (олар бар болса) деструкторын шақырады, сонан соң вектор элементтері орналасқан жады аймағын босатады;

- `vector` класының объектісі орналасқан жады көлемін босатады.

Осы операторлардың рекурсивті түрде жақсы жұмыс істейтініне назар аударыңыз (8.5.8 бөлімін қ.). Нақты (стандартты) `vector` класын қолдана отырып, келесі кодты орындай аламыз:

```
vector<vector<double>>* p = new vector<vector<double>>(10);
delete p;
```

Бұл жерде `delete p` нұсқауы `vector<vector<double>>` класының деструкторын шақырады, ал ол өз кезегінде `<vector<double>` класы элементтерінің деструкторын шақырады да, вектор түгелдей босатылады, бір де бір объект жойылмай қалмайды және жады көлемі кемімейді.

`delete` операторы деструкторларды шақыратын болғандықтан (олар қарастырылған типтер үшін, мысалы, `vector` сияқты типтерде), көбінесе ол объектілерді жояды (`destoy`), оларды компьютер жадынан өшірмейді (`dellocate`) деп айтылады.

Әрқашанда мынаны есте сақтаған жөн: жай ғана бос жазылған `new` операторы конструктор сыртында `delete` операторын ұмытып кету қаупін тудырады. Егер сізде объектілерді жоюдың жақсы стратегиясы болмаса (ол нақты түрде қарапайым, мысалы, 13.10 және Д4 бөлімдеріндегі `vector_ref` класын қ.), онда `new` операторын конструкторға, ал `delete` операторын деструкторға қосып көріңіз.

Сонымен, бәрі де дұрыс сияқты, бірақ тек нұсқауышты ғана қолдана отырып, `вектор` мүшелеріне қалай қол жеткізе аламыз?

Барлық кластар өздерінің мүшелеріне объект атына жалғанған `.` (нүкте) операторы арқылы қол жеткізе (пайдалана) алатынына назар аударыңыз.

```
vector v(4);
int x = v.size();
double d = v.get(3);
```

Осылайша барлық кластар өздерінің мүшелеріне объектіге нұсқауыш арқылы қол жеткізу үшін, `->` (тілсызық) операторының жұмысын сүйемелдейді.

```
vector* p = new vector (4);
int x = p->size();
double  d = p->get(3);
```

`.` (нүкте) операторы сияқты `->` (тілсызық) операторын мәлімет-мүшелерге және функция-мүшелерге қол жеткізу үшін қолдануға болады. `Int` және `double`

сияқты құрамдас типтердің мүшелері жоқ болғандықтан, оларға `->` (тілсызық) операторы қолданылмайды. Көбінесе осы "нүкте" және "тілсызық" операторларын класс мүшелеріне *қол жеткізу операторы* (member access operators) деп атайды.

17.8 Типтермен шатасу: void* және типтерді келтіру операторлары

Компьютердің бос жадында орналасқан жиымдар мен нұсқауыштарды пайдалана отырып, біз аппараттық жабдықтамамен тығыз байланысқа түсеміз. Негізінде біздің нұсқауыштармен орындайтын операцияларымыз (инициалдау, меншіктеу, `*` және `[]`) тікелей машиналық нұсқауларда көрсетіледі. Бұл деңгейде тіл программалаушыларға типтер жүйесі арқылы қарастырылатын мүмкіндіктерді толық бермейді. Бірақ кей жағдайда олардағы қателердің қауіпсіздік деңгейі төмен болуына қарамастан, бас тартуға тура келеді.

Әрине, типтер жүйесі арқылы көрінетін қауіпсіздіктен біздің тіпті де бас тартқымыз келмейді, бірақ кей кезде бізде басқа логикалық балама жол (альтернатива) болмай қалады (мысалы, C++ тіліндегі типтер жайлы ешқандай мәліметі жоқ басқа тілде жазылған программа жұмысын қамтамасыз ету керек болған кезде). Сонымен бірге, статикалық типтердің қауіпсіздік жүйесін есепке алмай жазылған ескі программаларды қолдануға қажетті көптеген жағдайлар да кездесіп жатады.

Осындай сәттерде бізге екі нәрсе қажет болады:

- өзінде сақталып тұрған объектілер жөнінде ақпарат бермейтін жады аймағына сілтеме жасайтын нұсқауыш типі;
- осындай нұсқауыш көмегімен жады ұяшығына сілтеме жасалған кезде қандай мәліметтер типі (тексерусіз) қолданылатынын компиляторға хабарлайтын операция.

`void*` типі "сақталатын мәлімет типі компиляторға белгісіз болып тұрған жады ұяшығына нұсқауыш" деген мағынаны білдіреді. Бұл мүмкіндік программаның бір бөлігінен екінші бөлігіне адресті жіберу кезінде қолданылады, мұнда олардың әрқайсысы басқа бөліктің қандай типтегі объектімен жұмыс жасайтынын білмейді. Бұған мысал ретінде кері шақыру функцияларының аргументтері қызметін (16.3.1 бөлімін қ.) атқаратын адресті алуға және де ең төменгі деңгейдегі компьютер жадын бөлу (`new` операторын жүзеге асыру сияқты) ісін де жатқызуға болады.

`void` типті объектілер болмайды, бірақ `void` типі "функция ешқандай мән қайтармайды" деген түсінік береді.

```
void v;          // қате: void типті объектілер жоқ
void f();        // f() функциясы ешқандай мән қайтармайды
                 // бұл f() функциясы void типіндегі
                 // объекті қайтарады деген сөз емес
```

void* типіндегі нұсқауышқа кез келген типтегі объектіге нұсқауышты меншіктеуге болады. Мысал қарастырайық:

```
void* pv1 = new int; // OK: int* типі void* типіне айналады
void* pv2 = new double[10]; // OK: double* типі
                            // void* айналады
```

Компилятор **void*** типті нұсқауыштың қандай мәліметке сілтеме жасап тұрғанын білмейтіндіктен, біз оған осы туралы хабарлауға тиістіміз.

```
void f(void* pv)
{
   void* pv2 = pv; // дұрыс, void* осындай мақсатқа арналған
   double* pd = pv; // қате: void* типін double* типіне
                    // келтіруге болмайды
   *pv= 7;     // қате: void* типін атаусыз етуге болмайды
   // (нұсқауыш сілтеме жасап тұрған объект типі белгісіз)
   pv[2] = 9; // қате: void* типін индекстеуге болмайды
   int* pi= static_cast<int*>(pv);
   // OK: тікелей түрде типке келтіру
   // . . .
}
```

static_cast операторы нұсқауыштар типін соған жақын (туыс) типке тікелей түрлендіруге мүмкіндік береді, мысалы **void*** типін **double*** типіне (A.5.7 бөлімі). **static_cast** атауы тек ең ақырғы қажеттілік туғанда ғана қолдануға тура келетін жағымсыз атау және ол жағымсыз (қауіпті) операторлар үшін саналы түрде таңдалып алынған болатын. Ол программаларда өте сирек кездеседі (егер ол бір жерлерде пайдаланылатын болса ғана). **static_cast** сияқты операцияларды *типті тікелей түрлендіру* (explicit type conversion) деп атайды немесе жай ғана "келтіру" деп атайды, себебі C++ тілінде типтерді келтіру үшін екі оператор қарастырылған, олар **static_cast** операторынан да келеңсіздеу болып келеді:

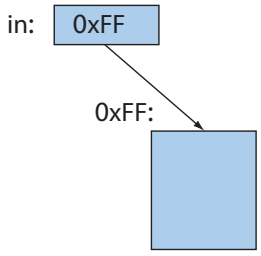
- **reinterpret_cast** операторы типті өзімен байланысты емес, мүлде басқа типке түрлендіруі мүмкін, мысалы int типін **double*** типіне түрлендіре алады.
- **const_cast** операторы **const** квалификаторын алып тастау мүмкіндігін береді.

Мысал қарастырайық:

```
Register* in = reinterpret_cast<Register*>(0xff);
```

```
void f(const Buffer* p)
{
    Buffer* b = const_cast<Buffer*>(p);
    // . . .
}
```

Алғашқы мысал – **reinterpret_cast** операторын қолдануды қажет ететін классикалық жағдай. Біз компиляторға жадының белгілі бір бөлігі (**0xFF** ұяшығынан басталатын бөлігі) **Register** (мүмкін арнайы семантика арқылы) класының объектісі ретінде қарастырылатынын хабарлаймыз. Мұндай код, мысалы, құрылғылардың драйверлерін жасау кезінде қажет болады.



Екінші мысалдағы **const_cast** операторы **p** нұсқауышындағы const **Buffer*** жариялануының **const** квалификаторын жояды. Әрине, біз бұларды не істеу керек екендігін түсіне отырып жасадық.

Негізінде, **static_cast** операторы нұсқауыштарды бүтін сандарға түрлендіруге немесе **const** квалификаторын жоюға мүмкіндік бермейді, сол себепті бір типті басқа типке түрлендіру кезінде **static_cast** операторын қолданған тиімді болады. Егер сізге типтерді түрлендіру қажет болып жатса, тағы да ойланыңыз: программаны типтерді келтірмей-ақ жазуға болмас па екен? Программаны типтер келтіру қажет болмайтындай етіп қайта жазып шығуға болады ма? Егер сізге басқалар жазған кодты қолдану немесе аппараттық жабдықтамаларды басқару қажет болмаса, онда **static_cast** операторын қолданудың мүлде қажеті жоқ. Кері жағдайда, анықталуы қиынға соғатын және де қауіпті қателер туындауы мүмкін. Егер сіз **reinterpret_cast** операторын қолданатын болсаңыз, онда сіздің программаңыз басқа компьютерде еш қиындықсыз жұмыс істейді деп күтуге болмайды.

17.9 Нұсқауыштар мен сілтемелер

Сілтемені (reference) автоматты түрде атаусыз етілетін (есімсізденетін) тұрақты нұсқауыш ретінде түсіндіруге немесе объектінің альтернативті атауы ретінде қарастыруға болады. Нұсқауыштар мен сілтемелер келесі айырмашылықтары бойынша ажыратылады:

- Нұсқауышқа басқа мән меншіктеген кезде нұсқауыш орнатылған объектінің емес, тек нұсқауыштың мәні өзгереді;
- Нұсқауышқа қол жеткізу үшін көбінесе **new** немесе **&** операторын пайдалану керек;
- Нұсқауыш орнатылған объектіге қол жеткізу үшін * немесе [] операторлары қолданылады;
- Сілтемеге басқа мән тағайындау, сілтеменің өзін емес сілтеме жасалған объектіні өзгертеді;
- Инициалданғаннан кейін сілтемені басқа объектіге орната алмаймыз;
- Сілтемелерді меншіктеу тереңдетіп көшіру әрекетіне негізделген (жаңа мән сілтеме нұсқап тұрған объектіге меншіктеледі); нұсқауыштарды меншіктеу тереңдетіп көшіруді (жаңа мән объектіге емес, нұсқауышқа меншіктеледі) қолданбайды;
- Нөлдік нұсқауыштар қауіпті болып табылады.

Мысал қарастырайық:

```
int  x = 10;
int* p = &x;  // нұсқауышты алу үшін & операторы қажет
*p = 7;       // p нұсқауышы  арқылы x айнымалысына
              // мән беру үшін * қолданылады
int x2 = *p; // x айнымалысын p нұсқауышы арқылы оқимыз
int* p2 = &x2; // int типті басқа айнымалыға нұсқауыш аламыз
p2 = p;       // p2 және p нұсқауыштары x айнымалысына
              // сілтеме жасайды
p = &x2;      // p нұсқауышы басқа объектіге сілтеме жасайды
```

Сілтемеге қатысты осыған эквивалентті Мысал келтірейік:

```
int y = 10;
int& r = y;  // & символы инициализаторды емес, типті білдіреді
r = 7;       // r сілтемесі арқылы (* операторы керек емес)
             // y айнымалысына мән беру
int y2 = r;// r сілтемесі арқылы (* операторы керек емес)
             // y айнымалысын оқимыз
int& r2 = y2;// int типті басқа айнымалыға сілтеме
r2 = r;      // y айнымалысының мәні y2
             // айнымалысына  тағайындалады
r = &y2;     // қате: сілтеме мәнін өзгертуге болмайды
// (int& сілтемесіне int* айнымалысын меншіктеуге болмайды)
```

Соңғы мысалға назар аударыңыз: бұл тек мұндай конструкция жұмыс істей алмайды деген сөз емес, инициалдаудан соң сілтемені басқа объектімен байланыстыруға болмайды. Егер сізге басқа объектіні көрсету қажет болып жатса, нұсқауышты қолданыңыз. Нұсқауыштарды пайдалану 17.9.3-бөлімінде сипатталған.

Сілтеме де, нұсқауыш та компьютер жадын адрестеуге негізделген, бірақ олар программалаушыға әртүрлі мүмкіндіктер береді.

17.9.1 Нұсқауыштар мен сілтемелер функция параметрлері ретінде

Егер айнымалының мәнін функция арқылы есептелген мәнге өзгерту керек болса, онда оның үш түрлі жолы бар. Мысал қарастырайық:

```
int incr_v(int x) {return x+1;}   // жаңа мәнді есептейді
                                  // және қайтарады
void incr_p(int* p) {++*p;}       // нұсқауышқа береді
        // (оны атаусыз етеді де, мәнін бірге арттырады)
void incr_r(int& r) {++r;}        // сілтемені береді
```

Қандай таңдау жасар едіңіз? Әрине, бірінші нұсқаны, яғни мәнді қайтару жолын таңдар едіңіз (қате тез пайда болатын жол).

```
int x=2;
x = incr_v(x);              // x-ті incr_v()-ға көшіреміз;
// сонан соң нәтижені көшіреміз және оны қайтадан меншіктейміз
```

Бұл стиль **int** типті айнымалылар сияқты онша үлкен емес объектілер үшін тиімді болады. Бірақ оған мәндерді жіберу және қайтару көбінесе дұрыс орындалмайды. Мысалы, **int** типті 10 мың айнымалыдан тұратын вектор сияқты көлемді мәліметтер құрылымын сипаттайтын функцияны жазуға болады; біз осындай 40 мың байтты (минималды түрде, екі есе) жеткілікті тиімділікпен көшіре алмаймыз.

Аргументті сілтеме немесе нұсқауыш бойынша жіберу арасында қандай таңдау жасауға болады? Өкінішке орай, осы екі нұсқаның өз артықшылықтары мен кемшіліктері бар, сол себепті оған нақты жауап та беруге болмайды. Әрбір программалаушы жағдайға байланысты өзі шешім қабылдауы керек.

Аргументті сілтеме бойынша жіберуді қолдану программалаушыны алаңдатады, өйткені мәннің өзгеріп кетуі де мүмкін. Мысал қарастырайық:

```
int x = 7;
incr_p (&x);              // бұл жерде & операторы қажет
incr_r(x);
```

`incr_p(&x)` функциясын шақыру кезінде `&` операторын қолдану қажеттілігі қолданушыға `x` айнымалысының өзгеріп кетуі мүмкін екендігін білдіреді. Бұған қарама-қарсы `incr_r(x)` функциясын шақыру "күнәсіз болып көрінеді". Бұл нұсқауыш арқылы берудің аздаған артықшылығын білдіреді.

Басқа жағынан алғанда, егер функция аргументінің орнына нұсқауышты қолданатын болсақ, онда функцияға нөлдік нұсқауыш, яғни нөлдік мәні бар нұсқауыш жіберілмеуін қадағалау қажет. Мысал қарастырайық:

```
incr_p(0);      // апатты жағдай: incr_p() функциясы нөлді
                                  // атаусыз етуге талпынады
int* p = 0;
incr_p(p);      // апатты жағдай: incr_p() функциясы нөлді
                                  // атаусыз етуге талпынады
```

Бұлардың дұрыс емес екені айдан анық көрінеді. `incr_p()` функциясын жазған адам қорғанысты да қарастыра алады.

```
void incr_p(int* p)
{
  if(p==0)
    error("incr_p() функциясына нөлдік нұсқауыш берілген");
  ++p*;         // нұсқауышты атаусыз етеміз де,
                // ол орнатылған объектіні 1-ге арттырамыз
}
```

Енді `incr_p()` функциясы қарапайым және тиімді болып көрінеді. 5-тарауда қателік кеткен аргументтерге байланысты мәселенің қалай шешілетіні көрсетілген болатын. Осыған қарама қарсы жағдайда сілтемелерді (мысалы, `incr_r()` функциясында) пайдаланатын қолданушылар сілтемені объектімен байланысты деп есептеуі керек.

Егер "бостан бос жіберуді" (негізінде объект жіберілмейді) функция семантикасы тұрғысынан қарастырғанда, дұрыс деп қабылдағанмен де, аргументті нұсқауыш арқылы жіберу керек. Ескерту: бұл жағдай инкрементация операциясына қатысты емес, себебі `p==0` болған жағдайда, аластама (exception) орын алуы тиіс.

Сонымен, дұрыс жауап былай айтылады: таңдау функция табиғатына тәуелді болады.

- Шағын объектілер үшін аргументті мән бойынша беру тиімді болып табылады;
- Өзінің аргументі ретінде "нөлдік объект" бере алатын (`0` мәнімен көрсетілген) функция үшін нұсқауышты беру тәсілін қолдану қажет (**нөлді** тексеруді де ұмытпау керек);
- Кері жағдайда параметр ретінде сілтемені қолдануға болады.

8.5.6-бөлімді де қарау керек.

17.9.2 Нұсқауыштар, сілтемелер және мұралау

14.3-бөлімде біз **Circle** сияқты туынды класты оның **Shape** ашық базалық класы объектісінің орнына қалай пайдалануға болатынын көрдік. Бұл идеяны нұсқауыштар мен сілтемелер терминінде өрнектеуге болады: **Circle*** нұсқауышын **Shape*** нұсқауышына қосалқы түрде түрлендіруге болады, өйткені **Shape** класы **Circle** класына қатысты ашық базалық класс болып табылады. Мысал қарастырайық:

```
voidrotate(Shape*s,intn); //*s фигурасын n бұрышқа бұрамыз

Shape* p = new Circle (Point (100,100),40);
Circle c (Point (200,200),50);
rotate(&c,45);
```

Мұны сілтеме көмегімен де жасауға болады.

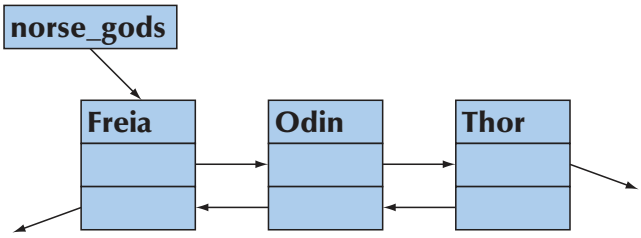
```
voidrotate(Shape&s,intn); //*s фигурасын n бұрышқа бұрамыз

Shape& r = c;
rotate (c,75);
```

Бұл факт программалаудың объектіге бағытталған технологиясының басым бөлігі үшін өте маңызды болып табылады (14.3, 14.4-бөлімдерді қ.).

17.9.3 Мысал: тізімдер

Ең көп таралған және тиімді мәліметтер құрылымы болып тізімдер саналады. Көбінесе тізім түйіндер арқылы құрылады, олардың әрқайсысының белгілі бір ақпараты және басқа түйіндерге нұсқауыштары бар. Бұл нұсқауыштарды қолданудың классикалық мысалы болады. Мысалы, норвегия құдайларының қысқаша тізімін келесі түрде көрсетуге болады.



Мұндай тізімді *қос байланысты тізім* (doubly-linked list) деп атайды, өйткені бұл тізімде өзіне дейінгі және кейінгі түйіндер бар. Тек қана кейінгі түйіндерден

ғана тұратын тізімді *бір байланысты тізім* (singly-linked list) деп атайды. Біз екі байланысты түйіндерді элементті өшіруді жеңілдету үшін қолданамыз. Тізім түйіндері келесі түрде анықталады:

```
struct Link {
    string value;
    Link* prev;
    Link* succ;
    Link(const string& v, Link* p = 0, Link* s=0)
        : value(v),prev(p),succ(s){ }
};
```

Басқаша айтқанда, **Link** типті объектіге ие бола отырып, біз **succ** нұсқауышын қолданып келесі элементке қол жеткізе аламыз, ал алдыңғы элементке **prev** нұсқауышын қолдана отырып қол жеткізуге болады. Нөлдік нұсқауыш түйіннің алдыңғы және кейінгі түйіндері жоқ екенін көрсетуге мүмкіндік береді. Норвегия құдайлары тізімін былай жазуға болады:

```
Link* norse_gods = new Link("Thor",0,0);
norse_gogs = new Link("Odin",0,norse_gods);
norse_gogs->succ->prev = norse_gogs;
norse_gogs = new Link("Freia",0,norse_gogs);
norse_gogs->succ->prev = norse_gogs;
```

Біз бұл тізімді **Link** құрылымының көмегімен құрдық: тізімнің басында Тор тұрады, одан кейін Один (ол Тордың алдында болған), ал тізімді Одиннің алдындағы Фрея аяқтайды. Нұсқауыштарға сүйене отырып, біздің дұрыс жасағанымызды көруге болады, әрбір **succ** және **prev** нұсқауыштары дұрыс бағытта сілтеме жасайды. Бірақ бұл код онша түсінікті емес, себебі біз кірістіру операциясын тікелей түрде анықтамадық және оған ат меншіктемедік.

```
Link* insert(Link* p, Link* n)     //n-ді p алдына кірістіру
                                                (фрагмент)
{
  n->succ = p;          //p n-нен кейін орналасқан
  p->prev->succ=n;      //np-ның алдындағысынан кейін орналасқан
  n->prev = p->prev;    //p-ның алдындағысы n-нің алдындағысы
                        // болады
  p->prev = n;          // n p-ның алдындағысы болады
  return n;
}
```

Бұл программаның фрагменті p нұсқауышы нақты түрде **Link** типті объектіге сілтеме жасаса және бұл объектінің алдында да объект бар болса ғана жұмыс істейді. Бұлардың осылай істейтініне көз жеткізу керек. Нұсқауыштар мен байланысқан құрылымдар туралы сөз болғанда, әсіресе **Link** типті объектілерден тұратын тізімдерде, біз әрқашан қағазда төртбұрыштар мен бағыттауыштардан тұратын диаграмма сызамыз, осылай программаны шағын мысалдарда тексеруге болады.

Келтірілген **insert()** функциясының нұсқасы толық емес, өйткені онда **n, p** немесе **p->prev** нұсқауыштары нөлге тең болатын жағдай қарастырылмаған. Осыған сәйкес тексеру енгізіп, шамалы күрделі, бірақ **insert** функциясының дұрыс нұсқасын аламыз.

```
Link*insert(Link*p,Link*n) //p-ныңалдынаn-дікірістіреді,
                           // n-ді қайтарады
{
    if(n==0) return p;
    if(p==0) return n;
    n->succ = p;              // p n-нен кейін тұр
    if(p->prev) p->prev->succ = n;
    n->prev = p->prev;        // p-ның алдындағысы n-нің
                              // алдындағысы болады
    p->prev = n;              // n p-ның алдындағысы болады
    return n;
}
```

Мұндай жағдайда біз мынадай код жаза аламыз:

```
Link* norse_gods = new Link("Thor");
norse_gods = insert(norse_gods,new Link("Odin"));
norse_gods = insert(norse_gods,new Link("Freia"));
```

Енді **prev** және **succ** нұсқауыштарына қатысты барлық қиындықтар шешілді. Нұсқауыштардың дұрыстығын тексеру өте шаршатады және ол қателіктерге төтеп бере алмайды, сол себепті оны *міндетті түрде* жақсы жобаланған және нақты тексерілген функцияларда жасыру қажет. Жеке жағдайда, программадағы көптеген қателер программалаушылардың нұсқауыштың нөлге тең екенін тексеруді ұмытып кетуінен болып жатады, мұндай жағдай **insert()** функциясының бірінші нұсқасында әдейі көрсетілген еді.

Конструкторды әрбір шақырған сайын қолданушыларды алдыңғы және кейінгі элементтерді көрсетуден босату үшін келісім бойынша аргументтерді (15.3.1, A.9.2 бөлімдерін қ.) қолданғанымызға назар аударыңыздар.

17.9.4 Тізімдермен орындалатын операциялар

Стандартты кітапхананың 20.4 бөлімінде сипатталатын **list** класы бар. Оның ішінде қажетті барлық операциялар іске асырылған, бірақ біз өзіміз стандартты тізімнің "қоршауында" не жасырылғанын білу үшін және нұсқауыштарды пайдаланудың бірнеше мысалын көрсету үшін бұл бөлімде **Link** класына негізделген тізім жасаймыз.

Нұсқауыштарға байланысты қателерден қорғану үшін қолданушыға қандай операциялар қажет? Әрине, мұны әркім өзі шешеді, дегенмен біз пайдалы бір мүмкіндіктерді келтіре кетейік:

- Конструктор;
- **insert:** элементке дейін кірістіру;
- **add:** элементтен кейін кірістіру;
- **erase:** элементті өшіру;
- **find:** берілген мәні бойынша түйінді іздеу;
- **advance:** келесі **n**-ші түйінге өту.

Бұл операцияларды мынандай түрде жазып шығуға болады:

```
Link* add(Link* p, Link* n)    // p-дан кейін n кірістіріледі;
                               // n қайтарылады
{
    // insert-ті еске түсіреді (11-жаттығуға қ.)
}

Link* erase(Link* p)          // *p түйінін тізімнен өшіреді;
                              // p-дан кейінгіні қайтарады
{
    if(p==0) return 0;
    if(p->succ) p->succ->prev = p->prev;
    if(p->prev) p->prev->succ = p->succ;
    retutn p->succ;
}

Link* find(Link* p, const sring& s) // s-ті тізімнен табады;
                                    // таппаса, 0 қайтарады
{
    while(p) {
      if(p->value == s) return p;
```

```
        p = p->succ;
        }
          return 0;
}

Link*advance(Link*p,intn)  // тізімненnпозицияныөшіреді,
                           // егер таппаса, 0 қайтарады
   // n оң болса, нұсқауышты n түйінге алға жылжытады,
   // n теріс болса, нұсқауышты n түйінге кейінге жылжытады,
{
   if(p==0) return 0;
   if(0<n) {
      while(n--){
         if(p->succ == 0) return 0;
         p = p->succ;
      }
   }
   if(n<0){
      while(n++){
         if(p->prev==0) return 0;
         p = p->prev;
      }
   }
   return p;
}
```

n++ постфикстік инкрементациясының қолданылып тұрғанына назар аударыңыздар. Ол алдымен айнымалының ағымдағы мәні қолданылатынын, содан соң барып оның бірге артатынын білдіреді.

17.9.5 Тізімдерді пайдалану

Шағын мысал ретінде екі тізім жасайық:

```
Link* norse_gods = new Link("Thor");
norse_gods = insert (norse_gods, new Link("Odin"));
norse_gods = insert (norse_gods, new Link("Zeus"));
norse_gods = insert (norse_gods, new Link("Freia"));

Link* greek_gods = new Link("Hera");
greek_gods = insert(greek_gods, new Link("Athena"));
greek_gods = insert(greek_gods, new Link("Mars"));
greek_gods = insert(greek_gods, new Link("Poseidon"));
```

Өкінішке орай, біз көп қате жібердік: Зевс – норвегтердің емес, грек құдайы, гректің соғыс құдайы – Марс емес, Арес (Марс оның римдік аты). Бұл қателерді келесі жолмен түзеуге болады:

```
Link* p = find(greek_gods, "Mars");
if(p) p->value = "Area";
```

`find()` функциясының **нөл** мәнін қайтаратынын тексеретінімізге назар аударыңыз. Біз, әрине, ондай жағдайдың болмайтынына сенімдіміз (негізінде, біз жаңа ғана Марс атын `greek_gods` тізіміне кіргіздік), бірақ іс жүзінде жағдай біз күткендей болмай қалуы да мүмкін ғой.

Жоғарыдағыдай, Зевсті грек құдайларының дұрыс тізіміне ауыстыра аламыз:

```
Link* p = find(norse_gods,"Zeus");
if(p) {
    erase(p);
    insert(greek_gods, p);
}
```

Сіз қатені байқадыңыз ба? Ол соншалықты елеусіз (әрине, егер сіз тізімдермен тікелей жұмыс істемейтін болсаңыз). Ал егер `erase(p)` функциясы арқылы босатылған түйінге `norse_gods` тізімінің бірнеше түйіндері сілтеме жасаған болса ше? Әрине, негізінде бұлай болған жоқ, бірақ өмірде бәрі де мүмкін, сондықтан жақсы программа мұны есепке алып отыруы тиіс.

```
Link* p = find(norse_gods, "Zeus");
if(p)
{
    if(p == norse_gods) norse_gods = p->succ;
    erase(p);
    greek_gods = insert(greek_gods,p);
}
```

Біз Зевсті алғашқы грек құдайының алдына қойып екінші қатені де жөндедік, біз оған тізім нұсқауышын орналастыруымыз керек. Нұсқауыштар – өте пайдалы және икемді құрал, бірақ тым сезімтал құрал. Қорытындылай келе, тізім басып шығарайық:

```
void print_all(Link* p)
{
    cout << "{";
    while(p) {
      cout << p->value;
      if(p = p->succ) cout << ",";
    }
    cout <<"}";
}
```

```
print_all(norse_gods);
cout<<"\n";

print_all(norse_gods);
cout<<"\n";
```

Нәтиже келесі түрде болуы керек:

```
{Freia, Odin, Thor}
{Zeus, Poseidon, Ares, Athena, Hera}
```

17.10 this нұсқауышы

Тізіммен жұмыс істейтін әрбір функция алғашқы аргумент ретінде сол объектіде сақталатын мәліметтерге қол жеткізу үшін **Link*** нұсқауышын алатынына назар аударыңыз. Мұндай функциялар көбінесе класс мүшелері болып табылады. Өзіне сәйкес класс мүшелерін қарастыра отырып, **Link** класын қарапайым етуге (немесе тізімді пайдалануға) бола ма?

Мүмкін нұсқауыштарды жабық етіп жасап, оларды тек кластың функция-мүшелері ғана пайдаланатындай етіп көру керек шығар? Қарастырып көрейік:

```
class Link {
public:
    string value;
    Link(const string& v, Link* p = 0, Link* s = 0)
        : value(v), prev(p), succ(s){}

    Link* insert(Link* n);       // n-ді берілген объектінің
                                 // алдына кірістіреді
    Link* add(Link* n);          // n-ді берілген объектіден
                                 // кейін кірістіреді
    Link* erase();  // берілген объектіні тізімнен өшіреді
    Link* find (const string& s); // тізімнен s-ті табады
    const Link* find(const string& s) const;
                                  // тізімнен s-ті табады

    Link* advance(int n) const; // тізімнен n позицияны өшіреді

    Link* next() const {return succ;}
    Link* previous() const {return prev;}
private:
    Link* prev;
    Link* succ;
};
```

Бұл фрагмент өте тиімді болып көрінеді. Біз тұрақты функция-мүшелердің көмегімен **Link** класы объектісінің қалпын өзгертпейтін операцияларды анықтадық. Қолданушылар тізімде орын ауыстыра алуы үшін біз **next()** және **previous()** функцияларын енгіздік, өйткені енді **prev** және **succ** нұсқауыштарымен тікелей қатынасуға (оларды тікелей пайдалануға) болмайды. Біз түйін мәнін кластың ашық бөлімінде қалдырдық, өйткені (әзірше) оны жасыруға қажеттілік (себеп) жоқ, олар жай ғана мәліметтер.

Енді алдыңғы нұсқаның көшірмесін алып, оны түрлендіре отырып, **Link::insert()** функциясын жүзеге асырып көрейік.

```
Link* Link::insert(Link* n)// n-ді p-ның алдына
                           // кірістіреді; n-ді қайтарады
{
    Link* p = this;        // берілген объектіге нұсқауыш
    if(n==0) return p;     // ешнәрсені кірістірмейміз
    if(p==0) return n;     // ешнәрсені кірістірмейміз
    n->succ = p;           // p n-нен кейін орналасады
    if(p->prev) p->prev->succ = n;
    n->prev=p->prev;  // p-ның алдындағы n-ніңалдына шығады
    p->prev = n;       // n  p-ның алдына шығады
    return n;
}
```

Link::insert функциясы шақырылған объектіге арналған нұсқауышты қалай алуға болады? Тілдің көмегінсіз оны жасау мүмкін емес. Бірақ әрбір функция-мүшеде **this** идентификаторы бар, ол өзі арнайы шақырылған объектіге нұсқауыш болып табылады. Ал біз оған балама (альтернатива) ретінде **p**-ның орнына жай ғана **this** сөзін жаза алатын едік.

```
Link* Link::insert(Link* n)    // n p-ның алдына қойылады;
{                              // n-ді қайтарады
  if(n==0) return this;
  if(this==0) return n;
  n->succ = this;          // бұл объект n-нен кейін тұрады
  if(this->prev) this->prev->succ = n;
  n->prev = this->prev;    // бұл объектінің алдындағысы
                           // n-нің алдына шығады
  this->prev = n;          // n осы объектінің алдына шығады
  return n;
}
```

Бұл түсініктеме аздап көп сөзді болып көрінеді, бірақ біз класс мүшесін пайдалануды **this** нұсқауышы қамтамасыз етеді деп айтуға міндетті емеспіз, сондықтан берілген кодты қысқартып жазуға да болады:

```
Link*Link::insert(Link* n)      // n p-ның алдына қойылады;
{                               // n-ді қайтарады
    if(n==0) return this;
    if(this==0) return n;
    n->succ = this;         // бұл объект n-нен кейін тұрады
    if(prev) prev->succ = n;
    n->prev = prev;         // бұл объектінің алдындағысы
                            // n-нің алдына шығады
    prev = n;               // n осы объектінің алдына шығады
    return n;
}
```

Басқаша айтқанда, класс мүшесін әрбір пайдаланған сайын **this** нұсқауышын қосалқы түрде қолданып отырамыз. Оны тікелей түрде айтуды қажет ететін жалғыз жағдай – барлық объектілерге сілтеме жасау керек болған кезде туындайды.

this нұсқауышының спецификалық (арнайы) мағынасы бар екеніне назар аударыңыз: ол функция-мүше шақырылатын объектіге сілтеме жасайды. Ол бұдан бұрын пайдаланылған объектілерді нұсқамайды. Компилятор біздің функция-мүшеде **this** нұсқауышының мәнін өзгерте алмайтынымызға кепілдік береді. Мысал қарастырайық:

```
sturct S {
   // . . .
   void mutate(S* p)
   {
     this=p;  //қате: this нұсқауышы өзгертулерге жол бермейді
     // . .  .
   }
};
```

17.10.1 Тағы да тізімдерді қолдану жайында

Программаларды жүзеге асыру мәселесімен айналыса отырып, тізімді пайдалану қандай түрде болатынын көре аламыз.

```
Link* norse_gods = new Link("Thor");
norse_gods = norse_gods->insert (norse_gods,new Link("Odin"));
norse_gods = norse_gods->insert (norse_gods,new Link("Zeus"));
norse_gods = norse_gods->insert (norse_gods,new Link("Freia"));

Link* greek_gods = new Link("Hera");
greek_gods = greek_gods->insert(greek_gods,new Link("Athena"));
greek_gods = greek_gods->insert(greek_gods,new Link("Mars"));
greek_gods=greek_gods->insert(greek_gods,newLink("Poseidon"));
```

Бұл код біздің алдыңғы программамыздың фрагментіне ұқсайды. Сол себепті бұрынғыша оның қателерін түзетейік. Мысалы, соғыс құдайының дұрыс атын көрсетейік:

```
Link* p= greek_gods->find("Mars");
if(p) p->value = "Ares";
```

Зевсті грек құдайларының тізіміне ауыстырамыз:

```
Link* p2 = norse_gods->find("Zeus");
if(p2) {
    if(p2==norse_gods) norse_gods = p2->next();
    p2->erase();
    greek_gods = greek_gods->insert(p2);
}
```

Соңында баспаға мынадай тізім шығарамыз:

```
void  print_all( Link* p)
{
    cout << "{";
    while(p) {
      cout << p->value;
      if (p=p->next()) cout << ", ";
    }
    cout << " }" ;
}

    print_all (norse_gods);
    cout<<"\n";

    print_all (greek_gods);
    cout<<"\n";
```

Ақырында, мынадай нәтиже аламыз:

```
{ Freia, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }
```

Бұл нұсқалардың қайсысы жақсы: құрамында `insert()` функциясы бар болып, қалғандары функция-мүше болғандары ма, жоқ әлде олардың класқа жатпайтындары ма? Мұнда бұлардың онша айырмасы жоқ, бірақ 9.7.5 бөлімінде жазылғандарды еске алыңызшы.

Ескеретін жағдай, біз тізім класын емес, түйін класын құрдық. Нәтижесінде біз қай нұсқауыштың алғашқы элементке сілтеме жасайтынын қадағалауымыз керек. Бұл операцияларды `List` класын анықтап алып, одан да жақсы етуге болатын еді, бірақ жоғарыда көрсетілген класс құрылымы жалпыға ортақ бекітілген нұсқа болып табылады.

ТАПСЫРМАЛАР

Бұл тапсырма екі бөліктен тұрады. Алдыңғы жаттығулар динамикалық жиымдар туралы және олардың `vector` класынан айырмашылығы жайлы мәлімет беруі тиіс.

1. `new` операторын пайдаланып, `int` типті 10 саннан тұратын жиымды бос жады аймағына орналастырыңыз.
2. `int` типті 10 санның мәнін `cout` ағымына шығарыңыз.
3. Жиым орналасқан жады аймағын мәліметтерден (`delete[]` операторын қолдану арқылы) босатыңыз.
4. A жиымының мәндерін `os` ағымына шығаратын `print_array10 (ostream& os, int* a)` функциясын жазып шығыңыз.
5. `int` типті 10 саннан тұратын жиымды бос жады аймағына орналастырыңыз; оның мәнін 100, 101, 102 және т.с.с. мәндермен инициалдаңыз; сол мәндерді баспаға шығарыңыз.
6. `int` типті 11 саннан тұратын жиымды бос жадыға орналастырыңдар; оның мәнін 100, 101, 102 және т.б. ауыстырыңдар; ол мәндерді баспаға шығарыңдар.
7. A жиымының (`n` элементі бар) мәндерін `os` ағымына шығаратын `print_array(ostream& os, int* a, int n)` функциясын жазып шығыңыз.
8. `int` типті 20 саннан тұратын жиымды бос жады аймағына орналастырыңыз; оның мәнін 100, 101, 102 және т.с.с. мәндермен инициалдаңыз; сол мәндерді баспаға шығарыңыз.
9. Сіз жиымды (массивті) өшіруді ұмытқан жоқсыз ба? (Егер ұмытсаңыз, оны қазір жасаңыз.)
10. `vector` класын жиым орнына және `print_vector()` функциясын `print_array()` функциясының орнына қолдана отырып, 5, 6 және 8 тапсырмаларды орындаңыз.

Тапсырманың екінші бөлігі нұсқауыштар мен олардың жиымдармен байланысына арналған. Соңғы тапсырмадағы `print_array()` функциясын пайдаланыңыз.

1. `int` типті айнымалыны бос жады аймағына орналастырыңыз да, оны 7 санымен инициалдап, адресін `p1` нұсқауышына меншіктеңіз.
2. `p1` нұсқауышының мәні мен ол сілтеме жасап тұрған `int` типті айнымалы мәнін баспаға шығарыңыз.
3. `int` типті жеті саннан тұратын жиымды бос жады аймағына орналастырыңыз; оны 1,2,4,8 және т.с.с. мәндермен инициалдаңыз; жиым адресін `p2` нұсқауышына меншіктеңіз.
4. `p2` нұсқауышы мен ол сілтеме жасап тұрған жиымды баспаға шығарыңыз.
5. `int*` типіндегі атауы `p3` болып келген нұсқауышты жариялап, оны `p2` нұсқауышының мәнімен инициалдаңыз.
6. `p1` нұсқауышын `p2` нұсқауышына меншіктеңіз.
7. `p3` нұсқауышына `p2` нұсқауышына меншіктеңіз.
8. `p1` және `p2` нұсқауыштарының мәндерін және де солар сілтеме жасап тұрған элементтерді баспаға шығарыңыз.
9. Пайдаланылған барлық жады аймағын босатыңыз.
10. `int` типті он саннан тұратын жиымды бос жады аймағына орналастырыңыз; оны 1,2,4,8 және т.с.с. мәндермен инициалдаңыз; жиым адресін `p1` нұсқауышына меншіктеңіз.
11. `int` типті 10 саннан тұратын жиымды бос жады аймағына орналастырыңыз; оның адресін `p2` нұсқауышына меншіктеңіз.
12. `p1` нұсқауышы сілтеме жасап тұрған жиымның мәндерін `p2` нұсқауышы сілтеме жасайтын жиымға көшіріп жазыңыз.
13. 10-12 тапсырмаларды жиымды қолданбай, `vector` класын пайдалана отырып, қайталап орындап шығыңыз.

БАҚЫЛАУ СҰРАҚТАРЫ

1. Элементтерінің саны айнымалы болып келетін мәліметтер құрылымы не үшін қажет?
2. Қарапайым программаларда қолданылатын жадының төрт түрін атап көрсетіңіз.
3. Бос жады дегеніміз не? Оны тағы қалай атайды? Бос жады аймағымен қандай операторлар жұмыс істейді?
4. Атаусыз ету операторы дегеніміз не және ол не үшін қажет?
5. Адрес дегеніміз не? C++ тілі адреспен қалай жұмыс жасайды?
6. Нұсқауыш өзі сілтеме жасап тұрған объект туралы қандай ақпарат береді? Ол қандай пайдалы ақпаратты жоғалтады?
7. Нұсқауыш неге сілтеме жасай алады?
8. Жады көлемінің азаюы (leak) дегеніміз не?

9. Ресурс дегеніміз не?
10. Нұсқауышты қалай инициалдауға болады?
11. Нөлдік нұсқауыш деген не? Ол не үшін қажет?
12. Нұсқауыш (сілтеме немесе атаулы объект емес) қандай кезде қажет болады?
13. Деструктор дегеніміз не? Ол қай кезде керек?
14. **Виртуалды** деструктор не үшін қажет?
15. Класс мүшелерінің деструкторлары қалай шақырылады?
16. Типтерді келтіру дегеніміз не? Ол қай кезде қажет?
17. Нұсқауыш көмегімен класс мүшесіне қалай қол жеткізуге болады?
18. Қос байланысты тізім дегеніміз не?
19. `this` айнымалысы нені білдіреді және ол қай кезде қажет?

ТЕРМИНДЕР

`delete`
`delete[]`
`new`
`this`
`void*`
адрес
адресті алу: `&`
атаусыз ету: `*`
босату
бос жады
виртуалды конструктор
деструктор
мүше деструкторы
диапазон
жады
жадының азаюы
индекс
индекстеу: `[]`
келтіру
контейнер
мүшеге қол жеткізу: `->`
нөлдік нұсқауыш
нұсқауыш
орналастыру
ресурстар азаюы
типті түрлендіру
тізім
түйін

ЖАТТЫҒУЛАР

1. Сіздің программалау тіліңіздің жүзеге асуында нұсқауыштың мәнін шығарудың форматы қандай?
2. `int`, `double` және `bool` типтері қанша байт орын алады? `sizeof` операторын қолданбай жауап беріңдер.
3. `s` тіркесіндегі бас әріптерді кіші әріптерге ауыстыратын `void to_lower(char* s)` функциясын жазыңыз. Мысалы, `"Hello, World!"` тіркесі `"hello, world!"` тіркесіне айналады. Стандартты кітапханадағы функцияларды қолданбаңыз. C тіліндегі тіркес стилі нөлмен аяқталатын символдар жиымынан тұрады, сол себепті егер `0` символы кездессе, онда ол жиым соңына келгеніңізді белгісі болады.
4. C тілі стилінде сөз тіркесін белгілеумен қатар оны бос жады аймағына

көшіретін `char* strdup(const char*)` функциясын жазыңыз. Стандартты кітапхана функцияларын қолданбаңыз.

5. `s` тіркесінің ішінен `x` тіркесінің бірінші кездесуін іздейтін `char* findx(const char* s, const char* x)` функциясын С тілі стилінде жазып шығыңыз.
6. Бұл бөлімде `new` операторын қолдану кезінде бөлінген жады аймағының сыртына шығып кеткен кезде қандай жағдай болатыны айтылмады. Бұл *жадының таусылуы* (memory exhaustion) деп аталады. Не болатынын анықтап көріңіз. Сізде екі балама жол бар: құжаттамамен танысу немесе тұрақты түрде жады бөлінгенімен, оның босатылуы ешқашанда орындалмайтын шексіз циклден шықпайтын программа жазу. Екі нұсқаны да қолданып көріңіз. Сіз ол таусылып біткенше, қанша жады көлемін пайдалана аласыз?
7. `cin` ағымындағы символдарды бос жады аймағында орналасқан жиымға оқитын программа жазыңыз. Жеке символдарды оқуды леп белгісі `(!)` енгізілгенше, жалғастыру керек. `std::string` класын қолданбаңыз. Жадының таусылуы жайлы уайымдамаңыз.
8. 7 жаттығуды тағы да бір рет қайталап орындаңыз, бірақ бұл жолы символдарды бос жады аймағына емес, `std::string` тіркесіне оқыңыз.
9. Стек қалай қарай өсіп отырады: жоғарыға (үлкен адрестер жағына қарай) қарай ма немесе төмен (кіші адрестер жағына қарай) қарай ма? Бастапқы кезде (`delete` командасы орындалғанша) қолданылатын жады қай бағытқа қарай өседі? Осы жағдайды анықтайтын программа жазып шығыңыз.
10. 7-жаттығудың шешіміне қараңыз. Енгізу жиымның толып кетуіне әкеле ме; басқаша айтқанда, сіз бөлінген жады көлемінен артық символдар санын енгізе аласыз ба (бұл елеулі қате)? Егер бөлінген жады мөлшерінен артық символдар санын енгізсек не болады? `realloc()` функциясын оқып алып, қажет болғанда оны бөлінген жады көлемін арттыру үшін қолданыңыз.
11. 17.10.1 бөліміндегі құдайлар тізімін құратын программаны аяқтаңыз да, оны орындап шығыңыз.
12. `find()` функциясының екі нұсқасы не үшін қажет?
13. 17.10.1 бөліміндегі `Link` класын түрлендіріп, ол `struct God` типіндегі мәндерді сақтайтындай ету керек. `God` класының `string` типті мынадай мүшелері болуы тиіс: аты, мифологиясы, көліктік құралдары және қаруы. Мысалы: `God("Зевс", "Греция", "", "найзағай") and God ("Один","Норвегия", "Слейпнер атты сегіз аяқты ұшатын ат", "")`. Әрбір жолға құдайлар аттары мен олардың атрибуттарын шығаратын `print_all()` программасын жазыңыз. Дұрыс лексико-графикалық позиция бойынша `new` операторының көмегімен жаңа элементті орналастыратын `add_ordered()` функция-мүшесін қосыңыз. `God`

типті мәндері бар `Link` класының объектілерін пайдалана отырып, үш мифологиядан тұратын құдайлар тізімін құрыңыз. Сонан соң осы тізімдегі элементтерді (құдайларды) үш лексикографикалық түрдегі реттелген тізімге – әр мифологияға бір-бірден кіретіндей етіп ауыстырып орналастырыңыз.

14. 17.10.1 бөліміндегі құдайлар тізімін бір байланысты тізім ретінде жазып шығуға бола ма; басқаша айтқанда, `Link` класындағы prev мүшесін өшіруге бола ма? Қандай себептер бізге осыны істетер еді? Қандай жағдайларда бір байланысты тізімдерді пайдаланған дұрыс болар еді? Осы мысалды бір байланысты тізім арқылы қайтадан жазып шығыңыз.

СОҢҒЫ СӨЗ

Нұсқауыш пен бос жады сияқты төменгі деңгейлі механизмдерді не үшін пайдаланымыз? Оның орнына неге жай ғана `vector` класын қолданбаймыз? Жауаптардың бірі мынадай түрде болады: біреулер `vector` класын және де сол тәрізді абстракцияларды жазды ғой, сондықтан бізге оны қалай жасауға болатынын білу маңызды. Құрамында нұсқауыштары мен төменгі деңгейлі программалауға байланысты ешқандай проблемалары жоқ программалау тілдері бар. Негізінде, осындай тілде жұмыс істейтін программалаушылар аппараттық жабдықтамаларға тікелей қол жеткізуге байланысты есептерді шығаруды C++ тілінде (немесе басқа да төменгі деңгейлі программалауды сүйемелдейтін тілдерде) жұмыс істейтін программалаушыларға жүктейді. Бірақ бізге мұның ең басты себебі программаның физикалық құрылғылармен қалай байланысатынын білмей, компьютер мен программалауды түсіну мүмкін еместігі болып көрінеді. Нұсқауыштар мен жады адрестері және т.с.с. туралы ешнәрсе білмейтін адамдар көбінесе өздері жұмыс істейтін программалау тілінің мүмкіндіктері туралы қате пікірде болады; осындай жаңсақ пікірлер "неден екені белгісіз жұмыс істемейтін" программалардың пайда болуына әкеліп соқтырады.

18

Векторлар мен жиымдар

"Сатып алушы, абай бол!"

– Пайдалы кеңес

Бұл тарауда векторларды қалай көшіруге және оларды индекстердің көмегімен қалай қолдануға болатыны баяндалады. Ол үшін біз жалпы көшіру мен вектордың төменгі деңгейлі жиымдармен байланысын талқылаймыз. Сонымен қатар, жиымның нұсқауыштармен байланысын және осы байланыс негізінде пайда болатын мәселелерді талдаймыз. Бұл тарауда кез келген типке негізделген бес түрлі ең негізгі операциялар қарастырылады, олар: жасау, келісім (үнсіз) бойынша жасау, көшіру арқылы жасау, көшіретін меншіктеу және жою (өшіру).

18.1 Кіріспе

Аспанға көтерілу үшін ұшақ жылдамдығын жерден көтерілу деңгейіне дейін жеткізуі керек. Ол жерден табан айырғанша ұшу алаңындағы ауыр әрі ыңғайсыз жүк тасығыш сияқты көрінеді. Бірақ аспанға көтерілген соң, ұшақ әдеттегідей емес, өте икемді әрі тиімді көлік құралына айналады. Бұл ұшақтың аспанда өз стихиясында болатындығымен түсіндіріледі.

Бұл тарауда біз ұшу алаңының ортасындағы үшақ тәріздіміз. Бұл бөлімнің мақсаты — тілдің құралдары мен программалау технологиялары арқылы компьютер жадын пайдалануға байланысты туындайтын шектеулер мен қиындықтардан арылу болып табылады. Біз программалаудағы типтердің логикалық қажеттіліктерге сәйкес келетін қасиеттерге ие бола алатын кезеңіне ұмтылатын боламыз. Ол үшін біз аппараттық жабдықтамаларға байланысты болатын іргелі шектеулерден арылуымыз керек:

- Компьютер жадындағы объектінің тұрақты көлемі болады.
- Объект жады аймағына орналасқан соң, нақты бір орын алады.
- Компьютер объектілермен ең қажетті операцияларды ғана орындай алады (мысалы, мәліметті көшіру, екі сөзді қосу, т.б.).

Негізінде, мұндай шектеулер C++ тілінің құрамындағы типтерге және операцияларға қатысты болып келеді (олар C тілінен мұра ретінде келген; 22.2.5 бөлімі мен 27-тарауды қ.). Біз 17-тарауда өзінің барлық элементтеріне қол жеткізуді басқаратын және қолданушы тұрғысынан алғанда, табиғи болып көрінгенмен, аппараттық жабдықтамаларды қолдану жағынан онша ыңғайлы болмайтын операцияларды орындауды қамтамасыз ете алатын `vector` типімен танысқан болатынбыз.

Бұл бөлімде біз өз назарымызды көшіру операцияларына аударамыз. Әрине, бұл маңызды, бірақ ол сонымен бірге техникалық түсінік болып табылады. Қарапайым емес объектіні көшіре отырып, біз нені көрсетеміз? Көшіру операциясын орындап болған соң, алынған көшірмелер қандай деңгейге дейін тәуелсіз болып саналады? Көшірудің қандай түрлері бар? Оларды қалай көрсетуге болады? Олар басқа да іргелі операциялармен, мысалы, инициалдау және тазалаумен қандай байланыста болады?

Біз **vector** және **string** сияқты жоғары деңгейлі типтердің көмегінсіз-ақ жадымен жұмыс істеу проблемаларын қарастырамыз, жиымдар мен нұсқауыштарды, олардың өзара байланысы мен қолдану тәсілдерін оқып үйренеміз, сонымен қатар, оларды қолдану кезіндегі туындайтын қиындықтарды да талқылаймыз. Бұл C немесе C++ тілінде жазылған төменгі деңгейдегі кодтармен жұмыс жасайтын программалаушылар үшін маңызды ақпарат болып табылады.

Vector класының нақтылықтары тек векторларға ғана емес, сонымен бірге, төменгі деңгейлі типтерден құрылатын жоғары деңгейлі типтерде де кездесіп отырады. Бірақ кез келген тілдегі әрбір жоғары деңгейлі тип (**string**, **vector**, **list**, **map**) бір-біріне ұқсас машиналық қарапайым мүмкіндіктерден (примитивтерден) құрылады және олар осы бөлімде сипатталатын іргелі мәселелердің әртүрлі жолмен шешілуін көрсетеді.

18.2 Көшіру

Vector класын 17-тараудың соңында көрсетілген күйінде қайтадан қарастырайық.

```
class vector{
  int sz;              // көлемі
  double* elem;        // элементтерге нұсқауыш
public:
  vector (int s)     // конструктор
       :sz(s),elem(new double[s]) {} // жады бөледі
  ~vector()           // деструктор
       { delete[] elem; }            // жадыны босатады
  // . . .
};
```

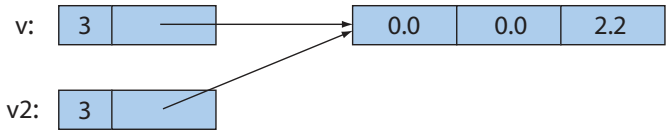
Осындай векторлардың біреуін көшіріп көрейік.

```
void f(int n)
{
   vector v(3);    // үш элементтен тұратын векторды анықтаймыз
   v.set (2, 2.2); // v[2]-нің мәнін 2.2-ге теңестіреміз
```

```
    vector v2 = v;  // мұнда не болады?
    // . . .
}
```

Теория жүзінде `v2` объектісі `v` объектісінің көшірмесі болуы керек (яғни, `=` операторы көшірме жасайды), басқаша айтқанда, `[0:v.size()]` диапазонындағы барлық `i`-лер үшін `v2.size()== v.size()` және `v2[i]== v[i]` шарттары орындалуы керек. Оның үстіне `f()` функциясынан шыққанда, барлық жады бос кеңістікке айналады. Стандартты кітапханадағы `vector` класы, бірақ біз жасаған қарапайым `vector` класы емес, дәл осы әрекетті орындайды. Біздің мақсат — осындай мәселелерді шешу үшін vector класын жақсарту болып табылады. Бірақ алдымен біз осы ағымдағы нұсқамыздың қалай жұмыс істейтінін түсініп алайық. Ол нені дұрыс жасамайды және неге? Осының себебін анықтап барып, біз мәселені шеше аламыз. Және мұнан да маңыздысы басқа жағдайларда да туындайтын осындай мәселелерді айқындау болып табылады.

Класқа қатысты, келісім бойынша көшіру дегеніміз "барлық мәлімет-мүшелерді көшіру" дегенді білдіреді. Оның өз мағынасы бар. Мысалы, `Point` класының объектісін оның координаттарының көшірмесін ала отырып көшіреміз. Бірақ нұсқауыш болып табылатын класс мүшелерін көшірген кезде қиындықтар туындайды. Жеке жағдайда, біздің мысалдағы векторлар үшін `v.sz == v2.sz` және `v.elem == v2.elem` шарттары орындалады, сол себепті біздің векторлар мынадай түрде болады:



Басқаша айтқанда, `v2` объектісі `v` объектісі элементтерінің көшірмелерінен тұрмайды. Ол бұл элементтерге `v` объектісімен бірдей түрде иелік етеді. Бізге келесі түрдегі кодты жазуға да болар еді:

```
v.set(1,99);        // v[1]-дің  мәнін 99-ға тең етеміз
v2.set(0,88);       // v2[0]-дің  мәнін 88-ге теңейміз
cout << v.get(0) << ' ' << v2.get(1);
```

Нәтижесінде `88 99` векторын алатын едік. Бірақ бұл біздің мақсатымыз емес. Егер `v` және `v2` объектілерінің арасында жасырын байланыс болмағанда, нәтиже `0 0` болатын еді, себебі `v[0]` немесе `v2[1]` ұяшығына ешқандай мән жазған жоқпыз. Сіз мұндай әрекетті қызықты, тиянақты немесе пайдалы деп есептеуіңіз мүмкін, бірақ біз мұндай жағдайды көздемедік және стандартты `vector` класында жүзеге асатын нәрсе бұл емес. Сонымен бірге `f()` функциясының нәтижесін қайтарған кезде, бәрі шиеленісіп, тіпті басқаша болып кететіні анық. Және де `v`

және **v2** объектілерінің деструкторлары жанамалы түрде шақырылатын болады. **v** объектісінің деструкторы қолданылған жады аймағын

```
delete[] elem;
```

функциясының көмегімен босатады.

Дәл осындай жағдайды **v2** объектісінің деструкторы да жасайды. Осындағы екі объект үшін де (**v** және **v2**) **elem** нұсқауышы бір ұяшыққа сілтеме жасайтын болғандықтан, бұл жады аймағы екі рет босатылады да, ол өте қисынсыз нәтижелердің туындауына әкеліп соғады.

18.2.1 Көшіру конструкторлары

Сонымен, не істеу керек? Мынадай әрекеттерді орындау керек: элементтерді көшіріп, бір векторды басқа вектормен инициалдау кезінде шақырылатын көшіру операциясын қарастыру қажет. Сонымен, бізге көшірме жасайтын конструктор керек. Мұндай конструктордың *көшіретін конструктор* (copy constructor) деп аталғаны дұрыс болады. Ол аргумент ретінде көшірілуге тиіс объектіге жасалған сілтемені қабылдайды. Яғни **vector** класы мынандай түрде болуы керек:

```
vector (const vector&);
```

Бұл конструктор **vector** класының бір объектісін басқа объектімен инициалдауға талпыну кезінде шақырылатын болады. Біз объектіні сілтеме арқылы жібереміз, өйткені көшіру ісінің негізін атқаратын констуктордың аргументін көшіргіміз келмейді. Бұл сілтемені біз **const** спецификаторы арқылы жібереміз, өйткені аргументті түрлендіргіміз келмейді. Енді **vector** класын нақтылап анықтайық:

```
class vector {
  int sz;
  double* elem;
  void copy (const vector& arg);  //copy элементтерін
                                  // arg-тен *elem-ге көшіру
public:
   vector(const vector&);          // көшіру конструкторы
   // . . .
};
```

copy() функция-мүшесі аргумент болып табылатын элементтерді вектордан көшіреді.

```
void  vector::copy(const vector& arg)
     //  [0:arg.sz-1] элементтерін көшіреді
```

```
{
    for (int i = 0; i<arg.sz; ++i) elem[i] = arg.elem[i];
}
```

Бұл кодта **copy()** функция-мүшесінің **arg** аргументіндегі және ол өзі көшірілетін вектордағы **sz** элементтерін пайдалана алатыны (қолжетімді екені) көрсетілген. Осы мүмкіндікті қамтамасыз ету үшін **copy()** функция-мүшесін жабық етіп жасадық. Бұл **copy()** функция-мүшесін тек **vector** класының құрамды бөлігі болып табылатын функциялар ғана шақыра алады. Бұл функциялар векторлар көлемдерінің бірдей болып келуін қамтамасыз етуі керек.

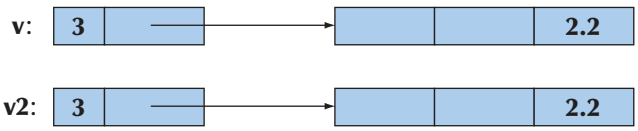
Көшіру конструкторы элементтер санын (**sz**) орнатады және **vector** аргументінен элементтер мәнін көшірер алдында осы элементтерге (**elem** нұсқауышын инициалдау арқылы) қажетті жады көлемін бөледі.

```
vector::vector(const vector& arg)
// элементтерді орналастырады,
// сонан соң оларды көшіру арқылы инициалдайды
    :sz(arg.sz),elem(new double[arg.sz])
{
    copy(arg)
}
```

Көшіру конструкторына қол жеткізе отырып, жоғарыда көрсетілген мысалға қайта оралуға болады.

```
vector v2 = v;
```

Бұл анықтама **v** аргументі бар **vector** класының көшіру конструкторын шақыра отырып, **v2** объектісін инициалдайды. Егер **vector** класының объектісі үш элементтен тұратын болса, онда келесі жағдай туындайтын еді:



Енді деструктор дұрыс жұмыс істей алады. Әрбір элементтер жиыны дұрыс өшірілетін болады. Енді **vector** класының екі объектісі бір-біріне тәуелді болмайды және **v** объектісі элементтерінің мәндерін **v2** объектісінің мазмұнына әсер етпей өзгерте аламыз. Мысал қарастырайық:

```
v.set(1,99); //
v2.set(0,88); //
cout << v.get(0) << ' ' << v2.get(1);
```

Мұның нәтижесі `0 0` болады.

```
vector v2 = v;
```

нұсқауы орнына

```
vector v2(v);
```

деп жазуға болар еді.

Егер `v` және `v2` объектілерінің типтері бірдей болса және бұл типте көшіру операциясы дұрыс жүзеге асырылған болса, онда жоғарыда көрсетілген нұсқаулар бір-бірімен эквивалентті болады да, оларды таңдау өзіңіздің жеке талабыңызға байланысты орындалады.

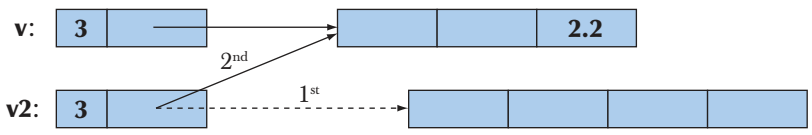
18.2.2 Көшіретін меншіктеу

Векторларды көшіру оларды инициалдау кезінде ғана емес, оларды меншіктеу кезінде де туындауы мүмкін. Инициалдау кезіндегі сияқты (үнсіз) келісім бойынша көшіру әрекеті әрбір элемент бойынша жеке-жеке жүргізіледі, сондықтан да екі рет өшіру әрекеті (18.2.1 бөлімін қ.) мен жады көлемінің азаюы орын алуы мүмкін. Мысал қарастырайық:

```
void f2(int n)
{
    vector v(3); // векторды анықтаймыз
    v.set(2,2.2);
    vector v2(4);
    v2 = v;       // меншіктеу: мұнда не болады?
    // . . .
}
```

Біз `v2` векторы `v` векторының көшірмесі (`vector` стандартты класы дәл осылай жұмыс істейді) болғанын қалап едік, бірақ біздің `vector` класымызда көшіру мағынасы анықталмағандықтан, келісім бойынша меншіктеу жүргізіледі. Басқаша айтқанда, меншіктеу әрбір мүше бойынша жеке-жеке орындалады және `v2` объектісінің `sz` және `elem` мүшелері `v` объектісінің `sz` және `elem` мүшелеріне сәйкес болады.

Мұндай жағдайды келесі түрде көрсетуге болады:



Көшіру конструкторын анықтамай тұрып, `f2()` функциясынан шыққан кезде 18.2 бөліміндегі `f()` функциясынан шыққан кездегі тәрізді қисынсыз жағдай орын алады: `v` және `v2` векторлары сілтеме жасап тұрған элементтер екі рет өшірілетін (`delete[]` операторы арқылы) болады. Сонымен қатар, төрт элементтен тұратын `v2` векторы үшін бастапқы бөлінген жады көлемінің азаюы пайда болады. Біз оларды өшіруді "ұмыттық". Негізінде, бұл проблеманы шешу әрекетінің көшіретін инициалдау есебін шешуден ешқандай айырмасы жоқ (18.2.1 бөлімін қ.). Көшіретін меншіктеу операторын анықтайық.

```
class vector {
  int sz;
  double* elem;
  void copy (const vector& arg); //элементтерді arg-тен
                                 // *elem-ге көшіру
public:
  vector& operator = (const vector &); //көшіретін меншіктеу
  // . . .
};

vector& vector::operator = (const vector& a)
  // бұл векторды a векторының көшірмесі етеді
{
  double* p = new double [a.sz];     // жаңа жады бөлу
  for(int i =0;i<asz;++i) p[i]=a.elem[i];
                                     // элементтерді көшіру
  delete[] elem;                     // жадыны босату
  elem = p;              // енді elem-ді жаңартуға болады
  sz = a.sz;
  return *this;          // ағымдағы объектіге сілтемені
                         // қайтару (17.10 бөлімді қ.)
}
```

Меншіктеу әрекеті объектілер құруға қарағанда шамалы күрделірек, себебі біз ескі элементтермен жұмыс жасауымыз керек. Біздің басты стратегиямыз `vector` класынан элементтерді көшіру болып табылады.

```
double* p = new double [a.sz];
for(int i=0; i<asz; ++i)  p[i] = a.elem[i];
```

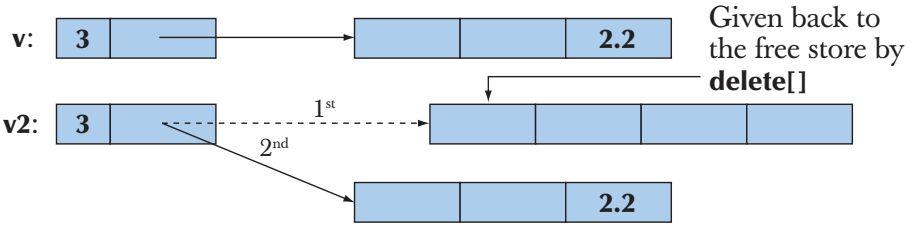
Енді `vector` класының мақсаттық объектісінен ескі элементтерді босатамыз.

```
delete[] elem;          // мәліметі бар жады аймағын босату
```

Соңында жаңа элементтерге **elem** нұсқауышын орнатамыз.

```
elem = p;      //енді elem нұсқауышын өзгерте аламыз
sz = a.sz;
```

Нәтижені келесі түрде көрсетуге болады.



Енді **vector** класындағы жады көлемінің азаюы жойылды, ал жады тек бір рет қана босайды **delete[]**.

Меншіктеуді іске асыра отырып, көшірме жасалғанша, ескі элементтер орналасқан жады аймағын босатып, кодты қарапайым етуге болады, бірақ егер сіз ақпаратты алмастыруға сенімді болмасаңыз, онда оны өшірудің қажеті жоқ. Сонымен бірге, егер сіз бұл әрекетті орындасаңыз, **vector** класы объектісінің өзін өзіне меншіктеу барысында түрлі жағдайлар орын алуы мүмкін.

```
vector v(10);
v=v;           // өзін өзіне меншіктеу
```

18.2.3 Көшіруге байланысты терминология

Көшіру көптеген программалар мен программалау тілдерінде кездеседі. Мұндағы басты мәселе ненің көшірілуі: нұсқауыш (немесе сілтеме) па әлде ол сілтеме жасайтын ақпарат көшіріле ме?

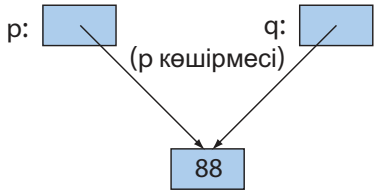
- *Шектелген көшіру* (shallow copy) тек нұсқауышты көшіруді көрсетеді, сондықтан мұның нәтижесінде бір объектіге екі нұсқауыш сілтеме жасауы мүмкін. Дәл осы көшіру механизмі нұсқауыштар мен сілтемелермен жұмыс істеу негізінде жатыр.

- *Тереңдетілген көшіру* (deep copy) нұсқауыш сілтеме жасап тұрған ақпаратты көшіруді қарастырады, мұның нәтижесінде екі нұсқауыш әртүрлі объектілерге сілтеме жасайды. Бұл көшіру механизмі негізінде **vector**, **string** және т.б. кластары жүзеге асырылған. Егер біз тереңдетілген көшіруді іске асырғымыз келсе, онда біздің кластарымызда көшіру конструкторы мен көшірілетін меншіктеу әрекеттері жүзеге асырылуы қажет.

Шектелген көшіру мысалын қарастырайық.

```
int*  p = new int(77);
int* q = p;  // p нұсқауышын көшіру
*p=88;       // p және q нұсқауыштары сілтеме жасап тұрған
                        // int айнымалысы мәнін өзгерту
```

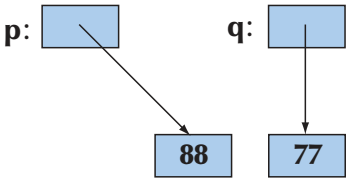
Мұндай жағдайды келесі түрде көрсетуге болады:



Осыған қарама-қарсы тереңдетілген көшіру механизмін жасай аламыз.

```
int* p = new int(77);
int* q = new int(*p); // жаңа int айнымалысын орналастыру,
      // сонан кейін p сілтеме жасап тұрған мәнді көшіру
*p = 88;      // p сілтеме жасап тұрған мәнді өзгерту
```

Мұндай жағдайды төмендегідей түрде көрсетуге болады:



Осы терминологияны қолдана отырып, біздегі **vector** класына байланысты туындаған қиындықтар оны шектелген көшіру жасауымызға байланысты болған және біз **elem** нұсқауышы сілтеме жасаған элементтерді көшірмеуімізде болып тұр. Біздің жетілдірілген **vector** класымыз стандартты **vector** класы сияқты элементтерге жаңа жады бөлу мен олардың мәнін көшіру арқылы тереңдетілген көшіруді орындайды. Шектелген көшіруді қарастыратын типтер туралы, олардың *нұсқауыштар семантикасы* (pointer semantics) немесе *сілтемелер семантикасы* (reference semantics) бар деп айтылады, яғни адрестерді көшіреді. Тереңдетілген көшіруді орындайтын типтерде *мәндер семантикасы* (value semantics) бар деп айтылады, яғни мұнда сілтеме жасалатын мәндер көшіріледі. Қолданушы тұрғысынан қарағанда, мәндер семантикасы бойынша типтер ешқандай нұсқауыш қол-

данылмаған сияқты жұмыс істейді де, тек көшірілетін мәндер болады. Көшіру тұрғысынан қарағанда, мәндер семантикасы бар типтердің **int** типінен онша көп айырмашылығы жоқ.

18.3 Негізгі операциялар

Кластың қандай конструкторлары болуы қажет, оның деструкторы болуы қажет пе және көшіретін меншіктеу талап етіле ме, жоқ па – міне, осы жайында әңгіме бастайтын кез келді. Келесі негізгі бес операцияны қарастыру қажет:

- Бір немесе бірнеше аргументтері бар конструктор.
- Келісім бойынша конструктор.
- Көшіру конструкторы (типтері бірдей объектілерді көшіру).
- Көшіретін меншіктеу (типтері бірдей объектілерді көшіру).
- Деструктор.

Көбінесе кластың аргументтері объектіні инициалдайтын бір немесе бірнеше конструкторлары болуы тиіс.

```
string s("Триумф");     // "Триумф" тіркесімен s объектісін
                                          // инициалдаймыз
vector<double> v(10);  // double типті 10 саннан  тұратын
                                        // v векторын құрамыз
```

Көріп отырғанымыздай, инициализатордың мағынасы мен оның қолданылуы толығынан конструктормен анықталады. **string** класының стандартты конструкторы бастапқы мән ретінде символдық тіркесті қолданады, ал **vector** класының стандартты конструкторы параметр ретінде элементтер санын алады. Көбінесе конструктор инвариантты (9.4.3 бөлімді қ.) орнату үшін қолданылады. Егер біз класс үшін тиімді инвариант анықтай алмасақ, онда ол, бәлкім, кластың немесе мәліметтер құрылымының дұрыс жобаланбағанынан шығар.

Аргументтері бар конструктор өздері жүзеге асырылған (орналасқан) класқа тәуелді болады. Қалған операциялардың стандартты құрылымдары болады.

Класқа үнсіз келісім бойынша конструктор қажет екенін қалай білуге болады? Ол біз инициализаторсыз класс объектілерін құрғымыз келген жағдайда талап етіледі. Осындай жағдайдың кең таралған қарапайым мысалы **vector** типті стандартты контейнерге класс объектілерін орналастыру кезінде туындайды. Төмендегі нұсқаулар **int**, **string** және **vector<int>** типтері үшін келісім бойынша қарастырылған мәндер тұрғандықтан жұмыс істейді:

```
vector<double> vi(10); // double типті 10 элементі бар вектор
          // олардың әрқайсысы 0.0 мәнімен инициалданған
vector<string> vs(10); // string типті 10 элементі бар вектор
          // олардың әрқайсысы " " мәнімен инициалданған
vector<vector<int>> vvi(10); // 10 вектордан тұратын вектор,
// олардың әрқайсысы vector конструкторымен инициалданған
```

Сонымен, келісім бойынша конструктордың болуы көбінесе пайдалы болып табылады. Мынандай сұрақ туындайды: қандай кезде келісім бойынша конструктордың болғаны дұрыс? Жауабы: біз келісім бойынша мағыналы және күтілген мәні бар класс инвариантын орната алатын болсақ, сонда қолдануға болады. **int** және **double** тәрізді сандық типтер үшін күтілетін мән **0** (**double** типі үшін **0.0**) болып табылады. **string** типі үшін күтілетін таңдау **" "** болып табылады. **Vector** класы үшін бос векторды қолдануға болады. Егер **T** типінің келісім бойынша мәні бар болса, онда ол **T()** конструкторымен беріледі. Мысалы, **double() 0.0** мәніне тең, **string() " "** мәніне тең болса, **int** типті айнымалыларды сақтау үшін **vector<int>** класындағы бос **vector** қолданылады.

Егер кластың ресурстары болса, онда оның деструкторы болуы тиіс. Ресурс (қор) — ол сіз "бір жерден" алған нәрсе және қолданып болған соң, оны қайтадан қайтару қажет. Мұның мысалы ретінде, **new** операторының көмегімен бөлінген жады аймағын айтамыз, оны пайдаланып болған соң, **delete** және **delete[]** операторларын қолданып, босату қажет. **vector** класы өзінің элементтерін сақтау үшін жады көлемін қажет етеді, сондықтан ол оны соңынан қайтаруы керек, сол себепті оның деструкторы болуы тиіс. Күрделі программаларда қолданылатын басқа ресурстар — бұл файлдар (егер сіз файл ашсаңыз, оны жабуға тиіссіз), бұғаттауыштар (locks), ағындар дескрипторлары (thread handless) және де процестер мен қашықтағы компьютерлер арасындағы өзара байланысты қамтамасыз ету үшін қолданылатын *қосбағытты арналар* (sockets).

Класқа деструктордың қажет екенінің басқа да белгісі бар, ол нұсқауыш немесе сілтеме болып табылатын класс мүшелерінің болуы. Егер класс мүшелерінің біреуі нұсқауыш немесе сілтеме болып табылса, онда оған деструктор және көшіру операциясы қажет.

Деструкторы болуы тиіс класс тәжірибе жүзінде әрқашанда да көшіру конструкторы мен көшіретін меншіктеудің болуын талап етеді. Оның себебі егер объектінің ресурсы болса, онда келісім бойынша көшіру (мүшелер бойынша шектелген көшіру тәсілі) әрекеті көбінесе қателікке алып келеді. Оның классикалық мысалы болып **vector** класы саналады.

Егер туынды кластың деструкторы болуы керек болса, онда базалық кластың виртуалды деструкторы болуы тиіс.

18.3.1 Тікелей конструкторлар

Бір аргументі бар конструктор осы аргумент типін өзінің класына түрлендіруді анықтайды. Бұл өте пайдалы болуы мүмкін. Мысал қарастырайық:

```
class complex {
public:
   complex(double);          //double типін complex типіне
                                 // түрлендіруді анықтайды
   complex(double,double);
   // . . .
};

complex z1 = 3.18; // ОК: 3.18-ді (3.18,0) етіп түрлендіреді
complex z2 = complex(1.2,3.4);
```

Бірақ тікелей емес түрлендіруді сирек қолданып, абай болу керек, себебі олар күтілмеген келеңсіз әсерлерге алып келуі мүмкін. Мысалы, біздің жоғарыда анықтаған **vector** класымыздың **int** типті аргумент қабылдайтын конструкторы бар. Осыдан барып, оның **int** типін **vector** класына түрлендіруді анықтайтыны шығады. Мысал қарастырайық:

```
class vector {
   // . . .
   vector(int);
   // . . .
};

vector v=10;   //double типті 10 элементі бар вектор жасаймыз
v = 20;              // v векторына double to v типіндегі
                  // 20 элементі бар жаңа вектор меншіктейміз
void f(const vector&);
f(10);         //  double  типті  10  элементтен  тұратын
                     // жаңа вектормен f функциясын шақыру
```

Біз күткен нәрседен де көп нәтижеге жеттік. Қуанышқа орай, мұндай тікелей емес (жанама) түрлендіруді басу жеңіл болып табылады. **explicit** түйінді сөзі бар конструктор тек қарапайым құру семантикасын жүзеге асыра алады да, жанама түрлендіруге жол бермейді. Мысал қарастырайық:

```
class vector {
    //  . . .
    explicit vector(int);
    // . . .
};
```

```
vector v = 10;          // қате: int типін vector<double>
                        // типіне түрлендіру жоқ
v = 20;                 // қате: int типін vector<double>
                        // типіне түрлендіру жоқ
vector v0(10) ;         // OK

void f(const vector<double>&);
f(10);                  // қате: int типін vector<double>
                        // типіне түрлендіру жоқ
f(vector<double>(10)); // OK
```

Күтпеген түрлендіруден абай болу үшін, біз – және тіл стандарты да – бір ғана аргументі бар **vector** класы конструкторының **explicit** спецификаторы болуын талап еттік. Барлық конструкторлардың келісім бойынша **explicit** спецификаторының болмағаны өкініштi-ақ, егер күмәндансаңыз онда **explicit** түйінді сөзін қолданып, бір аргументпен шақырылатын конструкторды жариялаңыз.

18.3.2 Конструкторлар мен деструкторларды түзетіп жөндеу

Конструкторлар мен деструкторлар программаның нақты анықталған және белгілі бір жерлерінде шақырылады. Алайда біз көбінесе, мысалы, мынадай **vector()**, тікелей шақырулар жасай бермейміз; кей кездері біз **vector** класының объектісін жариялауды жазамыз да, оны мәні бойынша функцияның аргументі ретінде жібереміз немесе оны **new** операторының көмегімен бос жады аймағында құрамыз. Бұл жағдай синтаксис терминдері арқылы ой түйетін қолданушылар үшін түсінбестік тудыруы мүмкін. Конструкторды шақыруды диспетчерлеуді жүзеге асыратын синтаксистік конструкция жоқ. Конструктор мен деструктор жөнінде былай ойлау жеңілірек болады:

- **X** класының объектісі құрылған кезде, оның конструкторларының біреуі шақырылады;
- **X** типті объект жойылған кезде, оның деструкторы шақырылады.

Деструктор әрқашанда класс объектісі жойылған кезде шақырылады, бұл жағдай объект көріну аймағынан шығып кеткенде, программа жұмысын тоқтатқанда немесе объект нұсқауышына **delete** операторы қолданылған кезде орын алады. Бұған лайықты конструктор класс объектісі құрылған сайын шақырылады, мұндай әрекет айнымалыны инициалдау кезінде, **new** операторының көмегімен объектіні құрғанда және де объектіні көшіру кезінде болады.

Мұндайда не болады? Осы жағдайды түсіну үшін конструкторларға көшіретін меншіктеуді және шығару операторының деструкторын енгізейік. Мысалы:

```
struct X {                              // қарапайым тест класы
   int val;

   void out (const string& s)
     { cerr<< this<< "->" << s << ": " << val << "\n"; }

   X() {out("X()"); val=0;}  //келісім бойынша конструктор
   X(int v) { out("X(int) "); val = v; }
   X(const X& x){out("X(X&)"); val = x.val;}
                                        // көшіру конструкторы
   X& operator = (const X& a)           // көшіретін меншіктеу
     { out("~X::operator = ()"); val = a.val; return *this; }
   ~X() { out("~X()"); }                // деструктор
} ;
```

X класының объектісімен операциялар орындаған кезде не болатынын қадағалайық. Мысал келтірейік:

```
X glob (2);                             // глобальды айнымалы

X copy(X a) {return a;}

X copy2(X a) {X aa = a; return aa;}

X& ref_to(X& a) {return a;}

X* make(int i) {X a(i); return new X(a);}

struct XX { X a; X b; }

int main()
{
   X loc(4);                            // локальді айнымалы
   X loc2 = loc;
   loc = X(5);
   loc2 = copy(loc);
   loc2 = copy2(loc);
   X loc3(6);
   X& r = ref_to(loc);
   delete make(7);
   delete make(8);
   vector<X> v(4);
   XX loc4;
```

```
    X* p = new X(9); //бос жады аймағында X класының объектісі
    delete p;
    X* pp = new X[5];     //бос жады аймағында X класының
                          //объектілері жиымы
    delete []  pp ;
}
```

Осы программаны орындап көріңіз.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Төмендегі программаны орындап, оның нәтижесін түсінетіндігіңізге көз жеткізіңіз. Егер түсінсеңіз, онда объектілерді құру және жою жөнінде білуге керек заттардың басым бөлігін меңгердіңіз деуге болады.

Өз компиляторыңыздың сапасына қарай сіз `copy()` және `copy2()` функцияларын шақыруға байланысты орындалмай қалған көшірулер бар екенін байқай аласыз. Біз (қолданушылар) бұл функциялардың ештеме істемейтінін көреміз, олар тек мәндерді енгізу ағымынан шығару ағымына ешқандай өзгеріссіз көшіреді. Егер компилятор өте сапалы жұмыс істейтін болып, осы жағдайды байқайтын болса, онда ол көшіру конструкторының осындай шақыруларын өшіріп тастай алады. Басқаша айтқанда, компилятор көшіру конструкторының тек көшірме жасап, басқа ештеңе де жасамайтынын анықтайды деп айтуға болады. Кейбір компиляторлар, тіпті жалған көшірмелерді алып тастай алатын деңгейде "ақылды" болып көрінеді.

Сонымен, неге "түкке тұрғысыз `X` класымен" бас ауыртамыз? Бұл сазгерлердің саусақтарын жетілдіретін жаттығулар жасағанын еске салады екен. Мұндай жаттығулардан кейін мағынасы тереңірек көптеген заттар түсінікті әрі жеңіл болып көрінеді. Бұған қоса, егер сізде конструктор мен деструкторды қолданған кезде қиындықтар туындайтын болса, онда олардың ішіне шығару операторларын енгізіп, қалай жұмыс жасайтынын байқауға болады. Көлемді программалар үшін мұндай әрекеттер тиімсіз болып келеді, бірақ олардың да өздеріне сәйкес жөндеп түзету технологиялары бар. Мысалы, біз конструктор мен деструкторды шақыру сандарының айырмасы нөлге тең екенін анықтап, жады көлемінің азаюы орын алатынын (алмайтынын) байқай аламыз. Программалаушылар көбінесе жады көлемін бөлетін немесе объектіге нұсқауыштардан тұратын кластар үшін көшіретін меншіктеу немесе көшіру конструкторын анықтауды ұмытып кетіп жатады. Ал бұл өз кезегінде күрделі мәселелердің (олардан оңай құтылуға болады) туындауына әкеліп соғады.

Егер бұл мәселе мұндай қарапайым құралдар көмегімен шешілмейтіндей, тым күрделі болатын болса, онда программаны жөндеп түзетудің кәсіби құралдарын меңгеру керек, олар *азаю детекторлары* (leak detectors) деп аталады. Бірақ ең дұрысы, мұндай келеңсіздікті жою емес, оның туындауына жол бермейтіндей етіп программалау болып табылады.

18.4 Вектор элементтеріне қол жеткізу

Осы кезге дейін біз вектор элементтеріне қол жеткізу үшін `set()` және `get()` функция-мүшелерін қолданып келдік. Бірақ бұл тәсіл көлемді әрі қолайсыз. Біз қарапайым `v[i]` түрінде индекстеуді қолданғымыз келеді. Ол үшін алдымен `operator[]` болып аталатын функция-мүшені анықтап алу керек. Оның алғашқы нұсқасы мынадай болмақ:

```
class vector {
     int sz;         // мөлшері
     double* elem;   // элементтерге нұсқауыш
public:
     // . . .
     double operator[] (int n) { return elem[n]; }
     // элемент қайтарамыз
};
```

Бұл – өте жақсы және ыңғайлы болып көрінгенмен, өкінішке орай, өте қарапайым. Индекстеу операторына (`operator[]()`) мән қайтаруға рұқсат бере отырып, біз элементтерді жазуға емес, оқуға рұқсат еттік.

```
vector v(10);
int x = v[2];     // жақсы
v[3] = x;         // қате: = операторының сол жағында v[3]
                                        // тұра алмайды
```

Бұл жерде `v[i]` өрнегі `v.operator[](i)` операторын шақыру ретінде қолданылады, ол `v` векторының `i` нөмірлі элементінің мәнін қайтарады. `vector` класының мұндай өте қарапайым нұсқасында `v[3]` мәні жылжымалы нүктесі бар саннан тұратын айнымалы емес, жылжымалы нүктесі бар сан болып табылады.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

`vector` класының бір нұсқасын құрып алып, оны компиляциядан өткізіңіз де, `v[3]=x` нұсқауы үшін компилятордың берген мәліметіне көз салыңыз.

Келесі нұсқада біз `operator[]` операторына нұсқауышты өзіне сәйкес элементке қайтаруға рұқсат етеміз:

```
class vector {
     int sz;         // мөлшері
     double* elem;   // элементке нұсқауыш
```

```
public:
    // . . .
    double* operator[] (int n) {return &elem[n];}
    // нұсқауышты қайтарамыз
};
```

Мұндай анықтау кезінде біз элементтерді жаза аламыз.

```
vector v(10);
for(int i=0;i<v.size();++i){ // жұмыс істейді,
                             // бірақ бұрынғыша әдемі емес
   *v[i] = i;
   cout << *v[i];
}
```

Бұл жерде `v[i]` өрнегі `v.operator[](i)` операторын шақыру әрекетін орындайды да, ол нұсқауышты `v` векторының `i` нөмірлі элементіне қайтарады. Ендігі мәселе біз осы элементке сілтеме жасайтын нұсқауышты атаусыз ету үшін `*` операторын жазуымыз керек. Бұл `set()` және `get()` функциясы сияқты онша әдемі болмайды. Егер индекстеу операторынан сілтемені қайтаратын болсақ, бұл мәселені шешуге болады.

```
class vector {
   // . . .
   double& operator[](int n) {return elem[n];}
   //сілтемені қайтару
};
```

Енді келесі нұсқаны былай жазуға болады:

```
vector v(10);
for(int i = 0; i<v.size(); ++i){      // жұмыс істеп тұр!
   v[i]=i;  // v[i] сілтемені i нөмірлі элементке қайтарады
   cout << v[i];
}
```

Біз дәстүрлі белгілеуді қамтамасыз еттік: `v[i]` өрнегі `v.operator[](i)` операторын шақыру ретінде түсіндіріледі, яғни, `i` нөмірлі `v` векторының элементіне сілтемені қайтарады.

18.4.1 const түйінді сөзін асыра жүктеу

Жоғарыда анықталған **operator[]** функциясының бір кемшілігі бар: оны тұрақты (константалық) вектор үшін шақыруға болмайды. Мысал көрсетейік:

```
void f(const vector& cv)
{
     double d = cv[1];    // күтілмеген қате
     cv[1] = 2.0;         // күтілген қате
}
```

Мұның себебі **vector::operator[]()** функциясы **vector** класының объектісін әлеуетті (потенциалды) түрде өзгертіп жібере алады. Негізінде, ол мұндай жағдайды жасамайды, бірақ компилятор мұны білмейді, себебі біз оған хабарлауды ұмытып кеттік. Осы мәселені шешу үшін алдымен **const** спецификаторы бар функция-мүшені қарастыру қажет (9.7.4 бөлімді қ.). Мұны жасау жеңіл болып табылады.

```
class vector {
   // . . .
   double& operator[](int n);
      // константалық емес векторлар үшін
   double operator[](int n) const;
      // константалық векторлар үшін
};
```

Әрине, біз **double** типіндегі сілтемені **const** спецификаторы бар нұсқадан қайтара алмас едік, сондықтан **double** типіндегі мәнді қайтарамыз. Осындай жолмен біз **const double&** типіндегі сілтемені де қайтара алар едік, бірақ **double** типіндегі объект шағын болып келеді, сол себепті сілтемені қайтарудың қажеті жоқ (8.5.6 бөлімді қ.), мұнда біз мән қайтаратын болып шештік. Енді келесі кодты жаза аламыз:

```
void ff(const vector& cv, vector& v)
{
     double d = cv[1];
     // тамаша (константалық нұсқаны [] пайдаланады)
     cv[1] = 2.0;
     // қате (константалық нұсқаны [] пайдаланады)
     double d = v[1];
     // тамаша (константалық емес нұсқаны [] пайдаланады)
     v[1] = 2.0;
     // тамаша (константалық емес нұсқаны [] пайдаланады)
}
```

`vector` класының объектілері көбінесе константалық сілтемеде берілетіндіктен, `const` түйінді сөзі бар `operator[]()` операторының осы нұсқасы оның толықтырылуы болып табылады.

18.5 Жиымдар

Бұған дейін біз *жиым* (array) сөзін бос жады аймағында орналастырылған объектілер тізбегін атау үшін қолданып келдік. Дегенмен, жиымдарды атаулы айнымалылар ретінде кез келген жерде орналастыруға болады. Негізінде, бұл кең таралған жағдай болып табылады. Олар келесі түрде қолданылуы мүмкін:

- Ауқымды (глобальді) айнымалылар ретінде (алайда ауқымды айнымалыларды қолдану жақсы шешім болып табылмайды);
- Жергілікті (локальді) айнымалылар ретінде (бірақ жиымдар оларға салмақты шектеулер қояды);
- Функциялар аргументтері ретінде (бірақ жиым өз көлемін білмейді);
- Класс мүшелері ретінде (класс мүшелері болып табылатын жиымдарды инициалдау қиынға түседі).

Жиымға қарағанда, біздің `vector` класына көбірек назар аударатынымызды байқаған шығарсыз. `vector` класын кез келген жағдайда қолдану керек. `vector` класын кез келген мүмкіндікте пайдалануға тырысу керек. Алайда жиымдар векторлар пайда болғанға дейін кең таралған болатын және көптеген тілдерде солардың жақын прототипі болып табылды (әсіресе, C тілінде). Сондықтан ескі программалармен жұмыс істеу мүмкіндігіне ие болу үшін немесе `vector` класының артықшылығын мойындамайтын қолданушылар жазған программаларды пайдалану үшін оларды жақсы білген жөн.

Сонымен, жиым дегеніміз не? Оны қалай анықтаймыз және пайдаланамыз? *Жиым* — қатар орналасқан жады ұяшықтарында сақталатын біртекті объектілер тізбегі. Басқаша айтқанда, жиымның барлық элементтері бір типте болады және олардың арасында бос орын болмайды. Жиым элементтері нөлден бастап, өсу ретімен нөмірленеді. Жиымды жариялағанда, квадрат жақша көрсетіледі.

```
const int max = 100 ;
int gai[max];   //глобальді жиым (int типті 100 саннан тұрады);
                // "әрқашанда болады"

void f(int n)
 {
```

```
    char lac[20];        // локальді жиым; көріну аймағының
                                    // соңына дейін "болады"
    int lai[60];
    double lad[n];
    // қате: жиым өлшемі константа болып табылмайды
    // . . .
}
```

Шектеуге назар аударыңыз: атаулы жиым элементтерінің көлемі компиляция кезеңінде белгілі болуы керек. Егер біз жиым элементтерінің көлемі айнымалы болғанын қалайтын болсақ, онда оны бос жады аймағына орналастырып, оны нұсқауыш арқылы пайдалануымыз керек. vector класы жиым элементтерімен дәл осылай жұмыс жасайды.

Бос жады аймағында орналасқан жиым элементтерін пайдалану сияқты атаулы жиым элементтерін қолдану да индекстеу және атаусыз ету (`[]` және `*`) операторларының көмегімен орындалады. Мысал қарастырайық:

```
void f2()
{
  char lac[20];    // локальді жиым; көріну аймағының
                   // соңына дейін "болады"
  lac[7] = 'a';
  *lac = 'b';     // lax[0] = 'b' нұсқауының баламасы
  lac[-2] = 'b';  // ??
  lac[200] = 'c'; // ??
}
```

Бұл функция компиляциядан өтеді, бірақ біз білетіндей, компиляциядан өткен функциялардың бәрі дұрыс жұмыс істей бермейді. `[]` операторын қолдану керек, бірақ берілген аралықтан шығуды тексеру орындалып тұрған жоқ, сондықтан `f2()` функциясы компиляциядан өтеді, ал `lac[-2]` және `lac[200]` өрнегінің нәтижесі келеңсіз жағдайға ұшыратады (берілген аралықтан шығып кеткен бұрынғы жиымдар сияқты). Мұны жасамаңыз. Жиымдар берілген аралықтан шығып кетуді тексермейді. Мұнда тағы да бізге физикалық жадымен жұмыс істеуге тура келеді, өйткені жүйелік сүйемелдеуге сенуге болмайды.

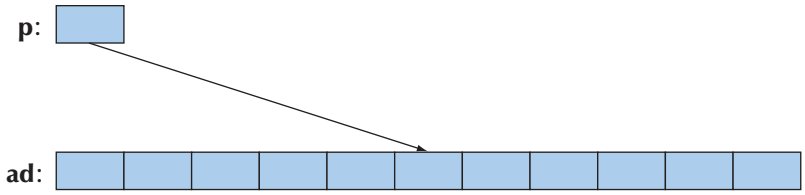
Компилятор `lac` жиымының тек жиырма элементтен тұратынын компилятор көріп қалып, сол себепті `lac[200]` өрнегін қате деп санамас па екен? Негізінде, солай болуы да мүмкін, бірақ бізге белгілісі, қазіргі кезде ондай бір де бір компилятор жоқ. Нақтырақ айтсақ, компиляция кезеңінде жиым шекарасын бақылау тіпті мүмкін емес, ал қарапайым қателерді (жоғарыда көрсетілгендер тәрізді) анықтау барлық мәселелерді шеше алмайды.

18.5.1 Жиым элементтеріне нұсқауыштар

Нұсқауыш жиым элементіне сілтеме жасай алады. Мысал қарастырайық:

```
double ad[10];
double* p = &ad[5];    // ad[5] элементіне сілтеме жасайды
```

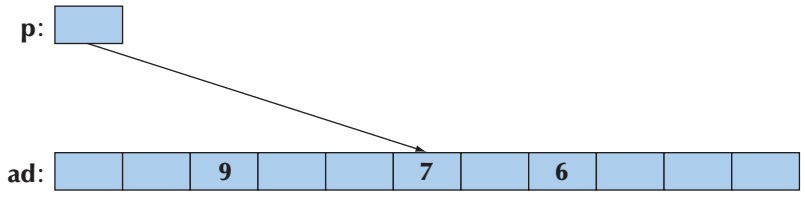
`p` нұсқауышы `ad[5]` ретінде белгілі болып тұрған `double` типті айнымалыға сілтеме жасайды.



Бұл нұсқауышты индекстеп, оны атаусыз етуге болады.

```
*p = 7;
p[2] = 6;
p[-3] = 9;
```

Енді бұл мысал мынадай түрге келеді:

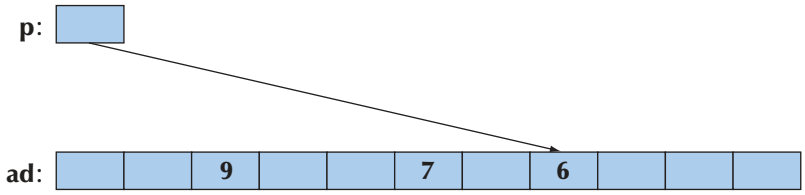


Басқаша айтқанда, біз нұсқауышты оң сандар арқылы да және теріс сандар көмегімен де индекстей аламыз. Себебі, нәтижелер берілген аралықтан шығып кетпеген, мұндай өрнек дұрыс болып табылады. Алайда берілген аралықтан шығып кету заңсыз әрекет болып табылады (бос жады аймағында орналасқан жиымдар тәрізді, 17.4.3 бөлімді қ.). Көбінесе жиымның берілген аралықтан шығып кетуін компилятор көрсете алмайды да (ерте ме, кеш пе), бұдан келеңсіз жағдайлар туындайды.

Егер нұсқауыш жиым ішіндегі элментке сілтеме жасап тұрса, онда оны басқа элементке ауыстыру үшін қосу мен азайту операцияларын қолдануға болады. Мысал қарастырайық:

```
p+ = 2;  // p нұсқауышын екі элемент оңға ауыстыру
```

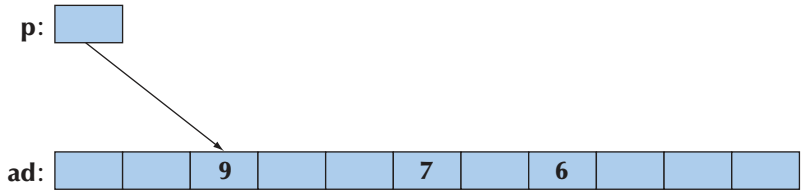
Сөйтіп, келесі жағдайға келеміз:



Осы сияқты

```
p -= 5;
```

Нәтижесі келесі түрде болады:



Нұсқауыштарды ауыстыру үшін `+`, `-`, `+=`, `-=` операцияларын қолдану – *нұсқауыштар арифметикасы* (pointer arithmetic) деп аталады. Негізінде, осы операцияларды қолданған кезде біз берілген аралықтан шығып кетпеуді қамтамасыз етуіміз керек.

```
p += 1000;
// абсурд: p 10 саны ғана бар жиымға сілтеме жасап тұр
double d=*p;
// заңсыз: мүмкін мәні қате шығар (тіпті айтуға келмейтін мән)
*p = 12.34;
// заңсыз: белгісіз басқа мәліметтерге тиісуіміз мүмкін
```

Өкінішке орай, көбінесе нұсқауыштар арифметикасына қатысты күрделі қателерді анықтау оңай болмайды. Сол себепті нұсқауыштар арифметикасын қолданбаған тиімді болып табылады.

Көп таралған нұсқауыштар арифметикасына нұсқауыш инкрементациясы (`++` операторы) жатады, ол келесі элементке сілтеме жасайды және нұсқауыш декрементациясы (`--` операторы) алдыңғы элементке сілтеме жасау үшін қолданылады. Мысалы, `ad` жиымының элементтерін келесі түрде шығаруға болатын еді:

```
for(double* p = &ad[0]; p < &ad[10]; ++p) cout << *p << '\n';
```

Және кері бағытта:

```
for(double* p = &ad[9]; p >= &ad[0]; --p) cout << *p << '\n';
```

Нұсқауыштар арифметикасын осылай қолдану өте кең таралмаған. Бірақ біздің ойымызша, соңғы мысал ("кері") қауіпсіз емес. Неге `&ad[10]` емес, `&ad[9]`? Бұл мысалдардың екеуінде де индекстеуді қолдансақ, тіпті жақсы (екеуіне де тиімді) болатын еді. Сонымен бірге, олар берілген аралықтан шығып кетуді оңай тексеретін `vector` класына пара-пар (эквивалентті) болып келетін еді.

Көптеген нақты программаларда нұсқауыштар арифметикасы функцияның аргументі ретінде нұсқауышты беру әрекетімен тығыз байланысты болып келетінін айта кетейік. Бірақ мұндай жағдайда компилятор нұсқауыштың қанша элментке сілтеме жасайтынын біле алмайды, сондықтан оны өзіңіз қадағалап отыруыңыз керек.

Жалпы, неге C++ тілінде нұсқауыштар арифметикасын қолдануға рұқсат берілген? Негізінде, мұның жұмысы көп және ол индекстеумен орындалатын әрекеттерге қарағанда жаңа ештеңе бермейді. Мысал келтірейік:

```
double* p1 = &ad[0];
double* p2 = p1+7;
double* p3 = &p1[7];
if (p2 != p3) cout << "impossible!\n";
```

Негізі, бұлар тарихи себептерге байланысты орын алған. Мұндай ережелер C тілі үшін ондаған жыл бұрын жасалған болатын және оларды алып тастасақ, көптеген керекті программаларды жоғалтып алуға тура келер еді. Мұны қолданудың бір себебі, кейбір жағдайларда нұсқауыштар арифметикасын қолдану төменгі деңгейлі программалармен, мысалы, компьютер жадын басқару механизмінде ыңғайлы түрде жұмыс істеуді қамтамасыз етеді.

18.5.2 Нұсқауыштар мен жиымдар

Жиым атауы барлық жиым элементтеріне қатысты болып саналады. Мысалы:

```
char ch[100];
```

`ch` жиымының көлемі, яғни, `sizeof(ch)` 100-ге тең. Бірақ жиым атауы көрнекті себептерсіз-ақ, нұсқауышқа айналып кетеді.

```
char* p = ch;
```

Бұл жерде p нұсқауышы **&ch[0]** адресімен инициалданады, ал **sizeof(p)** көлемі 4-ке тең (100 емес) болады.

Бұл қасиет пайдалы болып шығуы мүмкін. Мысалы, нөлмен аяқталатын символдар жиымындағы символдар санын есептейтін **strlen()** функциясын қарастырайық.

```
int strlen(const char* p)          // strlen() стандартты
                                   // функциясы сияқты
{
    int count = 0;
    While (*p) {++count; ++ p;}
    return count;
}
```

Енді оны **strlen(ch)** және **strlen(&ch[0])** аргументтерімен де шақыруға болады. Мұндай белгілеудің аз ғана артықшылығы бар екенін байқаған шығарсыз, оған біз де келісеміз. Жиым атауының нұсқауышқа айналып кету себептерінің бірі – бұл жердегі мәліметтердің үлкен көлемін мән бойынша жіберуден бас тарту болып табылады. Мысал қарастырайық:

```
int strlen(const char a[])         // strlen() стандартты
                                   // функциясы сияқты
strlen()
{
    int count = 0;
    while (a[count])  { ++count;}
    return  count ;
}
char lots[100000];

void f()
{
    int nchar = strlen(lots);
    // . . .
}
```

Біз алдын ала ойлағандай (жартылай негізсіз де емес), осы шақыруды орындаған кезде **strlen()** функциясының аргументі ретінде берілген жүз мың символ көшіріледі деген болжам орындалмайды. Оның орнына **char p[]** аргументін жариялау **char* p** жариялауымен баламалы (эквивалентті) болып қарастырылады, ал **strlen(lots)** шақыруы **strlen(&lots[0])** шақыруына эквивалент болып табылады. Бұл шығын шығарып көшіруді болдырмайды, бірақ ол сізді таңдандыратын шығар. Өйткені басқа кез келген жағдайда, объектіні жіберу кезінде сіз нақты түрде оның сілтеме бойынша (8.5.3–8.5.6 бөлімдерін қ.) жіберілуін талап етпесеңіз, онда объект көшірілетін болады.

Мынаған назар аударыңыз: жиым атынан құралған нұсқауыш оның бірінші элементіне орнатылады және айнымалы болып табылмайды, яғни, оған ешқандай мән меншіктей алмаймыз.

```
char ac [10];
ac=new char[20];
// қате: жиым атына ештеңені меншіктеуге болмайды
&ac[0]=new char [20];
// қате: нұсқауышш мәніне ештеңені меншіктеуге болмайды
```

Ал соңында компилятор анықтай алатын мәселені көрсетейік!

Жиым атының жанамалы түрде нұсқауышқа айналуына байланысты біз тіпті меншіктеу операторының көмегімен де жиымды көшіре алмаймыз.

```
int x[100];
int y[100];
// . . .
x = y;              // қате
int z[100] = y;     // қате
```

Бұл логикалық түрде қисынды, бірақ ыңғайсыз. Егер жиымды көшіру қажет болса, онда сіз бұдан күрделірек код жазуыңыз керек. Мысал қарастырайық:

```
for(int i=0; i<100; ++i) x[i]=y[i];
// int типті 100 сан көшіреміз
memcpy(x,y,100*sizeof(int));
// 100*sizeof(int) байт көшіреміз
copy(y,y+100,x);
// int типті 100 сан көшіреміз
```

С тілінде **вектор** жоқ болғандықтан, онда көбінесе жиымдар қолданылады. Осының нәтижесінде С++ тілінде жазылған көптеген программаларда **жиымдар** қолданылады (бұл туралы толығырақ 27.1.2 бөлімінде). Жеке жағдайларда С тілі стиліндегі тіркестер (нөлмен аяқталатын символдар жиымы. Бұл тақырып 27.5 бөлімінде қарастырылады) өте кең таралған.

Егер көшіруді орындағыңыз келсе, онда **vector** класы сияқты класты қолданыңыз. Жоғарыдағыларға эквивалентті **vector** класының объектілерін көшіру кодын келесі түрде жазуға болады:

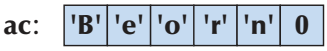
```
vector <int> x(100);
vector<int> y(100);
// . . .
x = y;          // int типті 100 сан көшіреміз
```

18.5.3 Жиымды инициалдау

Жиымдардың векторлар мен қолданушылар анықтаған басқа контейнерлерге қарағанда бір елеулі артықшылығы бар: C++ тілінде жиымдарды инициалдайтын мүмкіндік қарастырылған. Мысалы:

```
char ac[] = "Beorn"; // алты символдан құрылған жиым
```

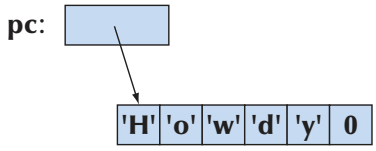
Осы символдарды санап шығыңыз. Олар бесеу, бірақ `ac` алты символдан тұратын жиым болып шығады, себебі компилятор тіркестік литерал соңына нөлді қосып жазады.



Нөлмен аяқталатын тіркес C тілі және басқа жүйелер үшін қалыпты жағдай болып табылады. Осындай нөлмен аяқталатын символдар жиымын *C тілі стиліндегі тіркестер* (C-style string) деп атайды. Барлық тіркестік литералдар C тілі стиліндегі тіркестер болып саналады. Мысал қарастырайық:

```
char* pc = "Howdy";  // pc нұсқауышы 6 символдан тұратын
                     // жиымға сілтеме жасайды
```

Графикалық түрде оны былай көрсетуге болады:



Сандық мәні `0` болып келетін `char` типті айнымалы — бұл `'0'` символы емес және әріп те емес, сан да емес. Бұл аяқтаушы нөлдің негізгі мақсаты функцияға тіркестің соңын табуға көмектесу болып табылады. Мынаны есте сақтаңыз: жиым өзінің көлемін білмейді. Соңғы нөлдік кодты пайдалануға сенім арта отырып, келесі мысалды жаза аламыз:

```
int strlen(const char* p)
// strlen() стандартты функциясына ұқсас
{
   int n = 0;
   while (p[n]) ++n;
   return n;
}
```

Негізінде, бізге **strlen()** функциясын анықтаудың еш қажеті жоқ, себебі ол **<string.h>** тақырыптық файлында анықталған стандарттық кітапхананың функциясы (27.5 және Б.10.3 бөлімдері). **strlen()** функциясы символдарды санайтынына, бірақ тіркес соңындағы нөлге еш мән бермейтініне назар аударыңыз; басқаша айтқанда, С тілі стиліндегі тіркесте *n* символдарды сақтау үшін **char** типті *n+1* айнымалыны сақтауға арналған жады көлемі болуы қажет.

Литералды тұрақтылар көмегімен тек символдық жиымдарды ғана инициалдауға болады, бірақ кез келген жиымды оның элементтері мәндерінің тізімі бойынша өздеріне сәйкес типте инициалдауға болады. Мысал қарастырайық:

```
int ai[] = {1,2,3,4,5,6};
// int типті алты саннан тұратын жиым
int ai2[100] = {0,1,2,3,4,5,6,7,8,9};
// қалған 90 элемент нөлмен инициалданады
double ad[100] = { };
// барлық элементтер нөлмен инициалданады
char chars[] = {'a','b','c'};
//тіркес соңындағы аяқтаушы нөл жоқ
```

ai жиымының элементтерінің саны алтыға (жетіге емес) тең, ал **chars** жиымының элементтерінің саны үшке (төртке емес) тең, "соңына **нөл** қосу" ережесі тек тіркестік литералдарға ғана жүреді. Егер жиым көлемі тікелей берілмесе, онда ол инициалдау тізімі бойынша анықталады. Бұл барынша пайдалы ереже болып табылады. Егер инициалдау тізіміндегі элементтер тізімі жиым элементтерінің санына қарағанда аз болса, онда қалған элементтер берілген типтегі келісім (үнсіз) бойынша қарастырылған мәндермен инициалданады.

18.5.4 Нұсқауыштарды қолданғанда туындайтын мәселелер

Қолданушылар бірсыпыра жағдайда нұсқауыштар мен жиымдарды қажеттіліктен тыс көп қолданады. Адамдар көбінесе нұсқауыштар мен жиымдарды қолдану арқылы өздеріне қиындықтар туғызып жатады. Жеке жағдайда нұсқауыштарға қатысты барлық негізгі мәселелер жады аймағын пайдалануға байланысты туындайды, ол қажетті типтегі объектіні дұрыс сақтамай жатады, сонымен қатар, көптеген мәселелер жиымның өз сақталу шекарасынан шығып кетуіне байланысты пайда болады. Осындай мәселелер тізімін қарастырайық:

- Нөлдік нұсқауышпен қатынасу;
- Инициалданбаған нұсқауышпен қатынасу;
- Жиым шекарасынан оның сыртына шығып кету;
- Өшірілген объектімен қатынасу;
- Көріну аймағынан шығып кеткен объектімен қатынасу.

Іс жүзінде жоғарыда көрсетілген мәселелердің барлығында да программалаушылар алдында тұрған басты мәселе – мәліметке нақты қол жеткізу (қатынасу, пайдалану), сырт қарағанда, дұрыс болып көрінгенмен, мұндайда көбінесе нұсқауыш дұрыс мәнді көрсетпей, басқа бір мәнге сілтеме жасап тұрады. Мұнан да жаманы (нұсқауыш көмегімен жазу кезінде), программамен байланыссыз басқа бір объект зиян шеккеннен кейін, проблема өте кеш анықталуы мүмкін. Келесі мысалды қарастырайық:

Компьютер жадына нөлдік нұсқауыш көмегімен қатынас жасамаңыз.

```
int* p = 0;
*p = 7;  // ой!
```

Әрине, егер инициалдау мен нұсқауышты пайдалану арасында қандай да бір код орналастырылса, онда жоғарыдағы жағдай (код) нақты программа ішінде орын алуы мүмкін. Көбінесе қателіктер **p** нұсқауышын функцияға бергенде немесе функция жұмысы нәтижесінде оны алғанда туындайды. Нөлдік нұсқауышты ешқайда да жібермеуге тырысыңыз, егер де ондайды істеп қалсаңыз, пайдалану алдында нұсқауышты тексеріп шығыңыз. Мысалы:

```
int* p = fct_that_can_return_a_0();
if (p == 0) {
    // бірдеңе жасаймыз
}
else {
    // p-ны пайдаланамыз
   *p = 7;
}
```

және

```
void fct_that_can_receive_a_0(int* p)
{
   if (p == 0) {
      // бірдеңе жасаймыз
   }
   else {
      // p-ны пайдаланамыз
      *p = 7;
   }
}
```

Нөлдік нұсқауыштарға байланысты қателерді болдырмауға мүмкіндік беретін негізгі құралдар болып сілтемелер (17.9.1 бөлімі) мен аластамалар (5.6 және 19.5 бөлімдері) саналады.

Нұсқауыштарды инициалдаңыз:

```
int* p;
*p = 9;                    // ой!
```

Жеке жағдайларда, класс мүшелері болып табылатын нұсқауыштарды инициалдауды ұмытпаңыз.

Жиымның жоқ элементтерін пайдалануға тырыспаңыз:

```
int a[10];
int* p = &a[10];
*p = 11;              // ой!
a[10] = 12;           // ой!
```

Циклдің бірінші және соңғы элементтерін пайдаланғанда, сақ болыңыз, оның алғашқы элементтері ретінде нұсқауыш арқылы берілетін жиымдарды қолданбауға тырысыңыз. Оның орнына **vector** класын қолданыңыз. Егер сізге іс жүзінде жиымды бірнеше функцияларда (оны аргумент бере отырып) қолдану керек болып жатса, ерекше сақ болып, жиым өлшемін беруді ұмытып кетпеңіз.

Жады аймағын өшірілген нұсқауыштар арқылы пайдаланып жүрмеңіз:

```
int* p = new int(7);
// . . .
delete p;
// . . .
*p = 13;              // ой!
```

Онан кейін орналасқан **delete p** нұсқауы немесе программалық код абайсызда ***p** мәнін қолдануы немесе оны жанамалы түрде пайдалануы мүмкін. Мұндай жағдайларды болдырмауға тырысу керек. Бұдан қорғанудың ең тиімді тәсілі "ештеңесіз" **delete** операторын орындауды талап ететін "ештеңесіз" **new** операторын орындауға тыйым салу болып табылады: **new** және **delete** операторларын конструкторлар мен деструкторларда орындаңыз немесе **Vector_ref** тәрізді контейнерді (E.4 бөлімі) пайдаланыңыз.

Нұсқауышты жергілікті (локальді) айнымалыға қайтармаңыз:

```
int* f()
{
    int x = 7;
    // . . .
    return &x;
}
```

```
// . . .

int* p = f();
// . . .
*p = 15;       // ой!
```

`f()` функциясынан қайтып оралу кезінде немесе онан кейін орналасқан код абайсызда `*p` мәнін қолдануы немесе оны жанамалы түрде пайдалануы мүмкін. Мұның себебі функцияда жарияланған локальді айнымалылар стекте функцияны шақыруға дейін орналасады да, одан шыққанда, өшіріледі. Жеке жағдайда, егер класс объектісі локальді айнымалы болатын болса, онда оның деструкторы шақырылады (17.5.1 бөлімді қ.). Локальді айнымалыларға нұсқауыштарды қайтаруға байланысты көптеген мәселелерді компиляторлар анықтай алмайды, дегенмен олардың кейбіреулерін анықтап та жатады.

Осыған эквивалентті мысал қарастырайық:

```
vector& ff()
{
     vector x(7);
     // . . .
     return x;
}    // мұнда x векторы жойылған болатын

// . . .

vector& p = ff();
// . . .
p[4] = 15;        // ой!
```

Тек кейбір компиляторлар ғана локальді айнымалыларға нұсқауыштарды қайтаруға байланысты туындайтын мәселелер түрлерін анықтай алады. Көбінесе программалаушылар мұны өз деңгейінде бағалай алмайды. Дегенмен көптеген тәжірибелі программалаушылар қарапайым жиымдар мен нұсқауыштарды пайдалану арқылы туындайтын шексіз мәселелерге кез болып, сәтсіздіктерге ұшырап жатады. Мұның шешімі, әрине, мынадай: өз программаларыңыздағы нұсқауыштар, жиымдар, `new` және `delete` операторларының санын азайтыңыз. Егер оны жасамасаңыз, онда жай ғана сақ болу бар мәселені шешпейді. Векторларға, RAII ("Resource Acquisition Is Initalization" – "Ресурс алу – бұл инициалдау"; 19.5 бөлімді қ.) тұжырымдамасына және де басқа жады аймағы мен өзге де ресурстарды басқарудың жүйелік тәсілдеріне көңіл бөліңіз.

18.6 Мысалдар: палиндром

Техникалық мысалдар жетер! Аздап ойландыратын басқа бір шағын есеп шығарайық. *Палиндром* (palindrome) – бұл солдан оңға қарай және оңнан солға қарай бірдей болып оқылатын сөз. Мысалы, *anna*, *petep*, *malayalam* сөздері палиндром болып табылады, ал *ida* және *homesick* сөздері палиндром емес. Сөздің палиндром екенін анықтайтын екі негізгі тәсіл бар:

- Қарама-қарсы жақта орналасқан әріптер көшірмесін жасап оны түпнұсқамен салыстыру арқылы;
- Бірінші әріптің соңғы әріппен, екінші әріптің соңғының алдындағы әріппен, т.с.с. сөз ортасына дейінгі әріптердің бір-бірімен сәйкесінше бірдей екенін тексеру арқылы.

Біз екінші тәсілді таңдап алайық. Бұл идеяны кодтар арқылы жүзеге асыруға болатын көптеген әдістер бар. Олар сөзді бейнелеу жолына және де символдарды біртіндеп салыстыру тәсілдеріне байланысты түрде өзгеріп отырады. Біз сөздің палиндром екенін бірнеше түрде тексеретін шағын программа жазамыз. Ол бізге программалау тілдерінің әртүрлі ерекшеліктерінің программаның сыртқы түріне және жұмысына қалай әсер ететінін білуге көмектеседі.

18.6.1 string класы арқылы жасалған палиндромдар

Алдымен біз `string` стандартты класын пайдаланатын программа нұсқасын жазайық, онда салыстырылатын әріптер индекстері `int` типіндегі айнымалы арқылы беріледі:

```
bool is_palindrome(const string& s)
{
    int first = 0;                  // бірінші әріп индексі
    int last = s.length()- 1;       // соңғы әріп индексі
    while (first < last) {
    // біз әлі сөз ортасына жеткен жоқпыз
        if (s[first]!=s[last]) return false;
        ++first;        // алға қарай
        --last;         // артқа қарай
    }
    return true;
}
```

Біз сөз ортасына жеткенше, әріптер айырмашылығы болмаса, `true` мәнін қайтарамыз. Сізге осы кодты қарап шығып, оның сөз тіркесінде әріп болмаған кез-

дерде, тіркес бір әріптен ғана тұратын сәтте, тіркестегі әріптер саны жұп және тақ болған кездерде дұрыс жұмыс істейтініне көз жеткізуді ұсынамыз. Әрине, біз тек логикаға сүйеніп қоя салмай, программаның дұрыс нәтиже беретініне сенімді болуымыз керек. Осы `is_palindrome()` функциясын пайдаланып көрейік:

```
int main()
{
  string s;
  while (cin>>s) {
      cout << s << " is";
      if (!is_palindrome(s)) cout << " not";
      cout << " a palindrome\n";
  }
}
```

Негізінде, біздің `string` класын пайдалану себебіміз – бұл кластың объектілері сөздермен (сөз тіркестерімен) жақсы жұмыс істейді. Олар бос орындармен бөлінген сөздерді жеңіл оқиды және өз мөлшерін (көлемін) де біледі. Егер біз `is_palindrome()` функциясын бос орындары бар тіркестерге қолданғымыз келсе, онда жай ғана `getline()` функциясы арқылы оларды оқитын едік (11.5 бөлімді қ.). Мұны *ah ha* және *as df fd sa* тіркестері мысалында көрсетуге болады.

18.6.2 Жиымдар арқылы жасалған палиндромдар

Егер бізде `string` (немесе `vector`) класы болмай, символдарды жиымда сақтауға тура келсе, не болар еді? Оны да қарастырайық:

```
bool is_palindrome(const char s[], int n)
      // s нұсқауышы n символдан тұратын жиымның бірінші
                                  // символына сілтеп тұр
{
   int first = 0;            // бірінші әріп индексі
   int last = n-1;           // соңғы әріп индексі
   while (first < last) {
                  // біз әлі сөз ортасына жеткен жоқпыз
         if (s[first]!=s[last]) return false;
         ++first;                // алға қарай
         --last;                 // артқа қарай
   }
   return true;
}
```

is_palindrome() функциясын орындау үшін, алдымен символдарды жиымға жазып алу керек. Қауіпсіз тәсілдердің бірі (жиымның толып кетуін болдырмайтын) мынадай болады:

```
istream& read_word(istream& is, char* buffer, int max)
            // buffer жиымына max-1 мөлшерінен артпайтын
                                  // символдар санын оқиды
{
    is.width(max);      // келесі >> операторын орындағанда,
    // max-1 мөлшерінен артпайтын символдар саны оқылады
    is >> buffer;
    // бос орындармен бөлінген сөзді оқып, соңғы символдан
    // соң нөл қосып жазамыз
    return is;
}
```

istream ағымының енін дұрыс орнату келесі >> операторын орындау кезінде жиымның толып кетуін болдырмайды. Өкінішке орай, бұл символдарды оқу бос орынмен аяқталды ма, жоқ әлде буфер толды ма (сондықтан бізге оқуды жалғастыра беру керек), ол жағының бізге белгісіз екенін білдіреді. Оның үстіне, **width()** функциясының енгізу кезіндегі өз ерекшеліктерін кім есіне сақтап отыр? Негізінде, **string** және **vector** стандартты кластары буферлік енгізуге қарағанда, жақсырақ болып саналады, өйткені олар енгізу кезіндегі буфердің көлемін реттей алады. Тіркесті аяқтайтын соңғы 0 символы қажет болады, өйткені символдар жиымдарымен орындалатын көптеген операциялар жиымды нөлмен аяқталады деп санайды. **read_word()** функциясын пайдалана отырып, келесі кодты жазуға болады:

```
int main()
{
      const int max = 128;
      char s[max];
      while (read_word(cin,s,max)) {
          cout << s << " is";
          if (!is_palindrome(s,strlen(s))) cout << " not";
          cout << " a palindrome\n";
      }
}
```

read_word() функциясын шақыруды орындағаннан кейін **strlen(s)** функциясын шақыру жиымдағы символдар санын қайтарады, ал **cout<<s** нұсқауы жиымдағы **нөл**мен аяқталатын символдарды экранға шығарады.

Есептерді **string** класы арқылы шығару оларды жиымдар көмегімен шығаруға қарағанда, ыңғайлырақ болып табылады. Бұл ұзын тіркестермен жұмыс істеу кезінде айқынырақ көрінеді (10 жаттығуды қ.).

18.6.3 Нұсқауыштар арқылы жасалған палиндромдар

Символдарды айқындау (идентификациялау) үшін индекстерді пайдалану орнына нұсқауыштарды қолдануға болады:

```
bool is_palindrome(const char* first, const char* last)
  // first нұсқауышы бірінші әріпке сілтеме жасайды
  // last нұсқауышы соңғы әріпке сілтеме жасайды
{
  while (first < last) {   // біз әлі сөз ортасына жеткен жоқпыз
    if (*first!=*last) return false;
    ++first;       // алға қарай
    --last;        // артқа қарай
  }
  return true;
}
```

Біз нұсқауыштарды инкременттей де, декременттей де алатынымызды айта кетейік. Инкременттеу нұсқауышты жиымның келесі элементіне, ал декременттеу алдыңғы элементіне алып барады. Егер жиымда алдыңғы немесе соңғы элемент болмаса, онда жиымның берілген шегінен шығып кетуге байланысты қате туындайды. Бұл нұсқауыштарды пайдаланудан шығатын тағы бір мәселе болып табылады.

Біз `is_palindrome()` функциясын былай шақыра аламыз:

```
int main()
{
  const int max = 128;
  char s[max];
  while (read_word(cin,s,max)) {
        cout << s << " is";
        if (!is_palindrome(&s[0],&s[strlen(s)-1]))
        cout << " not";
        cout << " a palindrome\n";
  }
}
```

Біз ойын ретінде `is_palindrome()` функциясын былай етіп қайта жазайық:

```
bool is_palindrome(const char* first, const char* last)
   // first нұсқауышы бірінші әріпке сілтеме жасайды
   // last нұсқауышы соңғы әріпке сілтеме жасайды
{
```

```
    if (first<last) {
       if (*first!=*last) return false;
       return is_palindrome(++first,--last);
     }
     return true;
}
```

Палиндромның анықтамасын аздап өзгертіп төмендегідей етіп жазар болсақ, бұл кодтың дұрыстығы талас тудыра қоймас: егер сөздің алдыңғы жақтағы және соңғы жақтағы символдары бірдей болса, оларды алып тастағанның өзінде де, қалған сөз де палиндром болатын болса, онда бұл сөз палиндром болып табылады.

ТАПСЫРМА

Бұл тарауда біз екі тапсырма береміз: олардың біреуін жиымдар арқылы, ал екіншісін – векторлар арқылы орындау керек. Екі тапсырманы да орындап болған соң, оларға жұмсаған күшіңізді салыстырып көріңіз.

Жиымдарға арналған тапсырма:

1. 1,2,4,8,16, т.с.с. сандармен инициалданған он элементтен тұратын `int` типіндегі глобальді `ga` жиымын анықтаңыз.
2. Аргумент ретінде `int` типіндегі жиымды және жиым элементтері санын беретін `int` типіндегі айнымалыны қабылдайтын `f()` функциясын анықтаңыз.
3. `f()` функциясында мыналарды орындаңыз:
   a. Он бүтін саннан тұратын `int` типіндегі локальді `la` жиымын анықтаңыз;
   b. `ga` жиымының мәндерін `la` жиымына көшіріңіз;
   c. `la` жиымы элементтерін баспаға шығарыңыз;
   d. `int` типіндегі айнымалыға сілтеме жасап тұрған `p` нұсқауышын анықтаңыз да, оны бос жады аймағында орналасқан жиым адресімен инициалдаңыз. Бұл жиымда сақталатын элементтердің саны функция аргументі болып келетін жиым элементтері санымен бірдей болып табылады;
   e. Функция аргументі болып келетін жиым мәндерін бос жады аймағында орналасқан жиымға көшіріп жазыңыз;
   f. Бос жады аймағында орналасқан жиым элементтерін баспаға шығарыңыз;
   g. Бос жады аймағында орналасқан жиымды өшіріңіз.
4. `main()` функциясында келесі әрекеттерді орындаңыз:
   a. `ga` аргументі бар `f()` функциясын шақырыңыз;
   b. Он элементі бар `aa` жиымын анықтаңыз да, оны факториалдың (1, 2*1, 3*2*1, 4*3*2*1, т.с.с.) алғашқы он мүшелері мәндерімен инициалдаңыз;
   c. `aa` аргументі бар `f()` функциясын шақырыңыз.

Стандартты `векторға` арналған тапсырма:

1. Глобальді `vector<int> gv` стандартты векторын анықтаңыз да, оны он бүтін санмен: 1, 2, 4, 8, 16, т.с.с. инициалдаңыз.
2. `vector<int>` типіндегі аргументті қабылдайтын f() функциясын анықтаңыз.
3. `f()` функциясында мыналарды орындаңыз:
   a. Элементтерінің саны функция аргументі болып табылатын вектордағымен бірдей болып келетін локальді `vector<int> lv` векторын анықтаңыз;
   b. `gv` векторының мәндерін `lv` векторына көшіріңіз;
   c. `lv` векторының элементтерін баспаға шығарыңыз;
   d. Локальді `vector<int> lv2` векторын анықтаңыз; оны функция аргументі болып табылатын вектордың көшірмесімен инициалдаңыз;
   e. `lv2` векторы элементтерін баспаға шығарыңыз.
4. 4. `main()` функциясында келесі әрекеттерді орындаңыз:
   a. `gv` аргументі бар `f()` функциясын шақырыңыз;
   b. `vector<int> vv` векторын анықтаңыз да, оны факториалдың (1, 2*1, 3*2*1, 4*3*2*1, т.с.с.) алғашқы он мәндерімен инициалдаңыз;
   c. `vv` аргументі бар `f()` функциясын шақырыңыз.

БАҚЫЛАУ СҰРАҚТАРЫ

1. "Сатып алушы, сақ бол!" деген өрнек нені білдіреді?
2. Келісім (үнсіз) бойынша класс объектілерін көшірудің қандай түрі қолданылады?
3. Келісім (үнсіз) бойынша қолданылатын класс объектілерін көшіру қашан қажет болады, ал қашан қажет емес?
4. Көшіру конструкторы деген не?
5. Көшіретін меншіктеу деген не?
6. Көшіретін меншіктеу мен көшіретін инициалдаудың бір-бірінен айырмашылығы неде?
7. Шектелген көшіру деген не? Тереңдетілген көшіру деген не?
8. `vector` класы объектісінің көшірмесі өз прототипімен қалай салыстырылады?
9. Класпен орындалатын бес операцияны тізбелеп көрсетіңіз.
10. `explicit` түйінді сөзі бар конструктор нені білдіреді? Оны қашан келісім бойынша конструктордан артық деп айтуға болады?
11. Класс объектісіне жанамалы түрде қандай операциялар қолдануға болады?
12. Жиым деген не?
13. Жиымды қалай көшіруге болады?

14. Жиымды қалай инициалдауға болады?
15. Қай кезде аргументке нұсқауыш беру оны сілтеме арқылы беруден басымырақ болады?
16. С стиліндегі тіркес немесе С-тіркес деген не?
17. Палиндром деген не?

ТЕРМИНДЕР

`explicit` конструкторы	көшіретін меншіктеу	палиндром
жиым	көшіру конструкторы	тереңдетілген көшіру
жиымды инициалдау	негізгі операциялар	шектелген көшіру
келісім бойынша конструктор		

ЖАТТЫҒУЛАР

1. Бос жады аймағына, онда орын босата отырып, С тілі стилінде тіркесті көшіретін `char* strdup(const char*)` функциясын жазып шығыңыз. Ешқандай стандартты функцияларды қолданбаңыз. Индекстеуді пайдаланбаңыз, оның орнына `*` атаусыз ету операторын қолданыңыз.
2. `x` ішкі тіркесінің `s` тіркесіне алғашқы кіруін С тілі стилінде анықтайтын `char* findx(const char* s, const char* x)` функциясын жазып шығыңыз. Ешқандай стандартты функцияларды қолданбаңыз. Индекстеуді пайдаланбаңыз, оның орнына `*` атаусыз ету операторын қолданыңыз.
3. С тілі стилінде екі тіркесті салыстыратын `int strcmp(const char* s1, const char* s2)` функциясын жазып шығыңыз. Егер лексикографикалық мағынада `s1` тіркесі `s2` тіркесінен кіші болса, функция теріс мән қайтаруы тиіс, егер олар тең болса, нөл мәнін, ал `s1` тіркесі `s2` тіркесінен лексикографикалық мағынада үлкен болатын болса, функция оң мән қайтаруы тиіс. Ешқандай стандартты функцияларды қолданбаңыз. Индекстеуді пайдаланбаңыз, оның орнына `*` атаусыз ету операторын қолданыңыз.
4. Егер `strdup()`, `findx()` және `strcmp()` функцияларына аргумент ретінде С тілі стиліндегі тіркесті бермесе, не болар еді? Алдымен нөлмен аяқталатын символдар жиымына сілтеме жасамайтын `char*` нұсқауышын қалай алуға болатынын анықтап алу керек, сонан соң оны пайдалану қажет (оны ешқашанда нақты кодта (тәжірибелік емес) жасамаңыз; ол апаттық жағдайға алып келуі мүмкін). Бос жады аймағында немесе стекте орналасқан С стиліндегі дұрыс берілмеген тіркестермен тәжірибелер жа-

сап көріңіз. Егер нәтижелері дұрыс болып көрініп жатса, жөндеп түзету режимін алып тастаңыз. Барынша көп болатындай етіп алынған тіркестегі символдар санын тағы бір аргумент етіп алып, үш функцияны да қайта жасап, орындап шығыңыз. Сонан соң функцияларды C тілі стилінде дұрыс және қате құрылған тіркестермен орындап тесттен өткізіңіз.

5. Ортасына нүкте қойылған екі тіркесті біріктіріп (конкатенация) жазатын `string cat_dot(const string& s1, const string& s2)` функциясын жазып шығыңыз. Мысалы, `cat_dot("Нильс","Бор")` функциясы `Нильс.Бор` тіркесін қайтарады.

6. Алдыңғы жаттығудағы `cat_dot()` функциясын түрлендіріп, ол өзінің үшінші аргументі ретінде ажыратқыш таңба (нүкте емес) болып саналатын тіркесті алатындай ету керек.

7. Алдыңғы жаттығулардағы `cat_dot()` функциясының түрлендірілген нұсқаларын жазыңыз. Ол өзінің аргументтері ретінде C тілі стиліндегі тіркестерді қабылдап алып, бос жады аймағында орналасқан C тілі стиліндегі тіркестерді қайтарады. Ешқандай стандартты функцияларды немесе типтерді қолданбаңыз. Индекстеуді пайдаланбаңыз, оның орнына `*` атаусыз ету операторын қолданыңыз. Сол функцияларды бірнеше тіркестер арқылы орындап тесттен өткізіңіз. `new` операторы арқылы алып пайдаланылған жады көлемінің `delete` операторы көмегімен босатылғанына көз жеткізіңіз. Осы жаттығуға жіберген мүмкіндіктеріңізді 5 және 6 жаттығуларды орындауға жұмсаған күш-жігеріңізбен салыстырыңыз.

8. 18.6 бөлімде келтірілген функцияларды салыстыру үшін тіркестердің кері көшірмесін пайдалана отырып, қайта жазып шығыңыз. Мысалы, `"home"` тіркесін енгізіңіз де, одан `"emoh"` кері тіркесін алып, бұл *home* сөзінің палиндром емес екеніне көз жеткізу үшін, осы екі сөзді салыстырып шығыңыз.

9. 17.3 бөлімде сипатталған компьютер жадын бөлу (тарату) сұлбасын талдап шығыңыз. Статикалық жадының, стектің және бос жады аймағының қандай реттілікпен бөлініп берілетінін хабарлайтын программа жазыңыз. Стек қай бағытқа қарай өседі: жоғарғы адреске қарай ма, әлде төменгі адреске қарай ма? Жиым бос жады аймағында орналасқан делік, оның қай элементінің адресі үлкен болады – жоғарғы индекстісі ме, әлде төменгі индекстісі ме?

10. 18.6.2 бөліміндегі жиым негізіндегі палиндром туралы есептің шығарылуын талдап шығыңыз. Оны ұзын тіркестермен жұмыс істейтіндей етіп өзгертіңіз: 1) егер енгізілген тіркес өте ұзын болып кетсе, мәлімет шығарыңыз; 2) кез келген ұзын тіркестерді пайдалануға рұқсат беріңіз. Осы екі нұсқаның да күрделілігіне түсінік беріңіз.

11. *Ала тізім* (skip list) дегеннің не екенін ұғуға тырысыңыз және оның бір түрін іске асырыңыз. Бұл жай жаттығу емес.

12. "Вампус аулау" (немесе жай ғана Вамп) ойынының бір нұсқасын іске асырыңыз. Бұл – Грегори Йоб (Gregory Yob) ойлап шығарған қарапайым компьютерлік (графикалық емес) ойын. Ойынның мақсаты – қараңғы үңгірдегі лабиринтте тығылып жүрген әжептеуір ақылды монстрды іздеп тауып алу. Сіздің мақсатыңыз – садақ пен жебенің көмегімен вампусті өлтіру. Үңгірде вампустан басқа тағы екі қауіп бар: түпсіз терең шұңқырлар мен алып жарғанаттар. Егер сіз терең шұңқыры бар бөлмеге тап болсаңыз, сіз үшін ойын аяқталады. Ал егер жарғанаты бар бөлмеге кірсеңіз, онда ол сізді ұстап алып, басқа бөлмеге лақтырады. Егер де сіз вампусы бар бөлмеге кірсеңіз немесе ол сіз тұрған бөмеге кіріп келсе, ол сізді жеп қояды. Бөлмеге кіргенде, сіз мұнда болатын қауіп туралы ескерту аласыз.

    "Мен вампус иісін сезіп тұрмын" – ол көрші бөлмеде дегенді білдіреді.

    "Мен жел соққанын сезіп тұрмын" – ол көрші бөлмеде шұңқыр бар екенін білдіреді.

    "Мен жарғанаттың ұшқанын естіп тұрмын" – ол көрші бөлмеде тұрады дегенді білдіреді.

    Сізге ыңғайлы болуы үшін бөлмелер нөмірленіп қойылған. Әрбір бөлме басқа үш бөлмемен жерасты туннелі арқылы байланыстырылған. Сіз бөлмеге кірісімен, бірден хабар аласыз, мысалы: "Сіз 12-бөлмедесіз, ол туннель арқылы 1, 13 және 4-бөлмелермен жалғасқан: барасыз ба, әлде атасыз ба?". Мүмкін жауаптар: **`m13`** ("13-бөлмеге бару") және **`s13-4-3`** ("13, 4 және 3-бөлмелер арқылы ату"). Жебе үш бөлме арқылы ұшып бара алады. Ойын басында сізге 5 жебе беріледі. Атудың бір жаман жері сіз вампусті оятып жіберуіңіз мүмкін, онда ол ұйықтап жатқан бөлмесіне жақын көрші бөлмеге кіре алады, ал ол бөлме – сіз тұрған бөлме болуы ықтимал.

    Бұл ойынның ең қиын бөлігі үңгірді және басқа бөлмелермен байланысқан бөлмені таңдап алуды программалау болып табылады. Мүмкін сіз программаны іске қосқан сайын әртүрлі үңгірлер және жарғанаттар мен вампустардың да әртүрлісін таңдап алғыңыз келер, онда **`std_lib_facilities.h`** кітапханасындағы кездейсоқ сандар генераторын пайдалануды қаларсыз. Көмек: лабиринттің қалыпты жағдайын тексеріп алу үшін *түзетіп жөндеу* (debug) режимін қолданыңыз.

СОҢҒЫ СӨЗ

Стандартты **vector** класы нұсқауыштар мен жиымдар сияқты компьютер жадын төменгі деңгейде басқару құралдарын қолдануға негізделген. Оның негізгі атқаратын қызметі – программалаушыларға осындай жады аймағын басқару құралдарына байланысты қиындықтарды болдырмауға көмектесу болып табылады. Кез келген класты құра отырып, сіз оның объектілерін инициалдауды, көшіруді және өшіруді қарастыруыңыз керек.

Векторлар, шаблондар және аластамалар

"Табыс ешқашанда соңғы болмайды".

– Уинстон Черчиль (Winston Churchill)

Бұл тарауда біз STL кітапханасындағы ең белгілі және пайдалы контейнер – **`vector`** класын жобалау мен жүзеге асыру мәселелерін қарастыруды аяқтаймыз. Мұнда біз элементтерінің саны айнымалы контейнерлерді қалай жүзеге асыруға болатынын, типі параметр ретінде берілетін контейнерлердің қалай сипатталатынын және берілген аймақтан шығып кеткен кездерде пайда болатын қателерді қалай өңдеуге болатынын көрсетеміз. Әдеттегідей, мұнда қарастырылған тәсілдер **`vector`** класын, тіпті контейнерлердің өзін жүзеге асыру шеңберінен де тысқары шығып кететін әмбебап сипатта болады. Негізінде, біз әртүрлі типтегі мәліметтердің айнымалы көлемдерімен қалай қауіпсіз жұмыс істеуге болатынын көрсетеміз. Бұған қоса, жобалау мысалдарында нақты (шынайы) мүмкіндіктерді есепке алуға тырысамыз. Біздің программалау технологиямыз шаблондар мен аластамаларға негізделген, сондықтан біз шаблондарды қалай анықтауға болатынын және аластамалармен тиімді жұмыс істеуде өзекті рөл атқаратын ресурстарды басқарудың негізгі тәсілдерін көрсетеміз.

19.1 Мәселелер

18-тараудың соңында біздің `vector` класын жасаудағы жұмысымыз келесі операцияларды орындай алатын кезеңге жетті:

- Элементтері екі еселенген дәлдіктегі кез келген элементтер саны бар жылжымалы нүктелі сандар болып келетін `vector` класы объектілерін жасау;
- Меншіктеу мен инициалдау арқылы `vector` класы объектілерін көшіру;
- `vector` класы объектілері орналасқан жады аймағын ол көріну шегінен шығып кеткен кездерде дұрыстап босата білу;
- Қарапайым индекстік белгілерді пайдалану арқылы `vector` класы объектілері элементтерін қолдана білу (меншіктеу операторының оң жақтағы және сол жақтағы бөліктерінде де).

Мұның бәрі де жақсы әрі пайдалы, бірақ күрделіліктің күтілген деңгейіне шығу үшін біз тағы бірнеше мәселелерді шешуіміз керек.

- `vector` класы объектілері көлемін (оның элементтері санын) қалай өзгерту керек?

- **vector** класы объектілері шегінен шығып кетуге байланысты туындайтын қателерді қалай тауып өңдеуге болады?
- **vector** класы объектілерінің элементтері типін аргумент ретінде қалай беруге болады?

Мысалы, **vector** класын келесі кодтарды жазу мүмкін болатындай етіп қалай анықтау керек:

```
vector<double> vd;              // double типті элементтер
double d;
while(cin>>d) vd.push_back(d);
// барлық элементтерді сақтау үшін
// vd-ны өсіру керек

vector<char> vc(100);           //  char типті элементтер
   int n;
   cin>>n;
   vc.resize(n);       // n элементі бар vc объектісін жасау
```

Әрине, векторлармен мұндай операциялар орындау өте пайдалы шығар, бірақ ол неге программалау тұрғысынан қарағанда маңызды? Неге ол программалау тәсілдерінің стандартты жиынына кіргізуге тұрарлық болып саналады? Әңгіме мұндай операциялардың екі жақты икемділігінде болып отыр. Бізде бір ғана негіздеletін нәрсе бар, ол төмендегі екі тәсілмен өзгертуге болатын **vector** класының объектісі:

- Элементтері санын өзгерту.
- Элементтері типін өзгерту.

Бұл өзгерту түрлері өте керекті болып саналады және олар іргелі сипатта болады. Біз әрқашанда мәлімет жинаймыз. Өз жұмыс столымды көзбен шалсам, банк шот қағаздары, кредиттік кәртішкелер және телефонның төлем қағаздары жатыр. Осы шоттар мен қағаздардың әрқайсысы әртүрлі типтегі сандардан, әріптерден құралған мәліметтері бар жолдар тізімінен тұрады. Көз алдымда телефон тұр, онда да адамдардың аттары мен телефон нөмірлері сақталған. Кітап жәшігі сөрелерінде кітаптар тізіліп тұр. Біздің программалар да осыларға ұқсас: оларда әртүрлі типтегі элементтерден тұратын контейнерлер сипатталған. Әртүрлі мәліметтерді: телефон нөмірлерін, адамдардың аты-жөндерін, банк операциялары қағаздарын және де басқа құжаттарды сақтайтын контейнерлер (**vector** класы басқалардан гөрі жиірек қолданылады) бар. Негізінде, менің столымда жатқан заттардың бәрі де әйтеуір бір компьютерлік программалар көмегімен алынған.

Бұлардың ішіндегі ең ерекшесі телефон: ол *өзі* компьютер, ондағы телефон нөмірлерін қайталап қарағанымызда, біз өзіміз жазған программа сияқты басқа бір программаның нәтижесін көреміз. Мұндай нөмірлерді **vector<Number>** класының объектілерінде сақтаған өте ыңғайлы болып табылады.

Әрине, барлық контейнерлердің элементтер саны бірдей емес. Инициалдау кезінде мөлшері анықталатын векторлармен жұмыс ісеуге бола ма, яғни **resize()**, **push_back()** функцияларын немесе соларға парапар басқа операцияларды пайдаланбай, өз кодымызды жаза алар ма едік? Әрине, жаза алар едік, бірақ ол программалаушыға тіпті керек емес жұмыс істеуге әкелер еді: көлемі тұрақты контейнерлермен жұмыс істеу кезіндегі негізгі қиындық алғашқы тағайындалған көлем артып, олардың саны өте үлкен болып кеткенде, элементтерді үлкенірек контейнерге ауыстыру болып табылады. Мысалы, біз вектор көлемін өзгертпей, келесі код арқылы оны мәліметтермен толтыра алар едік:

```
// push_back функциясын қолданбай, векторды элементтермен
                                                  // толтыру:
vector<double>* p = new vector<double>(10);
int n = 0;        // элементтер саны
double d;
while(cin >> d)  {
   if (n==p->size())  {
      vector<double>* q = new vector<double>(p->size()*2);
      copy(p->begin(), p->end(), q->begin());
      delete p;
      p = q;
   }
   (*p)[n] = d;
   ++n;
}
```

Бұл әдемі емес. Оның үстіне бұл кодтың дұрыс жұмыс істейтініне сіз сенімдісіз бе? Қалай сенімді болуға болады? Біздің кенеттен нұсқауыштарды және компьютер жадын тікелей басқаруды пайдалана бастағанымызға көңіл бөліңіз. Біз тұрақты көлемдегі объектілермен (жиымдармен; 18.5 бөлімді қ.) жұмыс істеу кезінде машиналық деңгейге жақын программалау стилінің жұмысын көрсету үшін мұны істеуге мәжбүр болдық. **vector** класы сияқты контейнерлерді қолдануға негіз болған себептердің бірі жақсырақ бір программа жасау болатын; басқаша айтқанда, **vector** класының өзі контейнер көлемін өзгертіп, қолданушыларды мұндай жұмыстан босатып, қате шығу мүмкіндігін азайтқымыз келген. Сонымен, бізге қанша элемент керек болса, сонша элементті сақтап, өз көлемін өзі өзгерте алатын контейнерлер болса деп қалар едік. Мысал қарастырайық:

```
vector<double> d;
double d;
while(cin>>d) vd.push_back(d);
```

Контейнердің көлемін өзгерту қандай деңгейде кең таралған? Егер мұндай жағдай сирек кездесетін болса, онда олар үшін арнайы құрал қарастыру қажет

болмас еді. Бірақ көлемді өзгерту өте жиі кездеседі. Ең қарапайым мысал – енгізу ағымынан саны белгісіз мәндерді оқу. Басқа мысалдарға іздеу нәтижелерін жинақтауды (бізге алдын ала олардың қанша болатыны белгісіз ғой) және оларды біртіндеп жинақтан өшіру ісін жатқызуға болады. Сонымен, мәселе контейнер көлемін өзгертуді жасауда емес, оны қалай істеу керек дегенге келіп саяды.

Біз неге осы контейнердің көлемін өзгерту тақырыбын қозғадық? Неге контейнерге қажетті жады аймағын беріп, сонымен жұмыс істей бермейміз?! Осындай стратегия қарапайым әрі тиімді болып көрінеді. Дегенмен, бұл егер біз өте үлкен жады көлемін сұрамағанда ғана дұрыс болып табылады. Бұл стратегияны таңдап алған программалаушылар өз программаларын қайта жазуға мәжбүр болады (егер олар мұқият және жүйелі түрде жадының толып кетуін қадағалайтын болса) немесе адам айтқысыз келеңсіздіктерге кездеседі (егер олар жадының толуын тексермейтін болса).

Әрине, **vector** класының объектілері екі еселік дәлдіктегі сандарды, температура мәндерін, жазбаларды (әр түрдегі), сөз тіркестерін, операцияларды графикалық қолданушы интерфейсі батырмаларын, фигураларды, күн-ай мерзімдерін, терезелерге нұсқауыштарды және т.б. сақтауы керек. Бұларды тізбелеуді шексіз жалғастыра беруге болады. Контейнерлер де әр түрде болады. Бұл бізді нақты контейнердің түрін таңдап алу үшін жақсылап ойландыратын әжептеуір жалғасы бар маңызды нәрсе. Неге барлық контейнерлер вектор түрінде бола алмайды? Егер біз контейнердің тек бір түрімен ғана айналысатын болсақ, онда онымен орындалатын операцияларды программалау тілінің бір бөлігі етіп жіберер едік. Оған қоса, бізге басқа контейнерлермен айналысу да қажет болмас еді, біз әрқашанда тек **vector** класын пайдаланар едік.

Мәліметтер құрылымы қолданбалы программалардың көбісінде маңызды рөл атқарады. Мәліметтерді қалай ұйымдастыру керектігі жайлы көптеген қалың әрі пайдалы кітаптар жазылған. Олардың басым бөлігінде "Мәліметтерді жақсылап қалай сақтау керек?" деген сұрақ қарастырылады. Жауап біреу – бізге үлкен мөлшерде және әртүрлі контейнерлер керек, бірақ бұл біздің кітапта өз деңгейінде қарастыруға көнбейтін тым ауқымды тақырып. Әйтсе де, біз **vector** және **string** (**string** – бұл символдар контейнері) кластарын кең қолдандық. Келесі тарауларда біз **list**, **map** (**map** класы – екі мән сақталатын бұтақ) кластарын және матрицаларды сипаттайтын боламыз. Бізге әртүрлі контейнерлер керек болатындықтан, оларды сүйемелдеу үшін соларға сәйкес құралдар мен программалау технологиялары қажет болады. Мәліметтерді сақтау технологиялары мен олармен қатынас құруды (пайдалануды) ұйымдастыру есептеудің ең іргелі және күрделі формаларының бірі болып табылады.

Компьютерлік жады деңгейінде барлық объектілер тұрақты мөлшерде болып, олардың типтері болмайды. Мұнда біз элементтерінің саны айнымалы болып келетін әртүрлі типтегі объект контейнерлерін құруға мүмкіндік беретін программалау тілі мен технологиясының құралдарын қарастырамыз. Бұл программалардың икемділігі мен программалауды жеңілдетуді қамтамасыз етеді.

19.2 Мөлшерді өзгерту

Мөлшерді (көлемді) өзгерту үшін кітапханадағы стандартты **vector** класының қандай мүмкіндіктері бар? Онда қарапайым үш операция қарастырылған. Мысалы, программада **vector** класының келесі объектісі жарияланған болсын делік:

```
vector<double> v(n); // v.size()==n
```

біз мұның көлемін үш түрлі тәсілмен өзгерте аламыз:

```
v.resize(10);      // енді v-да 10 элемент бар

v.push_back(7);    // мәні 7-ге тең элементті
                   // v объектісінің соңына қосамыз
                   // v.size() көлемі бірге артады
v = v2;   // басқа вектор меншіктейміз; v – енді v2 көшірмесі
         // енді v.size() == v2.size()
```

Кітапханадағы стандартты **vector** класының вектор көлемін өзгертетін басқа да операциялары бар, мысалы, **erase()** және **insert()** (Б.4.7 бөлімін қ.), бірақ біз бұл жерде вектормен осы үш операцияны орындауды қалай жүзеге асыруға болатынын көрсетеміз.

19.2.1 Бейнелеу

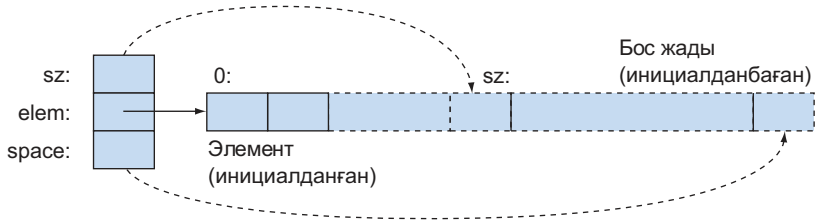
19.1 бөлімде біз көлемді өзгертудің қарапайым стратегиясын көрсеткен болатынбыз: жаңа элементтер санына арналған жады аймағын бөліп, соған ескі элементтерді көшіріп жазу. Бірақ контейнер көлемі жиі өзгеретін болса, онда мұндай стратегия тиімді болмайды. Практика жүзінде біз көлемді бір рет өзгертеміз де, оны кейін бірнеше рет қайталаймыз. Программаларда **push_back()** функциясын бір-ақ рет шақыру өте сирек кездеседі.

Сонымен, біз контейнер көлемін өзгертуді қарастырып, өз программамызды жақсарта аламыз. Негізінде **vector** класының барлық нұсқаларында элементтер саны да, одан кейінірек керек болатын бос жады аймағы да қатар қарастырылады. Мысалы:

```
class vector {
      int sz;              // элементтер саны
      double* elem;        // бірінші элемент адресі
      int space;
     // элементтер саны плюс "бос жады аймағы"/"слоттар"
     // жаңа элементтер үшін ("ағымдағы жады")
```

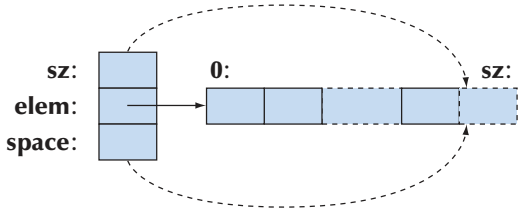
```
public:
      // . . .
};
```

Мұны графикалық түрде былай көрсете аламыз:



Элементтер нөмірі нөлден басталатын болғандықтан, **sz** айнымалысы (элементтер саны) соңғы элементтен кейін орналасқан ұяшыққа сілтеме жасайды, ал **space** айнымалысы соңғы слоттан кейін тұрған ұяшыққа сілтеме жасайды. Осыларға **elem+sz** және **elem+space** ұяшықтарына орнатылған нұсқауыштар сәйкес келеді.

Вектор алғаш рет құрылғанда, **space** айнымалысы нөлге тең болады.

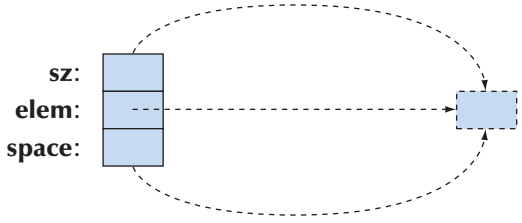


Біз элементтер саны өзгермегенше, қосымша слоттарды белгілеуді бастамаймыз. Әдетте, ол **space==sz** шарты орындалған сәттен басталады. Осыған орай, **push_back()** функциясын қолдана отырып, біз берілген жады аймағынан шығып кетпейміз.

Келісім бойынша конструктор (элементтері жоқ **vector** класы объектілерін жасайтын) кластың барлық үш мүшесін де нөлге теңестіреді:

```
vector::vector() :sz(0), elem(0), space(0) { }
```

Бұл ситуация былай болады:



"Шектен тысқары элемент" мұнда тек ойда тұрады. Келісім бойынша конструктор бос жады аймағын бөлмейді де, ең кіші көлемді алып тұрады (16-жаттығуға қ.).

Біздің `vector` класымыз стандартты векторды (басқа да мәліметтер құрылымын) жасауға болатын тәсілді көрсетеді, бірақ стандартты кітапханада жасалғандар өздерінің әртүрлілігімен ерекшеленеді, сондықтан бұл жүйедегі `std::vector` класы басқа стратегияны пайдалануы мүмкін.

19.2.2 `reserve` және `capacity` функциялары

Контейнер көлемін өзгерту (яғни, элементтер санын өзгерту) кезіндегі ең басты операция болып `vector::reserve()` функциясы саналады. Ол жаңа элементтерге арналған жады аймағын қосады:

```
void vector::reserve(int newalloc)
{
   if (newalloc<=space) return;      // көлем азаймайды
   double* p = new double[newalloc];
   // жаңа жады аймағын бөлеміз
   for (int i=0; i<sz; ++i) p[i] = elem[i];
   // ескі элементтерді көшіру
   delete[] elem;    // бұрынғы (ескі) жады аймағын босату
   elem = p;
   space = newalloc;
}
```

Бөлінген жады аймағындағы элементтерді инициалдамағанымызға назар аударыңыз. Біз тек жады аймағын бөлдік, ал оны қалай пайдалану – `push_back()` және `resize()` функцияларының жұмысы.

Әрине, `vector` класы объектісіндегі қолдануға болатын бос жады аймағының мөлшері қолданушыны қызықтырмай қоймайды, сондықтан біз стандартты кластағы тәрізді осындай ақпарат беретін функция-мүше қарастырдық:

```
int vector::capacity() const { return space; }
```

Басқаша айтқанда, `vector` класының `v` болып белгіленген `v.capacity()-v.size()` өрнегі `v` объектісіне `push_back()` функциясы арқылы қосымша жады бөлмей-ақ, жазуға болатын элементтер санын қайтарады.

19.2.3 resize функциясы

Бізде `reserve()` функциясы болғандықтан, `vector` класы үшін `resize()` функциясын жасау онша қиын болмайды. Бірнеше нұсқаларды қарастыру қажет:

- Жаңа көлем бұрынғы бөлінген жады мөлшерінен артық;
- Жаңа көлем бұрынғы бөлінген жады мөлшерінен артық, бірақ оның алдында бөлінген жады көлемінен кіші немесе соған тең;
- Жаңа көлем ескі көлеммен бірдей;
- Жаңа көлем ескі көлемнен төмен.

Енді не болғанын қарап шығайық:

```
void vector::resize(int newsize)
   // newsize элементі бар вектор құрамыз
   // әрбір элементті келісім бойынша 0.0 мәнімен инициалдаймыз
{
      reserve(newsize);
      for (int i=sz; i<newsize; ++i) elem[i] = 0;
   // жаңа элементтерді
   // инициалдау
      sz = newsize;
}
```

Жадымен негізгі жұмыс істеу `reserve()` функциясына жүктелген. Цикл жаңа элементтерді инициалдайды (егер олар бар болса).

Біз бұл нұсқалардың әрқайсысын тікелей бөлген жоқпыз, дегенмен, олардың дұрыс өңделетінін тексеру қиын емес.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Осы `reserve()` функциясының дұрыс жұмыс істейтініне сенімді болу үшін қандай нұсқаларды қарастыру (және тесттен өткізу) керек? `newsize == 0` шарты туралы не айта аласыз? `newsize == -77` шарты жайлы ше?

19.2.4 push_back функциясы

Алғаш қарағанда, **push_back()** функциясы жасауға қиын болып көрінуі мүмкін, бірақ reserve() функциясы барлығын да жеңілдетеді:

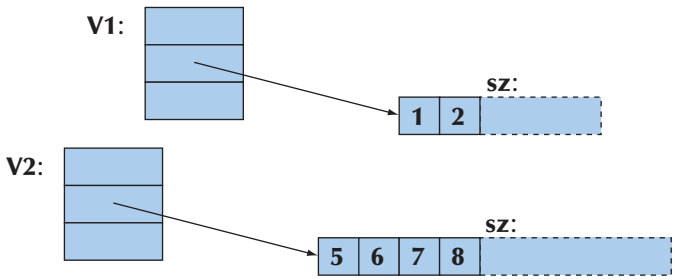
```
void vector::push_back(double d)
// вектор көлемін бірге арттырады;
// жаңа элементті d санымен инициалдайды
 {
   if (space==0) reserve(8);  // 8 элементке жады бөледі
   elseif(sz==space)reserve(2*space); //қосымшажадыбөледі
   elem[sz] = d; // вектор соңына d-ны қосады
   ++sz;          //көлемді ұлкейтеді (sz – элементтер саны)
 }
```

Басқаша айтқанда, егер бізде бос жады аймағы болмаса, онда бөлінген жады көлемін екі есе арттырамыз. Іс жүзінде бұл стратегия өте тиімді болып табылады, сондықтан ол кітапханадағы стандартты **vector** класында қолданылады.

19.2.5 Меншіктеу

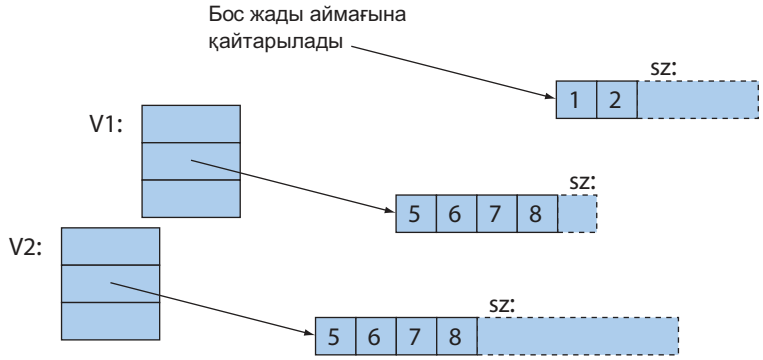
Векторларды меншіктеуді бірнеше тәсілмен анықтауға болады. Мысалы, векторлардың ені бірдей болса, біз меншіктеуді қабылдар едік. Бірақ, біз 18.2.2 бөлімде бұл жалпы сипаттағы әрекет деп және оның мағынасы да түсінікті болады деп шештік:

v1=v2 меншіктеуінен кейін **v1** векторы **v2** векторының көшірмесі болып шығады. Келесі суретті қарастырайық:



Әрине, біз элементтерді көшіріп алуымыз керек, бірақ бізде бос жады аймағы бар ма? Біз векторды ең соңғы элементтен кейінгі аймаққа көшіре аламыз ба? Көшіре алмаймыз: **vector** класының жаңа объектісі элементтер көшірмесін сақтайды, бірақ біз әлі оның қалай пайдаланатынын білмегендіктен, **вектор** соңында бос жады қалдыруды қарастырмадық. Қарапайым жүзеге асыру тәсілі төменде келтірілген:

- Көшірме үшін жады бөлеміз;
- Элементтерді көшіреміз;
- Бұрынғы жады аймағын босатамыз;
- **sz**, **elem** және **space** мүшелеріне жаңа мәндер меншіктейміз.



Сонда код төмендегідей түрде болады:

```
vector& vector::operator=(const vector& a)
   // көшіру конструкторына ұқсас, бірақ біз ескі
   // элементтермен жұмыс істеуге тиіспіз
{
    double* p = new double[a.sz]; // жаңа жады бөлеміз
    for(int i=0;i<a.sz;++i) p[i]=a.elem[i];
                                  // элементтерді көшіру
    delete[] elem;                // ескі жады аймағын босату
    space = sz = a.sz;            // жаңа көлем орнату
    elem = p;                     // жаңа элементтерді орнату
    return *this;                 // сілтемені өзіне қайтару
}
```

Жалпы қабылданған келісім бойынша, меншіктеу операторы мақсатты (меншіктелген) объектіге сілтеме қайтарады. *this өрнегінің мағынасы 17.10 бөлімінде айтылған болатын. Оны жүзеге асыру дұрыс орындалады, бірақ аздап ойланып, мұнда жады бөлу мен оны босату операцияларының артық екеніне көз жеткізуге болады. Егер нәтижелік вектордың элементтер саны меншіктелетін вектор элементтері санынан артық болса, не істер едік? Ал егер нәтижелік вектордың элементтер саны меншіктелетін вектор элементтері санымен бірдей болса, не істейміз? Көптеген қолданбалы программаларда соңғы жағдай жиі кездеседі. Кез келген жағдайда, біз элементтерді алдын ала нәтижелік векторға бөлінген компьютер жадына көшіріп жаза аламыз.

```
vector&  vector::operator=(const  vector& a)
{
    if (this==&a) return *this;  // өзін-өзі меншіктеу,
                                 // ешнәрсе жасау керек емес
    if (a.sz<=space) {
    // жады көлемі жетерлік деңгейде, жаңа жады керек емес
       for (int i = 0; i<a.sz; ++i) elem[i] = a.elem[i];
       // элементтерді көшіру
       sz = a.sz;
       return *this;
    }
    double* p = new double[a.sz];     // жаңа жады бөлеміз
    for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];
    // элементтерді көшіреміз
    delete[ ] elem;          // ескі жады аймағын босатамыз
    space = sz = a.sz;       // жаңа көлемді  орнатамыз
    elem = p;        // нұсқауышты жаңа элементке орнатамыз
    return *this;  // сілтемені мақсатты объектіге қайтару
 }
```

Кодтың бұл фрагментінде біз алдымен өзін-өзіне меншіктеуді (мысалы, `v=v;`) тексереміз, мұнда ешнәрсе істеу керек емес. Логикалық тұрғыдан алғанда бұл тексеру артық болып табылады, бірақ кейде ол программаны әжептеуір ықшам етуге мүмкіндік береді. Бұл тексеру **a** аргументінің функция-мүше шақырылатын объектінің өзі (яғни `operator=( )`) екендігін тексеруге мүмкіндік беретін `this` нұсқауышын қолдануды көрсетеді. Бұл кодтың одан `this ==&a` нұсқауын алып тастағанда да нақты түрде жұмыс істейтініне көз жеткізіңіз. `a.sz<=space` нұсқауы да кодты ықшамдау мақсатында қосылған. Бұл кодтың да одан `a.sz<=space` нұсқауын алып тастағанда да, жұмыс істей беретініне көз жеткізіңіз.

19.2.6 vector класының алдыңғы нұсқасы

Сонымен, біз `double` нақты түрде жұмыс істейтін `vector` класын алдық:

```
// double типіндегі сандар үшін шынайы вектор деуге болады
class vector {
/*
     инвариант:
        for 0<=n<sz elem[n]  n-элемент болып табылады
        sz<=space;
        егер sz<space болса, онда elem[sz-1] өрнегінен кейін
        double типіндегі  (space- sz) сандар үшін
*/
```

```
        int sz;             // көлемі
        double* elem;       // элементтерге (немесе 0-ге) нұсқауыш
        int space;          // элементтер саны плюс бос
        слоттар саны
public:
        vector() : sz(0), elem(0), space(0) { }
        vector(int s) :sz(s), elem(new double[s]), space(s)
        {
                for (int i=0; i<sz; ++i) elem[i]=0;
                // элементтер  инициалданған
        }

        vector(const vector&);          // көшіретін конструктор
        vector& operator=(const vector&); // көшіретін меншіктеу

        ~vector() { delete[] elem; }      // деструктор

        double& operator[ ](int n) { return elem[n]; }
        // қатынас құру
        const double& operator[](int n) const { return elem[n]; }

        int size() const { return sz; }
        int capacity() const { return space; }

        void resize(int newsize);         // арттыру
        void push_back(double d);
        void reserve(int newalloc);
    };
```

Бұл класта барлық негізгі операциялар бар екендігіне назар аударыңыз (18.3 бөлімді қ.): конструктор, келісім бойынша конструктор, көшіретін конструктор, деструктор. Оның құрамында тағы да мәліметтермен қатынасу (оларды индекстер [ ] арқылы пайдалану) операциялары, сол мәліметтер туралы ақпарат алу (**size()** және **capacity()**), сондай-ақ вектордың өсуін басқару жайлы операциялар да бар (**resize()**, **push_back()** және **reserve()**).

19.3 Шаблондар

Дегенмен бізге **double** типіндегі сандардан тұратын **вектор** жеткіліксіз болып табылады; біз өз векторларымыздың элементтері типін еркін түрде бергіміз келеді. Мысалы:

```
vector<double>
vector<int>
vector<Month>
vector<Window*>
// Windows класындағы объектілерге нұсқауыштар векторы
vector< vector<Record> >
// Record класы объектілері векторларының векторы
vector<char>
```

Бұл үшін біз шаблондарды анықтауды үйренуіміз керек. Негізінде біз басынан бастап, шаблондарды пайдаланған болатынбыз, бірақ бұған дейін біз оларды өзіміз анықтамаған едік. Стандартты кітапханада барлық керекті нәрселер бар, бірақ біз дайын рецептілерді қолданумен шектелмеуіміз керек, сондықтан стандартты кітапхананың, мысалы, `vector` класы мен `sort()` функциясының, қалай жобаланғанын және жүзеге асырылғанын қарастырып шығу қажет (21.1 және B.5.4 бөлімдер). Бұл тек теориялық қызығушылық емес, себебі көбінесе стандартты кітапхананы жасау кезінде қолданылатын құралдар мен тәсілдер өз программаларымызды құру кезінде көмектесіп жатады. Мысалы, 21 және 22 тарауларда біз шаблондар көмегімен стандартты контейнерлер мен алгоритмдерді қалай жүзеге асыруға болатынын көрсетеміз. Ал 24-тарауда ғылыми есептеулер үшін матрица кластарын қалай жасауға болатыны көрсетіледі.

Негізінде, *шаблон* (template) – бұл программалаушыға типтерді класс немесе функция параметрлері ретінде пайдалануға мүмкіндік беретін механизм. Осындай аргументтерді алғаннан кейін компилятор нақты класты немесе функцияны туындатады (генерациялайды).

19.3.1 Типтер шаблондар параметрі ретінде

Сонымен, біз элементтер типі `vector` класының параметрі болғанын қалаймыз. `vector` класын аламыз да, `double` түйінді сөзін `T` әрпіне алмастырамыз, мұндағы `T` – `double`, `int`, `string`,`vector<Record>` және `Window*` сөздерінің бірін мән ретінде қабылдайтын параметр. C++ тілінде типті беретін `T` параметрін сипаттау үшін "барлық `T` типтері үшін" дегенді білдіретін `template<class T>` префиксі қолданылады. Мысал қарастырайық:

```
// T типіндегі элементтер үшін шынайы вектор деуге болады
template<class T> class vector {
// "барлық T типтері үшін" деп оқылады
// (математикадағы сияқты деуге болады)
     int sz;                  // элементтер саны көлемі)
     T* elem;                 // элементтерге нұсқауыш
     int space;               // көлемі + бос жады аймағы
```

```
public:
     vector() : sz(0), elem(0), space(0) { }
     vector(int s);

     vector(const vector&);
     // көшіретін конструктор
     vector& operator=(const vector&);
     // көшіретін меншіктеу

     ~vector() { delete[] elem; }     // деструктор

     T& operator[](int n) { return elem[n]; }
     // қол жеткізу: сілтемені қайтарады
     const T& operator[](int n) const { return elem[n]; }

     int size() const { return sz; }  // ағымдағы көлем
     int capacity() const { return space; }

     void resize(int newsize);    // векторды ұлғайтады
     void push_back(const T& d);
     void reserve(int newalloc);
};
```

Бұл **vector** класының анықтамасы **double** типіндегі элементтері бар **vector** класының анықтамасымен (19.2.6 бөлімді қ.) дәл келеді, жалғыз айырмашылығы бұрынғы **double** түйінді сөзі енді **T** шаблондық параметрімен алмастырылған. Бұл **vector** шаблондық класын келесідей түрде пайдалануға болады:

```
vector<double> vd;          // T – double типінде
vector<int> vi;             // T – int типінде
vector<double*> vpd;        // T – double* типінде
vector< vector<int> > vvi; // T vector<int> типінде,
                            // мұндағы T – int типінде
```

Компьютер шаблондық параметр орнына қоятын белгілі бір типте класс (шаблондық аргументке сәйкес) туындатады деп есептеуге болады. Мысалы, компилятор программада **vector<char>** конструкциясын көрсе, ол шамамен төмендегідей кодты генерациялайды (жасап береді):

```
class vector_char {
   int sz;          // элементтер саны (көлемі)
   char* elem;      // элементтерге нұсқауыш
   int space;       // көлемі + бос жады аймағы
```

```
    public:
        vector_char();
        vector_char(int s);

        vector_char(const vector_char &);
                                    // көшіретін конструктор
        vector_char& operator=(const vector_char &);
                                    // көшіретін меншіктеу

        ~vector_char ();            // деструктор

        char& operator[] (int n);   // қатынас құру:
                                       сілтеме қайтарады
        const char& operator[] (int n) const;

        int size() const;           // ағымдағы көлем
        int capacity() const;

        void resize(int newsize);   // ұлкейту
        void push_back(const char& d);
        void reserve(int newalloc);
};
```

Мағынасы бойынша `vector<double>` конструкциясына сәйкес келетін ішкі атты пайдалана отырып, компилятор `vector<double>` класы үшін `double` типіндегі элементтері бар `vector` класы аналогын генерациялайды (19.2.6 бөлімді қ.)

Кейде шаблондық класты *типті туындатушы* (type generator) деп те атайды.

Берілген шаблондық аргументтер бойынша шаблондық класс көмегімен типтерді құру (генерациялау) процесі *шаблонның мамандануы (*specialization) немесе *нақтылануы* (instantiation) деп аталады. Мысалы, `vector<char>` және `vector<Poly_line*>` кластары `vector` класының мамандандырулары деп аталады. Қарапайым жағдайларда, мысалы, `vector` класымен жұмыс істеу кезінде нақтылану қиындық тудырмайды. Одан гөрі жалпылама және қиындау жағдайларда шаблонның нақтылануы өте күрделеніп кетеді. Қуанышқа орай, шаблонды қолданушылар үшін осындай күрделіліктердің барлығы да компиляторларды жасаушыларға жүктеледі.

Шаблонның нақтылануы программаның орындалу кезінде емес, оны компиляциядан өткізу кезінде немесе байланыстарды редакциялау кезеңінде орындалады.

Әрине, шаблондық кластың функция-мүшелері болуы мүмкін. Мысал қарастырайық:

```
void fct(vector<string>& v)
{
    int n = v.size();
    v.push_back("Norah");
    // . . .
}
```

Шаблондық кластың осындай функция-мүшесін қолдану кезінде компилятор соған сәйкес белгілі бір функцияны құрады. Мысалы, компилятор мынадай түрде берілген `v.push_back("Norah")` шақыруды кездестірсе, ол келесі функцияны туындатады:

```
void vector<string>::push_back(const string& d) { /* ... */ }
```

ол үшін мына шаблондық анықтау пайдаланылған:

```
template<class T> void vector<T>::push_back(const T& d) { /* ... */ };
```

Сонымен, мынадай шақыруды `v.push_back("Norah")` қолдану кезінде оған белгілі бір функция сәйкес келеді. Басқаша айтқанда, егер сізге белгілі бір типтегі аргументі бар функция керек болса, компилятор сіздің шаблоныңызды негізге ала отырып, оны өзі жазып шығады.

`template<class T>` префиксі орнына сіз `template<typename T>` префиксін пайдалана аласыз. Бұл екі конструкция да бірдей, бірақ кейбір программалаушылар `typename` түйінді сөзі қолданғанды ұнатады, "өйткені ол түсініктірек және ешкім де мұнда шаблондық аргумент ретінде ішкі құрамдас типтерді, мысалы, `int` типін, пайдалануға болмайды деп ойламайды". Біз `class` түйінді сөзінің өзі "тип" дегенді білдіреді деп санаймыз, сондықтан мұндай конструкциялардың арасында ешқандай да айырмашылық жоқ. Оның үстіне `class` сөзі қысқарақ жазылады.

19.3.2 Жалпыланған программалау

Шаблондар – C++ тілінде жалпыланған, яғни, әмбебап программалаудың негізі. Негізінде, C++ тіліндегі жалпыланған программалаудың қарапайым анықтамасы – ол "шаблондар арқылы программалау". Әрине, мұндай анықтама өте қарапайым түрде берілген. Программалаудың іргелі ұғымдардың анықтамасын программалау тілінің конструкциясы терминдерімен беру қажет емес. Мұндай конструкциялар программалау технологияларын сүйемелдеу үшін керек, керісінше емес. Кең таралған түсініктердің көпшілігі сияқты жалпыланған (яғни әмбебап) программалаудың бірнеше анықтамалары бар. Біз солардың ішіндегі ең қарапайымын ең дұрысы деп есептейміз:

Жалпыланған программалау – аргументтер түрінде берілген әртүрлі типтермен жұмыс істейтін кодтарды жасау, оның үстіне бұл типтер спецификалық түрдегі синтаксистік және семантикалық талаптарға сәйкес болу тиіс.

Мысалы, **вектор** элементтерінің типі көшіріп алуға болатын (көшіру конструкторы және көшіретін меншіктеу арқылы) тип түрінде болуы тиіс. 20-21 тарауларда аргументтері арифметкалық операциялар болып келетін шаблондар көрсетіледі. Біз класты параметрлер арқылы жазу (параметрлеу) кезінде *шаблондық класс* аламыз, оны көбінесе *параметрленген тип* (parameterized type) немесе *параметрленген класс* (parameterized class) деп те атайды. Біз функцияны параметрлеу кезінде *шаблондық функция* (function template) аламыз, оны да көбінесе *параметрленген функция* (parameterized function), ал кейде *алгоритм* (algorithm) деп атайды. Сол себепті жалпыланған программалауды кейде *алгоритмге бағытталған программалау* (algorithm-oriented programming) деп атайды. Мұндайда жобалау кезіндегі негізгі назарымыз қолданылатын типтерге емес, алгоритмдерге ауысып кетеді.

Программалауда параметрленген типтер түсінігі маңызды рөл атқаратындықтан, бұдан былай осы шатасқан терминологияны түсінуге тырысып көрейік. Бұл бұрыннан таныс терминдерді басқа мәтіндер ішінде кездестіргенде, түсініксіздіктерді болдырмауға мүмкіндік береді.

Тікелей шаблондық параметрлерге негізделген жалпыланған программалаудың осы формасын көбінесе *параметрлік полиморфизм* (parametric polymorphism) деп атайды. Бұған қарама-қарсы кластар иерархиялары (сатылары) мен виртуалды функцияларға байланысты туындайтын полиморфизмді *арнайы полиморфизм* (ad hoc polymorphism) деп атайды да, соған сәйкес стильді – *объектіге бағытталған программалау* деп айтады (14.3-14.4 бөлімдерді қ.). Программалаудың бұл екі стилін де полиморфизм (polymor-phism) деп атаудың себебі, бұлардың әрқайсысы программалаушыға бірыңғай интерфейс арқылы бір түсініктің көптеген нұсқаларын жасауға мүмкіндік беретінінде болып отыр. Полиморфизм грек тілінен аударғанда "көп форма" дегенді білдіреді. Сонымен, сіздің жалпы интерфейс арқылы әртүрлі типтермен жұмыс істеу мүмкіндігіңіз бар. 16-19-тарауларда қарастырылған **Shape** класына арналған мысалдарда осы **Shape** класы арқылы анықталған интерфейс көмегімен біз әртүрлі формалармен (**Text**, **Circle** және **Polygon** кластарымен) жұмыс істедік. **vector** класын пайдалана отырып, **vector** шаблондық класымен анықталған интерфейс арқылы біз көптеген векторлармен (мысалы, **vector<int>**, **vector<double>** және **vector<Shape*>**) нақты түрде жұмыс істейміз.

Объектіге бағытталған программалау (кластар иерархиялары мен виртуалдық функциялар көмегімен) мен жалпыланған программалау (шаблондар көмегімен) арасында бірнеше айырмашылықтар бар. Ең көзге түсері жалпыланған программалауда шақырылатын функцияны таңдауды компиляциядан өткізу кезінде компилятор анықтайды, ал объектіге бағытталған программалауда ол программаны орындау кезінде анықталады. Мысал қарастырайық:

```
v.push_back(x);          // x-ті v векторына жазу
s.draw();                // s фигурасын салу
```

`v.push_back(x)` функциясын шақыру үшін компилятор `v` объектісіндегі элементтер типін анықтайды да, соған сәйкес `push_back()` функциясын пайдаланады. Ал `s.draw()` функциясын шақыру үшін ол жанамалы түрде `draw()` функциясын шақырады (`s` объектісімен байланысқан виртуалды функциялар кестесі арқылы, 14.3.1 бөлімді қ.). Бұл объектіге бағытталған программалауға жалпыланған программалауда жоқ еркіндік береді және бұл кәдімгі жалпыланған программалауға бірсыпыра жүйелілік қасиетін беріп, оны түсінікті және тиімді етеді ("арнайы" және "параметрлік" деге анықтауларына байланысты).

Енді қорытындылайық:

- *Жалпыланған программалау* компиляция кезеңінде қабылданған шешімдерге негізделіп жасалған шаблондармен сүйемелденеді.
- *Объектіге бағытталған программалау* программаны орындау кезеңінде қабылданған шешімдерге негізделіп жасалған кластар иерархиясымен және виртуалдық функциялармен сүйемелденеді.

Осы стильдердің қатарласа қолданылуы әбден мүмкін және олар жұмыста пайдалы болып табылады. Мысал қарастырайық:

```
void draw_all(vector<Shape*>& v)
  {
        for (int i=0; i<v.size(); ++i) v[i]->draw();
  }
```

Мұнда біз басқа бір виртуалды функция көмегімен базалық кластағы (`Shape`) виртуалды функцияны (`draw()`) шақырамыз, бұл объектіге бағытталған программалаудың анықтамасынан шығады. Бірақ `Shape*` нұсқауыштары параметрленген тип болып саналатын `vector` класының объектісінде сақталады, міне, мұнда біз қатарластыра (қарапайым) жалпыланған программалауды да қолданып отырмыз.

Енді фиолософияны азайтайық. Негізінде шаблондар не үшін пайдаланылады? Икемді әрі жоғары өнімділікпен жұмыс істейтін программалар алу үшін.

- Программаның жұмыс өнімділігі маңызды рөл атқаратын кездерде шаблондарды қолданыңыз (мысалы, нақты уақыт кезеңінде көптеген есептеулер орындау үшін, бұл туралы толығырақ 24 және 25-тарауларда айтылады).
- Әртүрлі типтерден келіп түсетін мәліметтердің араласу икемділігі маңызды болатын сәттерде шаблондарды қолданыңыз (мысалы, C++ тілінің стандартты кітапханасымен жұмыс істеген кездерде; бұл тақырып 20-21 тарауларда қарастырылады).

Шаблондардың жоғары икемділігі және ыңғайлы жұмыс өнімділігі сияқты көптеген пайдалы қасиеттері бар, бірақ өкінішке орай, олар кемшіліксіз емес. Әр заттың артықшылықтарына қарай кемшіліктері де болып отырады. Шаблондардың негізгі кемшілігі олардың икемділігі мен жоғары жұмыс өнімділігіне қол жеткізу шаблонның "ішкі жағы" (оның анықталуы) мен оның интерфейсінің (жариялануы) дұрыс ажыратылмауы арқылы жүзеге асырылады. Бұл қателердің дұрыс анықталмауына әкеліп соғады, әсіресе, қателер жайлы берілетін мәлімдемелер сапасы өте төмен болады. Кейде осы қателер жайлы мәлімдемелер компиляциядан өткізу кезеңінде өз уақытынан өте кеш шығарылып жатады.

Шаблондарды қолданатын программаларды компиляциядан өткізу кезінде, компилятор шаблондар ішін және оның шаблондық аргументтерін "қадағалап" отырады. Мұны ол ең ықшам (оптималды) кодтар жазуға қажетті ақпарат алу үшін жасайды. Бұл ақпаратқа қол жеткізу үшін қазіргі компиляторлар шаблонның, ол қолданылатын барлық жерде, толық анықталғанын талап етеді. Осы жайт оның функция-мүшелеріне де және одан шақырылатын барлық шаблондық функцияларға қатысты болып саналады. Нәтижесінде шаблон авторлары олардың анықталуын тақырыптық файлдарға орналастыруға талпынады. Негізінде стандарт мұны талап етпейді, бірақ тілдің тиімді нұсқалары жасалып шыққанша, өз шаблондарыңызбен осылай жұмыс істегендеріңіз дұрыс: трансляциядан өтетін бірнеше программалық бірліктерде қолданылатын барлық шаблондарды тақырыптық файлға орналастырыңыздар.

Біз сіздерге өте қарапайым шаблондардан бастап, біртіндеп тәжірибе жинақтауды ұсынамыз. Жобалаудың пайдалы бір тәсілін `vector` класының мысалы арқылы көрсеткен болатынбыз: алдымен бір класс жасап алыңыз да, нақты типтерді қолдана отырып, оны тесттен өткізіңіз. Егер программа жұмыс істейтін болса, онда қолданылатын типтерді шаблондық параметрлермен алмастырыңыз. Программаның жалпылығын, типтік қауіпсіздігін және жоғары өнімділігін қамтамасыз ету үшін шаблондар кітаханасын, мысалы, C++ тілі стандартты кітапханасын пайдаланыңыз. 20-21 тараулар стандартты кітапханадағы контейнерлер мен алгоритмдерге арналған. Оларда шаблондарды пайдаланудың көптеген мысалдары келтірілген.

19.3.3 Контейнерлер және мұралау

Бұл объектіге бағытталған программалау мен жалпыланған программалаудың араласуының бір түрі, оны адамдар тұрақты түрде, бірақ нәтиже ала алмай, қолдануға тырысып келеді: туынды кластың объектілері контейнерлерін базалық кластың объектілері контейнері ретінде пайдалану. Мысал қарастырайық:

```
vector<Shape> vs;
vector<Circle> vc;
```

```
vs = vc;          // қате: vector<Shape> класы қажет
void f(vector<Shape>&);
f(vc);            // қате: vector<Shape> класы қажет
```

Бірақ неге? "Бұлардан кейін **Circle** класын **Shape** класына түрлендіре аламын!", – деп айта аласыз ба! Жоқ, айта алмайсыз. Сіз **Circle*** нұсқауышын **Shape*** нұсқауышына, **Circle&** сілтемесін **Shape&** сілтемесіне түрлендіре аласыз, бірақ біз әдейілеп **Shape** объектілерін меншіктеуге тиым салғанбыз, сондықтан сіз белгілі бір радиусы бар **Circle** класын радиусы жоқ **Shape** типті айнымалыға орналастырсаңыз не болатынын сұрауға да құқығыңыз болмайды (14.2.4-тарауын қ.). Егер бұл іске асатын болса, яғни біз мұндай меншіктеу жасауға рұқсат беретін болсақ, онда бүтін сандарды қысқартуға ұқсаған "қию" ("slicing") деп аталатын әрекет туындайды (3.9.2 бөлімді қ.).

Сонымен, қайтадан нұсқауыштарды пайдаланып көрелік.

```
vector<Shape*> vps;
vector<  Circle*> vpc;
vps = vpc;          // қате: vector<Shape*> класы қажет
void f(vector<Shape*>&);
f(vpc);             // қате: vector<Shape*> класы қажет
```

Қайтадан типтер жүйесі қарсылық білдіріп тұр; неге? **f()** функциясының не істей алатынын қарастырайық.

```
void  f(vector<Shape*>&  v)
{
   v.push_back(newRectangle(Point(0,0),Point(100,100)));
}
```

Әрине, біз **Rectangle*** нұсқауышын **vector<Shape*>** класының объектісіне жаза аламыз. Алайда, осы **vector<Shape*>** класының объектісі программаның бір жерінде **vector<Circle*>** класының объектісі ретінде қарастырылса, онда келеңсіз тосынсый туындауы мүмкін. Негізінде, егер компилятор жоғарыдағы мысалды өткізіп жіберетін болса, онда **Rectangle*** нұсқауышы **vpc** векторында не істер еді? Мұралау – қуатты әрі нәзік механизм, ал шаблондар оның мүмкіндіктерін жанамалы түрде кеңейтпейді. Мұралауды өрнектеу үшін шаблондарды пайдалану тәсілдері бар, бірақ ол тақырып бұл кітап ауқымынан тысқары жатыр. Тек кез келген C шаблондық класы үшін "**D** – бұл **B**" өрнегінің "**C<D>** – бұл **C<B>**" өрнегі емес екенін есте сақтаңыз. Біз мұны типтерді бұзудың әдейі жасалмағанына қарсы қорғаныс ретінде бағалай білуіміз керек. 25.4.4 бөлімін де қараңыз.

19.3.4 Бүтін типтер шаблондық параметрлер ретінде

Әрине типтер арқылы кластарды параметрлеу пайдалы болып табылады. Ал, мысалы, бүтін сандар немесе тіркестер арқылы кластарды параметрлеу жайында не айтуға болар екен? Негізінде, аргументтердің кез келген түрі пайдалы болуы мүмкін, бірақ біз тек типтерді және бүтін сан түріндегі параметрлерді қарастырамыз. Параметрлердің басқа түрлерінің пайдалы болып саналуы сирегірек орын алады да, C++ тілінің параметрлердің басқа түрлерін сүйемелдеуі күрделі сипатта болады және де ол кең әрі терең білімді керек етеді.

Біз шаблондық аргумент ретінде бүтін сандық мәнді пайдаланудың кең таралған түрінен: элементтерінің саны компиляциядан өткізу кезеңінде белгілі болатын контейнерден мысал келтірейік:

```
template<class T, int N> struct array {
    Telem[N]; //элементтерін кластың жиым-мүшесінде сақтайды

    // келісім бойынша конструкторларды,
    деструкторды және меншіктеуді пайдаланады

    T& operator[] (int n);
    // қол жеткізу (пайдалану): сілтемені қайтарады
    const T& operator[] (int n) const;

    T* data() { return elem; }  // T* типіне түрлендіру
    const T* data() const { return elem; }

    int size() const { return N; }
};
```

Біз `array` класын (20.7 бөлімді де қ.) шамамен былай қолдана аламыз:

```
array<int,256> gb;                  // 256 бүтін сан
array<double,6> ad = {0.0,1.1,2.2,3.3,4.4,5.5};
// инициализатор!
const int max = 1024;

void some_fct(int n)
{
    array<char,max> loc;
    array<char,n> oops;
    // қате: n мәні компиляторға белгісіз
    // . . .
    array<char,max> loc2 = loc; // резерв көшірме жасау
    // . . .
    loc = loc2;             // қалпына келтіру
    // . . .
}
```

Array класының өте қарапайым екені белгілі, ол – **vector** класына қарағанда қарапайымырақ және қуаты әлсіздеу, бірақ неге кейде **vector** класын емес осы **array** класын пайдалану керек? Оның бір жауабы – "тиімділігі". **Array** класы объектісінің көлемі компиляциядан өту кезеңінде белгілі болады, сондықтан оған компилятор бос жады аймағынан емес, статикалық жады (**gb** тәрізді ауқымды объектілер үшін) аймағынан немесе стектегі жадыдан (**loc** сияқты жергілікті объектілер үшін) орын бөле алады. Берілген диапазон шегінен шығып кеткендігін тексере отырып, біз константаларды (мысалы, **N** параметрі мөлшерін) салыстырамыз. Көптеген программалар үшін мұның тиімділігі онша еленерліктей емес, бірақ жүйенің маңыздырақ компонентін, мысалы, желі драйверін жазуымыз керек болса, онда азырақ айырманың өзі әжептеуір болып шығады. Бұдан да маңыздырағы кейбір программалар бос жады аймағын пайдалана алмайды. Мұндай программалар құрамдас жүйелермен және/немесе негізгі критериі қауіпсіздік болып табылатын программалармен жұмыс істейді (ол туралы 25-тарауда айтылады). Ондай программаларда **array** жиымының **vector** класына қарағанда негізгі шектеуді (бос жады аймағын пайдалануға тыйым салуды) бұзбайтын бірсыпыра артықшылықтары бар.

Бұған қарама-қарсы сұрақ қояйық: "Неге жай ғана құрамдас жиымдарды пайдаланбаймыз" емес, "Неге жай ғана **vector** класын пайдаланбаймыз?". 18.5 бөлімде көрсетілгендей, жиымдар қателер туындатуы мүмкін: олар өз көлемін білмейді, олар аз ғана мүмкіндік болғанның өзінде нұсқауыштарды түрлендіреді және дұрыс көшірілмейді де; ал **array** класында мұндай жағдайлар болмайды. Мысал қарастырайық:

```
double* p = ad;
// қате: нұсқауышқа жанамалы түрде түрлендіру жоқ
double* q = ad.data();     // OK: тікелей түрлендіру

template<class C> void printout(const C& c)
{
    for (int i = 0; i<c.size(); ++i) cout << c[i] <<'\n';
}
```

Бұл **printout()** функциясын **array** класынан да және **vector** класынан да шақыруға болады:

```
printout(ad);              // array класынан шақыру
vector<int> vi;
// . . .
printout(vi);              // vector класынан шақыру
```

Бұл – мәліметтерге қалай қол жеткізуге болатынын көрсететін жалпылама программалаудың қарапайым мысалы. Ол **array** класы үшін де және **vector** класы үшін де бір интерфейстің (**size()** функциясы және индекстеу операциясы) қолданылуы арқасында жұмыс істейді. Бұл стиль толығырақ 20 және 21-тарауларда қарастырылады.

19.3.5 Шаблондық аргументтерді шығару

Шаблондық класс негізінде нақты класс объектілерін жасай отырып, біз шаблондық аргументтерді көрсетеміз. Мысал қарастырайық:

```
array<char,1024> buf;
// buf жиымы үшін T параметрі – char, ал N == 1024
array<double,10> b2;
// b2 жиымы үшін T параметрі – double, ал N == 10
```

Әдетте, шаблондық функциялар үшін компилятор функция аргументтерінен шаблондық аргументтер шығарады. Мысал келтірейік:

```
template<class T, int N> void fill(array<T,N>& b, const T& val)
{
    for (int i = 0; i<N; ++i) b[i] = val;
}

void f()
{
    fill(buf, 'x');
    // fill() функциясы үшін T параметрі – char, ал N == 1024
    // өйткені аргумент buf объектісі болып табылады
    fill(b2,0.0);
    // fill() функциясы үшін T – double, ал N == 10
    // өйткені аргумент b2 объектісі болып табылады
}
```

Техникалық тұрғыдан қарағанда, `fill(buf,'x')` `fill<char,1024>(buf,'x')` жазбасының қысқаша формасы болып табылады, ал `fill(b2,0)` – `fill<double,10>(b2,0)` шақыруының қысқа түрі, бірақ қуанышқа орай біз әрқашанда нақты бола беруіміз қажет емес. Компилятордың өзі біз үшін осы ақпаратты шығарып алады.

19.3.6 vector класын жалпылау

Біз "`double` типіндегі элементтердің `vector`" класы негізінде жалпылама класс құрғанда және "`T` типіндегі элементтердің `vector`" шаблонын шығарғанда, біз `push_back()`, `resize()`, және `reserve()` функцияларының анықтамаларын тексермеген болатынбыз. Енді біз оны істеуге тиістіміз, өйткені 19.2.2 және 19.2.3 бөлімдерде бұл функциялар `double` типі үшін дұрыс болып табылатын айқындаулар негізінде анықталған еді, бірақ олар біз вектор элементтерінің төмен-

дегідей типтері ретінде пайдаланғымыз келген барлық типтер үшін орындалмайды:

- Егер **x** типінің келісім бойынша берілген мәні болмаса, **vector<X>** класын қалай программалауға болады?
- Онымен жұмыс істеп болған соң, вектор элементтерінің жойылатынына қалай кепілдік беруге алады?

Жалпы бұл проблемаларды біз шешуге тиіспіз бе? Біз "Келісім бойынша мәні жоқ типтер үшін **вектор** құрмаңдар" немесе "Деструкторлары проблема туғызатын типтер үшін векторларды пайдаланбаңдар" деп айта алар едік қой. Ортақ пайдаланылатын конструкция үшін мұндай шектеулер онша жағымды емес және олар программалаушы есеп мағынасын түсінбеген немесе қолданушылар туралы ойламаған деген ой туғызады. Көбінесе мұндай күмәнді жағдайлар дұрыс болып жатады, бірақ стандартты кітапханаларды жасаушылар бұл санатқа жатпайды. Стандартты **vector** класын қайталау үшін жоғарыдағы екі проблемадан құтылу керек.

Біз келісім бойынша мәні жоқ типтермен жұмыс істей аламыз, ол үшін қолданушыда олардың мәнін өздігінен беру мүмкіндігі болуы тиіс:

```
template<class T> void vector<T>::resize(int newsize, T def=T());
```

Егер қолданушы басқаша көрсетпесе, **T()** конструкторы арқылы жасалған объектіні келісім бойынша берілген мән ретінде пайдаланыңыз. Мысал қарастырайық:

```
vector<double> v1;
v1.resize(100);
// double() объектісінің 100 көшірмесін қосамыз, яғни 0.0
v1.resize(200, 0.0);
// 0.0 санының 100 көшірмесін қосу — ескерту жасау артық
v1.resize(300, 1.0);  //1.0 санының 100 көшірмесін қосамыз
struct No_default {
    No_default(int);
    // No_default класының жалғыз конструкторы
// . . .
};

vector<No_default> v2(10);
// қате: 10 санын жасауға талпыныс No_default()
vector<No_default> v3;
v3.resize(100,No_default(2));
//No_default(2) объектілерінің 100 көшірмесін қосу
v3.resize(200);
//қате: 100 санын жасауға талпыныс No_default()s
```

Деструкторға байланысты проблемадан құтылу қиын. Негізінде, біз нақты қиын жағдайда қалдық: мәліметтер құрылымындағы мәліметтердің бір бөлігі инициалданған, ал кейбір бөліктері инициалданбаған. Бұған дейін біз осылардан туындайтын инициалданбаған мәліметтер мен қателерді болдырмауға тырыстық. Енді біз **vector** класының жасаушылары ретінде бұрын **vector** класын қолданушылар ретінде кездестірмеген проблемаға кез болдық.

Біріншіден, біз инициалданбаған жады аймағын алатын және онымен жұмыс істейтін тәсіл табуымыз керек. Қуанышқа орай, стандартты кітапханада инициалданбаған жады аймағын бөлетін **allocator** класы бар. Оның аздап қарапайым етілген нұсқасы төменде көрсетілген:

```
template<class T> class allocator {
public:
    llocate(int n);   // T типіндегі n объектіге жады бөлу
    void deallocate(T* p, int n);  // p адресінен бастап,
    // T типіндегі n объект орналасқан жады аймағын босату

    void construct(T* p, const T& v);
    // p адресінде v-ға тең мәні бар T типіндегі объект құру
    void destroy(T* p);    // p адресіндегі T объектісін жою
};
```

Егер сізге бұл мәселе бойынша толық ақпарат керек болса, *The C++ Programming Language* кітабын немесе C++ тілі стандартын (**<memory>** тақырыбы сипаттамасын қ.) және де B.1.1 бөлімін қараңыз. Дегенмен, біздің программада келесі көрсетілген әрекеттерді орындауға мүмкіндік беретін төрт іргелі операторлар қарастырылады:

- Инициалданбаған **T** типіндегі объектіні сақтауға жеткілікті жады аймағын бөлу;
- Инициалданбаған жады аймағында **T** типіндегі объект құру;
- **T** типіндегі объектіні жойып, жады аймағын инициалданбаған қалыпқа қайтару;
- Инициалданбаған **T** типіндегі объектіні сақтауға жеткілікті инициалданбаған жады аймағын босату.

allocator класының **vector<T>::reserve()** функциясын жүзеге асыру үшін қажетті нәрсе екені таңданарлық емес шығар. **vector** класына **allocator** класының параметрін қосудан бастайық:

```
template<class T, class A = allocator<T> > class vector {
```

```
        A alloc;
        // allocator класы объектісін элементтер үшін
        // бөлінген жадымен жұмыс істеу үшін пайдаланамыз
        // . . .
};
```

new операторы орнына қолданылатын жады аймағын бөлгіштен басқа **vector** класын сипаттаудың қалған бөлігі бұрынғымен бірдей. **vector** класын қолданушылар ретінде оның элементтері үшін стандартты емес түрде бөлінген жады аймағын **vector** класының басқарғанын өзіміз қаламасақ, жады аймағын бөлгішті есепке алмауымызға болады. vector класын жасаушылар және іргелі проблемаларды түсініп, программалаудың негізгі технологияларын игергісі келетін студенттер ретінде, біз вектордың инициалданбаған жадымен қалай жұмыс істейтінін түсініп, қолданушыларға дұрыс құрылған объектілер жасап беруіміз керек. Мұндағы өзгертілуге тиіс жалғыз код – жадымен тікелей жұмыс істейтін vector класының функция-мүшелері, мысалы, **vector<T>::reserve()** функциясы:

```
template<class T, class A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return;  // көлем азайтылмайды
    T*p=alloc.allocate(newalloc); // жаңа жады аймағын бөлу
    for(int i=0;i<sz;++i) alloc.construct(&p[i],elem[i]);
    // көшіру
    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i]); // жою
    alloc.deallocate(elem,space);
    // бұрынғы жады аймағын босату
    elem = p;
    space = newalloc;
}
```

Біз элементті, инициалданбаған жады аймағында оның көшірмесін жасай отырып, жадының жаңа бөлігіне жылжытамыз да, артынан оның түпнұсқасын (оригиналын) жоямыз. Мұнда меншіктеуді пайдалануға болмайды, өйткені **string** сияқты типтер үшін меншіктеу дегеніміз жадының мақсатты аймағы инициалданған дегенді білдіреді.

reserve(), **vector<T,A>::push_back()** функциялары болғандықтан, еш қиындықсыз келесі кодтарды жаза аламыз:

```
template<class T, class A>
void vector<T,A>::push_back(const T& val)
```

```
{
  if (space==0) reserve(8);
    // 8 элемент үшін бөлінген жадыдан бастаймыз
    else if (sz==space) reserve(2*space);
    // жадыны көбірек бөлеміз
    alloc.construct(&elem[sz],val);
    // val мәнін соңына қосамыз
    ++sz;  // көлемді ұлғайтамыз
}
```

Осы сияқты етіп, `vector<T,A>::resize()` функциясын да жаза аламыз:

```
template<class T, class A>
void vector<T,A>::resize(int newsize, T val = T())
{
    reserve(newsize);
    for (int i=sz; i<newsize; ++i) alloc.construct(&elem[i],val);
    // құрамыз
    for (int i = newsize; i<sz; ++i) alloc.destroy(&elem[i]);
    // жоямыз
    sz = newsize;
}
```

Кейбір типтердің келісім бойынша конструкторлары болмағандықтан, біз жаңа элементтер үшін бастапқы мәндерді тағайындау мүмкіндігін қайта бердік.

Тағы бір жаңалық – вектор кішірейтілген кездегі артық элементтерге арналған деструктор. Белгілі бір типтегі объектіні жадының қарапайым ұялары жиынтығына айналдыратын деструкторды көзге елестетіңізші.

"Жады бөлгіштерді мәжбүрлеусіз пайдалану" – бұл әжептеуір күрделі, әрі айлалы өнер. Сарапшы деңгейіне көтерілгеніңізді сезінгенше, бұларды көп пайдалануға тырыспаңыз.

19.4 Аралықтар мен аластамаларды тексеру

Біз өзіміз жасаған `vector` класының ағымдағы жағдайын талдап шығып, оның берілген диапазоннан шығып кетуін тексеру қарастырылмағанын байқадық (қорқынышпен). Енді `operator[]` операторын жүзеге асыру қиындық тудырмайды:

```
template<class T, class A> T& vector<T,A>::operator[](int n)
{
      return elem[n];
}
```

Келесі мысалды қарастырайық:

```
vector<int> v(100);
v[-200] = v[200]; // ой!
int i;
cin>>i;
v[i] = 999;     // жадының кез келген бір ұясына зақым келтіру
```

Бұл код біз жасаған **vector** класының объектісіне жатпайтын жады аймағын пайдалана отырып, компиляциядан өтеді және орындалады. Ол үлкен келеңсіздіктер туғызуы мүмкін! Нақты жасалған программада мұндай код болмайды. Бұл проблеманы шешу үшін **vector** класын жақсартуға тырысайық. Ең қарапайым тәсіл – класқа қатынас құруды (пайдалануды) тексеретін **at()** атты операцияны қосу болып табылады:

```
struct out_of_range { /* . . . */ };  // берілген диапазоннан
// шығып кетуге байланысты қателер жайлы мәлімдейтін класс

template<class T, class A = allocator<T> > class vector{
    // . . .
    T& at(int n);               // тексеру арқылы қатынасу
    const T& at(int n) const;   // тексеру арқылы қатынасу
    T& operator[](int n);       // тексерусіз қатынасу
    const T& operator[](int n) const; // тексерусіз қатынасу
    // . . .
};

template<class T, class A > T& vector<T,A>::at(int n)
{
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}

template<class T, class A>T& vector<T,A>::operator[](int n)
// бұрынғыша
{
    return elem[n];
}
```

Сонымен, біз келесі функцияны жаза аламыз:

```
void print_some(vector<int>& v)
{
    int i = -1;
    cin >> i;
    while(i!= -1) try {
        cout << "v[" << i << "]==" << v.at(i) << "\n";
    }
    catch(out_of_range) {
        cout << "дұрыс емес индекс: " << i << "\n";
    }
}
```

Мұнда біз берілген диапазон шегінен шығып кетуді тексеру арқылы қатынас құруды қамтамасыз ету үшін `at()` функциясын пайдаланып отырмыз және вектор элементін рұқсатсыз пайдалану байқалса, `out_of_range` аластамасын жасаймыз.

Негізгі идея, егер бізге индекстің дұрыс екені белгілі болса, `[]` индекстеу операциясын пайдалану, ал егер берілген диапазон шегінен шығып кету мүмкін болса, онда `at()` функциясын пайдалану болып табылады.

19.4.1 Ескерту: жобалау сұрақтары

Сонымен, бәрі жақсы, бірақ бізге неге берілген диапазон шегінен шығып кетуді тексеру ісін `operator[]()` функциясына қоспасқа? Дегенмен, жоғарыда көрсетілгендей, `vector` стандарттық класының қол жеткізуді тексеретін жеке `at()` функциясы және оны тексермейтін `operator[]()` функциясы бар. Осы шешімге негіздеме беріп көрелік. Ол төрт аргументке негізделеді:

1. *Үйлесімділік.* Адамдар C++ тілінде аластама пайда болғаннан көп бұрын берілген диапазон шегінен шығып кетуді тексермей индекстеуді пайдалана білген.

2. *Тиімділік.* Оптималды тиімді индекстеу операторы негізінде берілген диапазон шегінен шығып кетуді тексеретін оператор жасауға болады. Бірақ осындай тексеруді орындайтын қатынасу (қол жеткізу) операторы негізінде жасалған, шектен шығуды тексеруі жоқ, оптималды жылдамдығы бар индекстеу операторын жасау мүмкін емес.

3. *Шектеу.* Кейбір орталарда аластамалар жасауға болмайды.

4. *Міндетті емес (қосымша) тексеру.* Негізінде стандарт `vector` класында диапазонды тексере алмайсыз деп тұжырым жасамайды, сондықтан егер тексеру жасағыңыз келсе, оны жүзеге асыра аласыз.

19.4.1.1 Үйлесімділік

Адамдар ескі кодты қайта жөндегенді онша жаратпайды. Мысалы, сіз миллиондаған жолдардан тұратын кодтар жазсаңыз, ондағы аластамаларды дұрыс пайдалану үшін кодтарды қайта жазып жөндеу өте қымбатқа түсер еді. Біз, әрине, мұндай жұмыстан кейін код жақсы болып шығар дер едік, бірақ олай істемейміз, өйткені қосымша уақыттың және ақшаның шығын болуын қаламаймыз. Оның үстіне, бұрын жазылған кодтарды пайдаланатын адамдар тексерілмеген код қауіпсіз емес деп санайды, бірақ олардың қолданып жүрген программалары тексерілген және көп жылдардан бері жұмыс істеп келеді, оның барлық қателері түзетілген деп тұжырым жасайды. Мұндай аргументтерге сыни көзбен қарап, әрбір нақты жағдайда салмақты түрде шешім қабылдау қажет. Әрине, C++ тілінде пайда болғанша, `vector` класын қолданған программалар болған жоқ, бірақ аластамалары болмағанмен, осыларға ұқсас кодтардың миллиондаған жолдары болды. Осы программалардың басым бөлігі кейіннен стандарт талаптарына сай қайта жазылып шыққан болатын.

19.4.1.2 Тиімділік

Рас, экстремалды жағдайларда, мысалы, желілік интерфейстер буферлерінде және жоғары өнімді ғылыми есептеулердегі матрицаларды шығару кезінде диапазон шегінен шығып кетуді тексеру өте күрделі болуы мүмкін. Бірақ мүмкін болатын диапазон шегінен шығып кетуді тексерудің бағасы жиі орындалатын қарапайым есептеулерде сирек есепке алынады. Сонымен, біз `vector` класындағы берілген диапазон шегінен шығып кетуді тексеруді әрбір мүмкіндік болған сайын жүргізіп отыруды ұсынамыз.

19.4.1.3 Шектеу

Мұнда да алдыңғы пункттердегі сияқты аргументтерді әмбебап деп қарастыруға болмайды. Осыны барлық программалаушылар мойындайды, егер сіз нақты уақыт кезеңіндегі есептеулермен байланыспаған ортада (25.2.1 бөлімді қ.) жаңа программа жаза бастасаңыз, оны жай ғана алып тастауға да болмайды, мұндайда аластамалар арқылы қателерді өңдеу мен мүмкін болатін диапазон шегінен шығып кетуді тексеретін векторды пайдаланыңыз.

19.4.1.4 Міндетті емес тексеру

ISO C++ стандарты вектордың мүмкін болатын диапазон шегінен шығып кетуінің кепілдік беретін семантикасы жоқ, сондықтан оны пайдалануға онша тырыспау керек деген тұжырым жасайды. Стандартқа сәйкес мүмкін болатын

диапазон шегінен шығып кету орын алғанда аластама жасау (генерациялау) керек. Сол себепті егер сіз `vector` класы аластама жасап, алғашқы үш аргументке байланысты проблема туындатпасын десеңіз, нақты программаларыңызда мүмкін болатын диапазон шегінен шығып кетуді тексеретін `vector` класын пайдалану керек. Бұл кітапта біз осы қағиданы ұстанамыз.

Қысқаша айтқанда, нақты программа сіз қалағаннан гөрі күрделірек болуы мүмкін, бірақ әрқашанда дайын шешімдерді көшіріп алуға мүмкіндік бар.

19.4.2 Таным: макрос

Біздің `vector` класымыз сияқты стандартты `vector` класының да көптеген нұсқалары (`[ ]`) индекстеу операторының көмегімен мүмкін болатын диапазон шегінен шығып кетуді тексеруге кепілдік бермейді, оның орнына осындай тексеруді орындайтын `at()` функциясын қолданады. Біздің программамыздың қай жерінде `std::out_of _range` аластамасы туындайды? Негізінде, біз 19.4.1 бөлімінен 4 нұсқаны таңдап алдық: `vector` класының бұл жүзеге асырылуы `[ ]` операторы көмегімен мүмкін болатын диапазон шегінен шығып кетуді тексеруге тиісті емес, бірақ оны басқаша жасауға тыйым салынған, сондықтан біз осы мүмкіндікті пайдаланамыз деп шештік. Дегенмен, `Vector` атты біздің жөндеп түзету нұсқамызда, кодты құра отырып, біз тексеруді `[ ]` операторы арқылы жүзеге асырдық. Бұл программаның өнімділігін аздап төмендету арқылы кодты жөндеп түзету уақытын қысқарту мүмкіндігін береді.

```
struct Range_error : out_of_range {     // мүмкін болатын
// диапазон шегінен шығып кету жайлы толық мәлімет
    int index;
    Range_error(int i) :out_of_range("Range error"), index(i) { }
};

template<class T> struct Vector : public std::vector<T> {
    typedef typename std::vector<T>::size_type size_type;

    Vector() { }
    Vector(size_type n) :std::vector<T>(n) {}
    Vector(size_type n, const T& v) :std::vector<T>(n,v) {}

    T&operator[](unsignedinti)  //ratherthanreturnat(i);
    {
        if (i<0||this->size()<=i) throw Range_error(i);
        return std::vector<T>::operator[](i);
    }
```

```
    const T& operator[](unsigned int i) const
    {
        if (i<0||this->size()<=i) throw Range_error(i);
        return std::vector<T>::operator[](i);
    }
};
```

Біз индекстеу операциясын жөндеп түзетуді жеңілдету үшін **Range_error** класын пайдаланамыз. **typedef** операторы 20.5 бөлімінде толығырақ жазылған ыңғайлы синоним енгізеді.

Vector класы өте қарапайым, мүмкін, өте қарапайым шығар, бірақ ол онша күрделі емес программаларды жөндеп түзету үшін пайдалы болып табылады. Оған балама ретінде жүйелі түрдегі тексеруді қарастыратын стандартты **vector** класы нұсқасын пайдалануға тура келер еді, – *мүмкін*, бізге осыны жасау керек еді; сіздің компиляторыңыз бен кітапханаңыз қарастыратын тексерудің қаншалықты тыңғылықты екені жайлы ақпарат бізде жоқ (ол стандарт шегінен тысқары шығады).

std_lib_facilities.h тақырыбында біз **vector** сөзі **Vector** дегенді білдіреді деп көрсетіп, қитұрқы әрекетті (макроқойылымды) пайдаланамыз:

```
// мүмкін болатын диапазон шегінен шығып кетуді тексеретін
// векторды алу үшін жасалған өте ұнамсыз макрос
#define vector Vector
```

Бұл сіз **vector** сөзін жазсаңыз, компилятор оны **Vector** деп түсінетінін біледі. Бұл келеңсіздіктің жаман жері сіз компилятор көріп тұрған кодтан басқа кодты көресіз. Нақты программаларда макростар көптеген шатасып жатқан қателердің қайнар көзі болып табылады (27.8 және A.17 бөлімдері).

Біз **string** класы үшін мүмкін болатын диапазон шегінен шығып кетуді тексеру үшін осы тәсілді пайдаландық.

Өкінішке орай, **vector [ ]** класындағы **[ ]** операциясы арқылы мүмкін болатын диапазон шегінен шығып кетуді тексеруді жүзеге асыратын стандартты, жеңіл көшірілетін, айқын тәсіл жоқ. Дегенмен, бұл тексеруді **vector** және **string** кластарында дәлірек әрі толығырақ жүзеге асыруға болады. Бірақ көбінесе бұл стандартты кітапхананы іске асыруды ауыстырумен, инсталляция опцияларын (нұсқаларын) анықтаумен немесе стандартты кітапхана кодтарына кірісумен байланысқан болып жатады. Бұл мүмкіндіктердің біреуі де программалауға жаңа кіріскен жастарға қолайлы емес, сондықтан біз 2 тараудағы **string** класын пайдаландық.

19.5 Ресурстар мен аластамалар

Сонымен, **vector** класының объектісі аластамалар жасай алады, егер функция одан талап етілетін әрекетті орындай алмаса, ол аластама жасап, өзін шақырған модульге мәлімдеме беруін істеуді ұсынамыз (5-тарауды қ.). Енді **vector** класының операторлары және де басқа функциялар туындатқан аластамаларды өңдейтін кодты қалай жазуға болатынын ойлайтын кез келді. Ең оңай жауап – "аластамаларды ұстау үшін **try** блогын пайдаланып, қате туралы мәлімет шығарыңыз да, программа жұмысын тоқтатыңыз" – бұл онша күрделі емес көптеген жүйелер үшін өте қарапайым шешім.

Программалаудың іргелі қағидаларының бірі – егер біз ресурс сұрайтын болсақ, оны тікелей немесе басқаша түрде жүйеге кері қайтаруымыз керек. Жүйе ресурстарын тізбелеп қарастырып шығайық:

- компьютер жады (memory);
- бұғаттамалар (locks);
- файлдар дескрипторлары (file handles);
- ағындар дескрипторлары (thread handles);
- сокеттер (sockets);
- терезелер (windows).

Негізінде, ресурс – бұл бір алуға болатын, кейін өздігінен немесе ресурс менеджерінің талабы бойынша қайтарылуға (босатылуға) тиіс нәрсе. Ресурстың ең қарапайым мысалы болып **new** операторы арқылы алынып, **delete** операторы арқылы қайтарылатын бос жады аймағы есептеледі. Мысалы:

```
void suspicious(int s, int x)
{
    int* p = new int[s];  // жады аймағын аламыз
    // . . .
    delete[] p;           // жады аймағын босатамыз
}
```

Біз 17.4.6 бөлімде көргендей, жады аймағын босату қажеттілігі есте тұруы тиіс, бірақ оны орындау да онша оңай емес. Аластама жағдайды одан да қиындатып жібереді, соның нәтижесінде тиянақсыздықтан немесе аңғалдықтан ресурстарды жоғалту (азайту) туындауы мүмкін. Мысал ретінде тікелей **new** операторын пайдаланып, нәтижелдік нұсқауышты жергілікті (локальді) айнымалыға меншіктей отырып, өте қауіпті жағдай жасайтын **suspicious()** функциясын қарастырайық.

19.5.1 Ресурстарды басқарудың әлеуеттік мәселелері

Келесі нұсқауышты қарапайым зиянсыз түрде меншіктеуде кездесетін қауіпті бір жағдайды қарастырайық:

```
int* p = new int[s]; // жады аймағын пайдалануға аламыз
```

ол мұндағы **new** операторына **delete** операторының сәйкес келетінін тексерудің қиындығында болып отыр. **suspicious()** функциясында компьютер жадын босата алатын **delete[] p** нұсқауы бар, бірақ мұның орындалмай қалатын бірнеше себептерін қарастырып көрейік. Компьютер жадының азаюы үшін көпнүктте ... қойылған орынға қандай нұсқаулар қоюға болар екен? Осындайда туындайтын мәселелерді көрсету үшін біз таңдап алған мысалдар сізді ойландыруға және осындай кодтарға қатысты күдік туғызуға тиіс. Оның үстіне, осы мысалдарға қарап, сіз баламалы шешімнің қарапайымдылығы мен қуатын бағалай аласыз. Біздің **delete** операторы арқылы жойғымыз келген объектіге **p** нұсқауышы бұдан кейін сілтеме жасамауы мүмкін:

```
void suspicious(int s, int x)
{
    int* p = new int[s];        // жады аймағын аламыз
    // . . .
    if (x) p=q;     // p нұсқауышын басқа объектіге орнатамыз
    // . . .
    delete[] p;             // жады аймағын босатамыз
}
```

Біз сіздің алдын ала **p** нұсқауышының мәні өзгергендігін немесе өзгермегендігін білмеуіңізге кепілдік беретін **if (x)** нұсқауын программаға енгіздік. Программа **delete** операторын ешқашан да орындамауы мүмкін:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // жады аймағын аламыз
    // . . .
    if (x) return;
    // . . .
    delete[] p;             // жады аймағын босатамыз
}
```

Программа **delete** операторын ешқашан да орындамауы мүмкін, өйткені ол аластама жасап шығарады:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // жады аймағын аламыз
    vector<int> v;
    // . . .
    if (x) p[x] = v.at(x);
    // . . .
    delete[] p;          // жады аймағын босатамыз
}
```

Соңғы мүмкіндік бізді бәрінен де көп алаңдатады. Адамдар мұндай мәселемен алғаш рет кездескенде, олар мұны ресурстарды басқарумен емес, аластамалармен байланысты деп есептейді. Олар мәселенің нақты себептерін түсінбей, аластамаларды ұстауға тырысады:

```
void suspicious(int s, int x)     // нашар код
{
    int* p = new int[s];          // жады аймағын аламыз
    vector<int> v;
    // . . .
    try {
       if (x) p[x] = v.at(x);
       // . . .
    }catch (...) {        // барлық аластамаларды ұстаймыз
       delete[] p;        // жады аймағын босатамыз
       throw;             // қайтадан аластама жасаймыз
    }
    // . . .
    delete[] p;           // жады аймағын босатамыз
}
```

Бұл код қосымша нұсқаулар мен ресурстарды босататын кодтарды қосарлау арқылы мәселені шешеді (мұнда `delete[] p`; арқылы). Басқаша айтқанда, бұл әдемі шешім емес, мұнан да жаманы – оны жалпылама ету қиын. Біз бірнеше ресурстарды іске қостық делік.

```
void suspicious(vector<int>& v, int s)
{
    int* p = new int[s];
    vector<int>v1;
    // . . .
    int* q = new int[s];
    vector<double> v2;
```

```
    // . . .
    delete[] p;
    delete[] q;
}
```

Егер **new** операторы бос жады аймағын бөле алмайтын болса, ол стандартты **bad_alloc** аластамасын жасап шығаратынына назар аударыңыз. Бұл мысалда **try**. . . **catch** тәсілі де жақсы жұмыс істейді, бірақ бізге бірнеше **try** блоктары керек болады да, код қайталанатын әрі қауіпті болып шығады. Біз қайталанатын және шатасқан кодтарды жақсы көрмейміз, өйткені қайталанатын кодты сүйемелдеу қиын, ал шатасқан кодты сүйемелдеу ғана қиын емес, оны түсіну де қиын.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Соңғы мысалға **try** блогын қосыңыз да, барлық ресурстар кез келген аластамаларда дұрыс босатылатынына көз жеткізіңіз.

19.5.2 Ресурстарды алу – бұл инициалдау

Қуанышқа орай, ресурстарды азайтуды болдырмас үшін бізге **try**. . . **catch** нұсқауын көшіру міндетті емес. Келесі мысалды қарастырайық:

```
void f(vector<int>& v, int s)
{
    vector<int> p(s);
    vector<int> q(s);
    // . . .
}
```

Бұл енді жақсырақ. Ресурсты (мұнда бос жады аймағы) конструктор алады да, өзіне сәйкес деструктормен босатады. Біз енді аластамаларға байланысты өзіміздің нақты есебімізді шығардық. Бұл есеп әмбебап сипатта шығарылған; оны барлық ресурс түрлеріне қолдануға болады: конструктор өздерін басқарып отырған объект үшін ресурс алады, ал соған сәйкес деструктор оларды қайтарады. Осындай тәсіл *мәліметтер базасын бұғаттаумен* (database locks), *сокеттермен* (sockets) және *енгізу*-шығару *буферлерімен* (I/O buffers) жұмыс істеу кезінде бәрінен де жақсы болып көзге түсті. Осыған сәкес қағида әдетте қисынсыздау айтылады: "Ресурс алу дегеніміз – инициалдау" ("Resource Acquisition Is Initialization", қысқашасы – RAII).

Алдыңғы мысалды қарастырайық. Біз **f()** функциясынан шыққан соң, **p** және **q** векторларының деструкторлары шақырылатын болады: өйткені **p** және **q** айны-

малылары нұсқауыш емес, біз оларға жаңа мәндер меншіктей алмаймыз, **return** нұсқауы деструкторларды шақыруға тосқауыл қоя алмайды және ешқандай аластамалар жасалмайды да. Бұл әмбебап ереже: басқару ағыны көріну аймағын тастап кеткенде, толығынан құрылған әрбір объект пен екпінді етілген ішкі объект үшін деструкторлар шақырылады. Егер объект конструкторы өз жұмысын аяқтаған болса, ол толық құрылған болып саналады. Осы екі тұжырымнан шығатын барлық мүмкіндіктерді (салдарды) зерттеу басты ауыртуы мүмкін. Тек жай ғана конструкторлар қашан және қай жерде керек болса, сол жерде шақырылады деп санайтын боламыз.

Бөле жарып айтар болсақ, егер көріну аймағында айнымалы көлемдегі бос жады аймағын бөлгіңіз (ерекшелегіңіз) келсе, тек жалқыланған **new** және **delete** операторларын емес, **vector** класын пайдалануды ұсынамыз.

19.5.3 Кепілдіктер

Егер векторды бір ғана көріну аймағымен (немесе оның ішкі аймағымен) шектеу мүмкін болмаса, не істеу керек? Мысал қарастырайық:

```
vector<int>* make_vec()      // толтырылған вектор құрады
{
    vector<int>* p = new vector<int>;
    // бос жады аймағын бөлеміз
    // . . .  векторды мәліметтермен толтырамыз,
    // аластама жасалуы мүмкін . . .
    return p;
}
```

Бұл кең таралған мысал: күрделі мәліметтер құрылымын жасау үшін біз функцияны шақырамыз да, осы құрылымды нәтиже ретінде қайтарамыз. Алайда, егер векторды толтыру кезінде аластама туындаса, **make_vec()** функциясы **vector** класының осы объектісін жоғалтып алады. Бұған қоса, егер функция өз жұмысын дұрыс аяқтаса, онда біреу **make_vec()** функциясы қайтарған объектіні жоюы керек (17.4.6 бөлімді қ.).

Аластама жасау үшін, біз **try** блогын қоса аламыз:

```
vector<int>* make_vec()      // толтырылған вектор құрады
{
    vector<int>*p=newvector<int>; //босжадыаймағынбөледі
    try {         // . . .векторды мәліметтермен толтырамыз;
                  // аластама жасалуы мүмкін . . .
      return p;
      }
```

```
    catch (...) {
      delete p;  // локальді тазалау
      throw;     // some_function() функциясы талап етілген
      // әрекетті орындамаса, шақыратын функция оған
      // жауап беретіндей етіп қайтадан аластама жасаймыз
    }
}
```

Бұл **make_vec()** функциясы қателерді өңдеудің өте кең таралған стилін көрсетеді: программа өз тапсырмасын орындауға тырысады, ал егер орындай алмаса, барлық жергілікті (локальді) ресурстарды (мұнда **vector** класы объектісі орналасқан бос жады аймағын) босатады да, ол туралы мәлімет беріп, аластама жасайды. Мұнда аластаманы басқа функция (**vector::at()**) жасап шығарады, **make_vec()** функциясы **throw;** операторы көмегімен аластама жасау әрекетін жай ғана қайталайды. Бұл – тұрақты түрде пайдалануға болатын, қателерді өңдейтін қарапайым әрі тиімді тәсіл.

- *Базалық кепілдік*. **try... catch** кодының мақсаты **make_vec()** функциясы жұмысты дұрыс аяқтайтынына немесе ресурстарды азайтпай, аластама жасайтынына кепілдік беру болып саналады. Мұны көбінесе *базалық кепілдік* (basic guarantee) деп атайды. Аластама жасалғаннан кейін өз жұмысын қалпына келтіретін программа бөлігі болып табылатын код толығымен базалық кепілдікті сүйемелдеуі тиіс.

- *Қатаң кепілдік*. Егер қателік туындағанда, барлық қарастырылатын мәндер өзінің бұрынғы мәндерін қалпына келтіре алатынына функция кепілдік бере алатын болса, онда мұндай функция, базалық кепілдіктен бөлек, *қатаң кепілдік* (strong guarantee) береді деп айтылады. Қатаң кепілдік – бұл функция үшін идеалды жағдай: мұнда функция күтілгендей түрде орындалады немесе қате кетіп аластама жасалғанынан басқа ешнәрсе де атқарылмайды.

- *Аластама болмағанына кепілдік* (no-throw guarantee). Егер біз ақау шықпайды деп тәуекелдік етіп, аластама жасамай қарапайым операцияларды орындай алмасақ, онда базалық және қатаң кепілдіктер шартына сәйкес код жаза алмас едік. Қуанышқа орай, C++ тілінің құрамындағы барлық құралдар аластама болмағанына кепілдік беруді сүйемелдейді: олар оны жасай алмайды. Аластама жасауды болдырмас үшін **throw**, **new** операторларын орындамаңыз және сілтемелік типтерге **dynamic_cast** операторын қолданбаңыз (A.5.7 бөлімі).

Программаның дұрыстығын талдау үшін базалық және қатаң кепілдіктер пайдалы болып есептеледі. Осы идеяларға сәйкес жазылған қарапайым әрі тиімді код-

ты жүзеге асыру үшін RAII қағидасы түбегейлі рөл атқарады. Толығырақ ақпаратты *C++ программалау тілі* кітабының E қосымшасынан табуға болады.

Әрине, нөлдік нұсқауышты атаусыз ету, нөлге бөлу және мүмкін болатын диапазон шегінен шығып кету сияқты анықталмаған операцияларды пайдаланбауға тырысу керек. Аластамаларды қолдану тілдің іргелі ережелерін сақтауды алып тастай алмайды.

19.5.4 auto_ptr класы

Сонымен, `make_vec()` сияқты функциялар аластамаларды пайдаланып ресурстарды дұрыс басқарудың негізгі ережелеріне бағынады. Бұл программаның жұмысын аластама жасалғаннан кейінгі қалпына келтіру кезіндегі барлық дұрыс жұмыс істейтін функциялар беруге тиіс базалық кепілдікті орындауды қамтамасыз етеді. Егер программаның векторды мәліметтермен толтыратын бөлігінде локальді емес айнымалылармен келеңсіз жағдайлар туындамаса, онда мұндай функциялар қатаң кепілдік береді деп тұжырым жасауға әбден болады. Бірақ мұндағы `try. . . catch` блогы бұрынғыша қисынсыз болып көрінеді. Мұның шешімі түсінікті шығар: бірдеңе етіп RAII қағидасын қолдану керек, басқаша айтқанда, `vector<int>` класының объектісін иеленетін объект қарастыру керек, егер аластама туындаса, ол оны жоя алатын болуы тиіс. Осы әрекетті жасау үшін стандартты кітапхананың `<memory>` тақырыбында `auto_ptr` класы бар:

```
vector<int>* make_vec() // толтырылған вектор жасайды
{
    auto_ptr< vector<int> > p(new vector<int>);
    // бос жады бөледі
    // ... векторды мәліметтермен толтырамыз;
    //     аластама жасалуы мүмкін ...
    return p.release();
    // p объектісі иеленген нұсқауышты қайтарамыз
}
```

`auto_ptr` класының объектісі функция нұсқауышына иелік етеді. Ол бірден `new` операторы арқылы жасалған объектімен инициалданады. Енді біз `auto_ptr` класының объектісіне қарапайым нұсқауыш сияқты `->` және `*` операторларын қолдана аламыз, сол себепті `auto_ptr` класының объектісін нұсқауыштың бір түрі деп санауға болады. Бірақ осыған сәйкес құжаттаманы оқымай, `auto_ptr` класын көшіруге асықпаңыз; бұл кластың семантикасы біз осы уақытқа дейін кездестірген кез келген типтің семантикасынан айрықша болады. `release()` функциясы `auto_ptr` класының объектісіне қарапайым нұсқауышты қайтаруға мәжбүр етеді, сонымен біз бұл нұсқауышты қайтара аламыз. Ал `auto_ptr` класының объектісі қайтарылған нұсқауыш орнатылған объектіні жоя алмайды. Егер

сіз **auto_ptr** класын бұдан қызықтырақ жағдайларда (мысалы, оның объектісін көшіру) пайдалануға асықсаңыз да, шыдай тұруға тура келеді. **auto_ptr** класы нұсқауышты иелену үшін және көріну аймағынан шығу кезінде объектіні жоюға кепілдік беру үшін қажет. Бұл класты басқаша пайдалану асқан шеберлікті талап етеді. **auto_ptr** класы **make_vec()** сияқты функцияларды қарапайым әрі тиімді түрде жүзеге асыруды қамтамасыз ету арнайы құрал болып табылады. Бөле жара айтқанда, **auto_ptr** класы бізге біздің берген кеңесімізді қайталау мүмкіндігін береді: **try** блоктарын тікелей пайдалануға күмәнмен қараңыз; олардың көбін **RAII** ("Resource Acquisition Is Initialization") қағидаларының бірін пайдалану арқылы алмастыруға болады.

19.5.5 vector класы үшін RAII қағидасы

Тіпті **auto_ptr** сияқты зерделі нұсқауыштарды пайдаланудың өзі жеткілікті деңгейде қауіпсіз болып көрінбеуі мүмкін. Қорғанысты талап ететін барлық нұсқауыштарды анықтадық деп қалай сенімді бола аламыз? Көріну аймағының соңында жойылмауы тиіс барлық нұсқауыштарды босаттық деп қалай сенімді бола аламыз? 19.3.5 бөліміндегі **reserve()** функциясын қарастырайық.

```
template<class T, class A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return;
    // көлем ешқашанда азаймайды
    T* p = alloc.allocate(newalloc);
    // жаңа жады аймағын бөлеміз

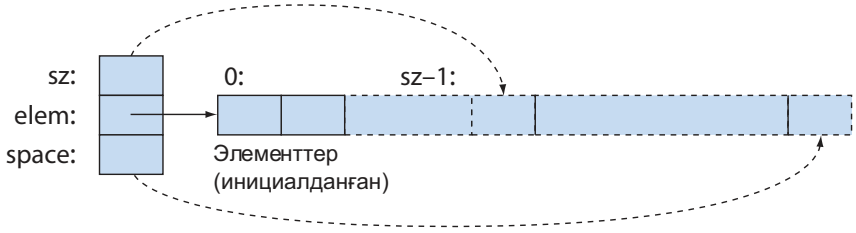
    for(int i=0; i<sz; ++i) alloc.construct(&p[i],elem[i]);
    // көшіру

    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i]);
    // жою

    alloc.deallocate(elem,space);
    // бұрынғы жады аймағын босату
    elem = p;
    space = newalloc;
}
```

Ескі элементті көшіретін **alloc.construct(&p[i],elem[i])** операциясының аластама туындата алатынына назар аударыңыз. Сондықтан **p** нұсқауышы – біз 19.5.1 бөлімінде ескерткен мәселенің мысалы болып табылады. Ой! **auto_**

`ptr` класын пайдалануға болатын еді ғой. Ал бұдан да жақсысы – кейінге оралып, вектор үшін бөлінетін жадының ресурс екенін түсіну болар, басқаша айтқанда, біз барлық уақытта пайдаланатын іргелі тұжырымдаманы өрнектеу үшін `vector_base` класын анықтай аламыз. Бұл тұжырымдама векторға арналған жады аймағын пайдалануды анықтайтын үш элементі бар келесі суретте көрсетілген:



Бейненің толық болуы үшін жады бөлгішті қоса отырып, келесі кодты аламыз:

```
template<class T, class A>
struct vector_base {
    A alloc;          // жады бөлгіш
    T* elem;          // бөлуді бастау
    int sz;           // элементтер саны
    int space;        // бөлінген жады көлемі

    vector_base(const A& a, int n)
      : alloc(a), elem(a.allocate(n)), sz(n), space(n) { }
    ~vector_base() { alloc.deallocate(elem,space); }
};
```

`vector_base` класының типтелген объектілермен емес, жадымен жұмыс істейтініне назар аударыңыз. Біздің `vector` класының жүзеге асырылуын элементтің керекті типі бар объектіні иеленуі үшін пайдалануға болады. Негізінде, `vector` класы `vector_base` класы үшін жай ғана ыңғайлы интерфейс болып табылады.

```
template<class T, class A = allocator<T> >
class vector : private vector_base<T,A> {
public:
    // . . .
};
```

Енді `reserve()` функциясын қарапайымдау әрі дұрысырақ етіп қайта жазып шығайық:

```
template<class T, class A>
void vector<T,A>::reserve(int newalloc)
{ if (newalloc<=space) return; // көлем ешқашанда азаймайды
vector_base<T,A> b(alloc,newalloc);
// жаңа жады аймағын бөлу
for(int i=0;i<sz;++i) alloc.construct
                                    (&b.elem[i],elem[i]);
                                    // көшіру
for (int i=0;i<sz;++i) alloc.destroy(&elem[i]);
// жадыны босату
swap< vector_base<T,A> >(*this,b);
// бейнелердің орнын ауыстыру
}
```

reserve() функциясынан шығарда, ол тіпті аластама туындатқан көшіру операциясы арқылы шығатын болса да, ескі жады аймағы **vector_base** класының деструкторы арқылы босатылады. **swap()** функциясы екі объектінің орнын алмастыратын стандартты кітапханалық алгоритм (**<algorithm>** тақырыбынан) болып табылады. Біз өте қарапайым болып келетін **swap(*this,b)** функциясын емес, **swap< vector_base<T,A> >(*this,b)** алгоритмін пайдаландық, өйткені ***this** және **b** объектілерінің типтері әртүрлі (сәйкесінше **vector** және **vector_base**), сондықтан **swap** алгоритмінің қандай мамандандырылған түрін орындау керек екендігін тікелей көрсету керек.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

reserve функциясын, ол **auto_ptr** класын пайдаланатындай етіп, толықтырыңыз. Функциядан шығарда жады аймағын босату керек екендігі есіңізде болсын. Осының нәтижесін **vector_base** класымен салыстырыңыз. Олардың қайсысының жақсы екенін және қайсысының жеңіл жүзеге асырылатынын анықтау қажет.

ТАПСЫРМА

1. `<class T> struct S { T val; };` класын анықтаңыз.
2. Оны `T` типімен инициалдауға болатындай етіп, оған конструктор қосыңыз.
3. `S<int>`, `S<char>`, `S<double>`, `S<string>` және `S<vector <int> >;` типтеріндегі айнымалыларды анықтаңыз; оларды өз таңдауларыңызға сәйкес мәндермен инициалдаңыз.
4. Сол мәндерді оқып, экранға шығарыңыз.
5. `val` мәніне сілтеме қайтаратын `get()` шаблондық функциясын қосыңыз.
6. `get()` функциясын кластан тысқары орналастырыңыз.
7. `val` мәнін жабық бөлімде орналастырыңыз.
8. `get()` функциясын пайдаланып, 4 п. орындаңыз.
9. `val` мәнін өзгертуге болатын `set()` шаблондық функциясын қосыңыз.
10. `get()` және `set()` функцияларын `operator[]` операторымен алмастырыңыз.
11. `operator[]` операторының константалық және константалық емес нұсқаларын жазып шығыңыз.
12. `cin` ағынынан `v` айнымалысына мәліметтер енгізетін `template<class T> read_val(T& v)` функциясын анықтаңыз.
13. 3 п. тізіп көрсетілген айнымалылардың `S< vector<int> >` айнымалысынан басқаларына мәлімет оқу үшін, `read_val()` функциясын пайдаланыңыз.
14. Бонус: `read_val()` функциясы `S< vector<int> >` айнымалысын өңдейтіндей етіп, `template<class T> ostream& operator<<(ostream&, vector<T>&)` класын анықтаңыз. Әрбір кезеңнен соң тесттен өткізуді ұмытпаңыз.

БАҚЫЛАУ СҰРАҚТАРЫ

1. Вектор көлемін неге өзгерту керек?
2. Әртүрлі типтегі элементтері бар әртүрлі векторлар не үшін керек?
3. Біз неге векторларға бірден жадының үлкен көлемін беріп сақтамаймыз?
4. Біз жаңа векторға бос жады көлемінің қанша аймағын бөлеміз?
5. Вектор элементтерін неге жаңа жады аймағына көшіреміз?
6. Векторды құрғаннан кейін `vector` класының қандай операциялары оның көлемін өзгерте алады?
7. `vector` класының объектісі ол көшірілгеннен кейін неге тең болады?

8. Қандай екі операция вектор көшірмесін анықтайды?
9. Келісім (үнсіз) бойынша класс объектілерін көшірудің қандай мағынасы бар?
10. Шаблон деген не?
11. Шаблондық аргументтердің ең пайдалы екі түрін айтып беріңіз.
12. Жалпыланған программалау дегеніміз не?
13. Жалпыланған программалаудың объектіге бағытталған программалаудан қандай айырмашылығы бар?
14. `array` класының `vector` класынан айырмашылығы неде?
15. `array` класының құрамдас типтегі жиымнан айырмашылығы неде?
16. `resize()` функциясының `reserve()` функциясынан айырмашылығы неде?
17. Ресурс деген не? Оған анықтама беріп, мысал келтіріңіз.
18. Ресурстың азаюы деген не?
19. `RAII` қағидасы деген не? Ол қандай мәселелерді шешеді?
20. `auto_ptr` класы қандай жұмыс атқарады?

ТЕРМИНДЕР

`#define`	`throw;`	мамандану
`at()`	аластама	нақтылау (конкретизация)
`auto_ptr`	базалық кепілдік	өзіндік меншіктеу
`push_back()`	кепілдіктер	ресурс
`RAII`	қатаң кепілдік	шаблон
`resize()`	қайта жасау (генерациялау)	шаблондық параметр
`this`	макрос	

ЖАТТЫҒУЛАР

Әрбір жаттығуда белгілі бір кластағы объектілер жиынын жасап, оларды тексеріп (баспаға шығарып) шығыңыз да, сіздің жобаңыз және оның жасалған нұсқасы сіз күткен дәрежеде жұмыс істейтінін көрсетіңіз. Оның аластама жасалған жерлеріндегі қате туындайтын орындарды тыңғылықты түрде қарап шығу керек болуы мүмкін.

1. Қосуды орындауға болатын элементтері кез келген типтегі векторларды қосатын шаблондық функция жазыңыз.
2. Аргументтер ретінде `vector<T> vt` және `vector<U> vu` типіндегі

объектілер алып, барлық `vt[i]*vu[i]` өрнектерінің қосындысын қайтаратын шаблондық функция жазыңыз.

3. Кез келген типтегі екі мәні болатын `Pair` шаблондық класын жазыңыз. Оны калькулятордағы тәрізді қарапайым сөз тіркестері кестесін жасау үшін пайдаланыңыз (7.8 бөлімін қ.).
4. 17.9.3 бөліміндегі `Link` класын шаблондық класқа айналдырыңыз. Сонан соң 17-тараудағы 13-жаттығуды `Link<God>` класы негізінде қайта орындап шығыңыз.
5. `int` типіндегі жалғыз мүшесі бар `Int` класын анықтаңыз. Сол үшін конструкторларды, меншіктеу операторын және `+, -, *, /` операторларын анықтаңыз. Осы класты тесттен өткізіп, қажет болса, оның құрылымын айқындаңыз (мысалы, қарапайым енгізу-шығару үшін `<<` және `>>` операторларын анықтаңыз.
6. Алдыңғы жаттығуды `Number<T>` класы үшін қайталаңыз, мұндағы `T` – кез келген сандық тип. `Number` класына `%` операторын қосып көріңіз де, `%` операторын `Number<double>` және `Number<int>` типтеріне қолдануға тырысқанда, не болғанына қараңыз.
7. `Number` типіндегі бірнеше объектілерге 2 жаттығудың шешімін қолданып көріңіз.
8. `malloc()` және `free()` (B.10.4 бөлімі) функцияларын пайдалана отырып, жады бөлгішті жүзеге асырыңыз. Бірнеше тесттік мысалдармен жұмыс істеу үшін, 19.4 бөлімнің соңында сипатталған сияқты `vector` класын құрыңыз.
9. Жадыны басқару үшін `allocator` класын (19.3.6 бөлімін қ.) қолдана отырып, `vector::operator=()` (19.2.5 бөлімін қ.) функциясын жүзеге асыруды қайталаңыз. Тек меншіктеу немесе көшіретін конструктор жасауға ұмтылмаңыз.
10. Тек конструкторы, деструкторы, `->, *` операторлары және де `release()` функциясы бар қарапайым `auto_ptr` класын жүзеге асырыңыз. Тек меншіктеу немесе көшіретін конструктор жасауға ұмтылмаңыз.
11. `T` типіндегі объектіге нұсқауышы бар `counted_ptr<T>` класын және сілтемелер санын (`int` типті айнымалы) есептейтін, барлық нұсқауыштарға ортақ, `T` типіндегі бір ғана объектіге сілтемелерді санайтын нұсқауышты жасап жүзеге асырыңыз. Сілтемелерді есептейтін санауышта `T` типіндегі осы сілтеме жасайтын объектіге санауыштар саны болуы тиіс. `counted_ptr` класының конструкторы бос жады аймағында `T` типіндегі объект пен сілтемелер санауышын орналастыруы тиіс. `counted_ptr` класының объектісіне `T` типіндегі бастапқы мән меншіктеңіз. `T` класы үшін `counted_ptr` класының соңғы объектісін жойғаннан кейін оның деструкторы `T` класының объектісін жоюы тиіс. `counted_ptr` класында оны нұсқауыш ретінде пайдалануға мүмкіндік беретін операциялар

қарастырыңыз. Бұл ақырғы қолданушы оған сілтеме жасауды тоқтатқанша объект жойылмайтынына кепілдік беретін "зерделі нұсқауыш" деп аталатын әрекет мысалы. `counted_ptr` класы үшін, оның объектілерін функцияларды шақырғанда аргументтер, контейнер элементтері, т.с.с. ретінде пайдаланатын тесттер жиынын жазып шығыңыз.

12. Конструкторы `string` типіндегі аргумент алып, файл ашатын, ал оның деструкторы файлды жабатын `File_handle` класын анықтаңыз.
13. Конструктор сөз тіркестерін енгізетін, ал деструктор шығаратын `Tracer` класын жазып шығыңыз. Конструктор аргументтері сөз тіркесі болуы тиіс. Осы мысалды `RAII` қағидасына сәйкес объектілер қалай жұмыс істейтінін көрсету үшін қолданыңыз (мысалы, локальді объектілер, кластың объект-мүшелері, глобальді объектілер, `new` операторы арқылы орналасқан объектілер, т.с.с. рөлін ойнайтын `Tracer` класының объектілерімен тәжірибе жасаңыз). Сонан соң көшіру кезеңінде `Tracer` класының объектілерінің не істейтінін көру үшін көшіретін конструктор мен көшіретін меншіктеу әрекеттерін қосыңыз.
14. "Вампус аулау" (18-тарауды қ.) ойыны үшін графикалық қолданушы интерфейсін және шығару құралдарын жасап шығыңыз. Редакциялау терезесінен мәліметтер енгізу әрекетін қарастырып, ойыншыға белгілі үңгір бөлігінің картасын экранға шығарыңыз.
15. "Жарғанаттар болуы мүмкін" және "түпсіз тұңғиық" сияқты білім мен болжамға негізделе отырып, қолданушыға бөлмелерді белгілеу мүмкіндігін беру үшін алдыңғы жаттығудағы программаны толықтырып шығыңыз.
16. Кейде бос вектордың барынша шағын көлемде болғаны дұрыс болады. Мысалы, көптеген векторлары бос болып келетін `vector< vector< vector<int> > >` класын екпінді түрде пайдалануға болады. `sizeof(vector<int>)==sizeof(int*)` шарты орындалатындай түрде, яғни вектор класы жиым элементтеріне нұсқауыштан, элементтер санынан және `space` нұсқауышынан тұратындай етіп, вектор анықтаңыз.

СОҢҒЫ СӨЗ

Шаблондар мен аластамалар өте қуатты тіл конструкторлары болып табылады. Олар өте икемді программалау технологияларын сүйемелдейді, негізінде, жауапкершіліктерді бөлу арқылы, яғни кез келген уақыт сәтінде бір мәселені ғана шешіп алу керек. Мысалы, шаблондарды пайдаланған кезде vector сияқты контейнерді оны элементтерінің типін жариялаудан бөле отырып, анықтай аламыз. Сол сияқты қателерді анықтап, кодтан тыс соларды өңдеуге арналған, қате туралы мәлімет беретін код жазуға болады. Вектор көлемін өзгертуге байланысты үшінші негізгі тақырып салыстырмалы түрде өте қарапайым болып келеді: `push_back()`, `resize()` және `reserve()` функциялары вектордың анықталуын оның көлемін спецификалаудан бөлуге мүмкіндік береді.

20

Контейнерлер мен итераторлар

"Бір ғана затты жасайтын және оны өте жақсы жасайтын программа жазыңыз. Программаларды бірге жұмыс жасау үшін жазыңыз".

- Дуг Макилрой (Doug McIlroy)

Осы және келесі тараулар контейнерлер мен алгоритмдерден тұратын C++ тілінің стандартты кітапханасының бір бөлігі болып табылатын STL кітапханасына арналған. STL кітапханасы – бұл C++ тілінде жазылған программадағы мәліметтерді өңдеуге арналған масштабталған қаңқа тәрізді. Біз алдымен қарапайым мысал қарастырамыз, сонан соң ортақ идеялар мен негізгі тұжырымдамаларды (концепцияларды) баяндаймыз. Біз байланысқан тізімдер арқылы итерация, манипуляция түсініктерін, сонымен қатар STL кітапханасының контейнерлерін қарастырамыз. Контейнерлер (мәліметтер) мен алгоритмдер (өңдеу) арасындағы байланыс тізбектер мен итераторлар арқылы қамтамасыз етіледі. Бұл тарауда келесі тарауда сипатталатын әмбебап, тиімді, әрі пайдалы алгоритмдердің негіздері баяндалған. Қарапайым қосымша мысалы ретінде мәтінді түзету (редакциялау) әрекеті қарастырылады.

20.1 Мәліметтерді сақтау және өңдеу

Аса көлемді мәліметтер топтамасын (коллекциясын) зерттемес бұрын біз мәліметтерді өңдеуге байланысты бірсыпыра есептер класын шешетін тәсілдерді бейнелейтін қарапайым мысал қарастырамыз. Джек пен Джилл автомобиль жылдамдықтарын жылжымалы нүктелі сандар түрінде жазып отыр делік. Джек С тілінде программалайды да, өз мәліметтерін жиымда сақтайды деп, ал Джилл өзінің өлшеген мәліметтерін `vector` класының объектісінде жазады деп санайық. Біз олардың мәліметтерін өз программамызда пайдаланғымыз келсе, оны қалай жасауға болар екен?

Біз кейін өз программамызға оқып алу үшін Джек пен Джилл программаларындағы мәндерді файлға жазып отырсын деп талап қоялық. Мұндайда біз Джек пен Джилл жасаған мәліметтер құрылымы мен оның интерфейсіне тәуелді болмаймыз. Көбінесе, осылай жекелеп жазу (изоляция) тәсілі толығымен дұрыс нәтиже береді. Мұны жүзеге асыру үшін біз өз есептерімізде 10 және 11-тарауларда сипатталған енгізу амалдары мен `vector<double>` класын пайдалануымызға болады.

Алайда, біздің есебімізді шығару үшін файлдарды қолдану күрделіленіп кететін болса не істеу керек? Мәліметтерді тіркеу коды функция түрінде жазылған болсын делік және ол әр секунд сайын өзгеріп отыратын жаңа мәліметтер жиынтығынан тұратыны белгілі болсын. Мұндай жағдайда, біз өңдеуге арналған мәліметтерді дер кезінде алып отыру үшін, секунд сайын Джек мен Джилдің функцияларын шақыруға мәжбүр боламыз.

```
double* get_from_jack(int* count)
//Джек double типіндегі сандарды жиымға жазып,
// *count жиымы элементтерінің санын қайтарады

vector<double>* get_from_jill();
// Джилл векторды толтырады

void fct()
{
      int jack_count = 0;
      double* jack_data = get_from_jack(&jack_count);
      vector<double>* jill_data = get_from_jill();
      // ... өңдейміз . . .
      delete[] jack_data;
      delete jill_data;
}
```

Біз бұл мәліметтер бос жады аймағында сақталады деп санаймыз, сондықтан оларды өңдегеннен кейін өшіріп отыру керек. Бұған қоса айтарымыз, біз Джек пен Джилл жазған кодтарды қайтадан жаза алмаймыз (немесе оны жазғымыз келмейді).

20.1.1 Мәліметтермен жұмыс істеу

Әрине, бұл мысал жеңіл түрде берілгенмен, ол көптеген басқа нақты есептерге өте ұқсас болып келеді. Егер біз бұл есепті әдемі шешетін болсақ, онда программалаудың басқа да көптеген есептерін шығара аламыз. Мұндағы іргелі мәселе, біз тапсырыс берушілер таңдап алған мәліметтерді сақтау тәсіліне әсер ете алмаймыз. Біздің міндетіміз – мәліметтерді қандай түрде алсақ, сондай түрде өңдеу немесе оларды оқып алып, өзімізге ыңғайлырақ түрде жазып алып барып пайдалану.

Мұндағы мәліметтермен біздің не істегіміз келеді? Оларды ретке келтіру ме? Олардың ең үлкен мәнін табу ма? Орташа мәнін есептеу ме? 65-тен үлкен барлық мәндерін табу ма? Джилл мен Джектің мәліметтерін салыстыру ма? Элементтер санын анықтау ма? Мұндай мүмкіншіліктер саны шектеусіз. Біз нақты бір программаны жазған кезде талап етілген есептеулерді орындаймыз. Осы жағдайда біз мәліметтерді қалай өңдейтінімізді және сандардың көлемді жиымымен қандай есептеулер орындайтынымызды нақтылап алғымыз келеді. Алдымен ең қарапайым әрекет орындайық: мәліметтер топтарының әрқайсысындағы ең үлкен элементті табайық. Ол үшін `...өңдеу...` комментарийін соған сәйкес нұсқаулармен ауыстыру қажет.

```
// . . .
double h = -1;
double* jack_high;// jack_high – ең үлкен элементке нұсқауыш
double* jill_high;// jill_high – ең үлкен элементке нұсқауыш

for(int i=0; i<jack_count; ++i)
    if (h<jack_data[i])
      jack_high = &jack_data [i];
      // ең үлкен элемент адресін сақтау
      h = jack_data[i];    // үлкен элементті жаңарту
    }

h = -1;
for (int i=0; i< jill_data ->size(); ++i)
    if (h<(*jill_data)[i])
      jill_high = &(*jill_data) [i];
      //ең үлкен элемент адресін сақтау
      h = (*jill_data)[i]; // үлкен элементті жаңарту
    }

cout << "Джилл максимумы: " << *jill_high
     << "; Джек максимумы: " << *jack_high;

// . . .
```

Джилл мәліметтеріне қол жеткізу үшін қолданылған келеңсіз конструкцияға: `(*jill_data) [i]` назар аударыңыз. `from_jill()` функциясы нұсқауышты `vector<double>*` векторына қайтарады. Мәліметтерді алу үшін біз алдымен, `*jill_data` нұсқауышы көмегімен векторға қол жеткізіп, оны атаусыз етуіміз керек, сонан соң оған индекстеу операциясын қолдануға тиіспіз. Бірақ та `*jill_data[i]` өрнегі нақ біз ойлағандай емес, ол `*(jill_data[i])` дегенді білдіреді, өйткені `[]` операторының приоритеті (орындалу реттілігі) * операторына қарағанда жоғары болғандықтан, бізге `*jill_data` конструкциясын жақшаға алу керек, яғни ол `(*jill_data) [i]` өрнегі түрінде болады.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Егер сіз Джиллдің кодын өзгерте алатын болсаңыз, келеңсіз конструкцияларды қолданбау үшін, интерфейсті қалай өзгертер едіңіз?

20.1.2 Кодты жалпылау

Мәліметтерді бейнелеу аздап өзгерген сайын программаны қайта-қайта көшіріп жазбау үшін, бізге мәліметтерге қол жеткізудің және қолдан жылдам өңдеудің (манипуляциялау) бірыңғай тәсілі қажет. Джек пен Джиллдың кодтарын қарастырып, оларды көбірек абстрактылы және біртекті етіп жасап шығамыз.

Әрине, Джектің мәліметтерімен не істесек, олар Джиллдың мәліметтеріне де қатысты болып табылады. Дегенмен, олардың программаларының арасында өкінішті екі айырмашылық бар: `jack_count` және `jill_data->size()` айнымалылары, сонымен қоса `jack_data[i]` және `(*jill_data)[i]` конструкциялары. Соңғы айырмашылықты сілтеме енгізу арқылы алып тастауға болады.

```
vector<double>& v = *jill_data;
for (int i=0; i<v.size(); ++i)
  if (h<v[i]){
      jill_high = &v[i];
      h= v[i];
  }
```

Бұл Джектің мәліметтерін өңдеуге арналған кодтарына өте ұқсас. Мүмкін, Джиллдың да, Джектың да мәліметтерінің есептеулерін орындайтын функция жазу керек шығар? Мұның әртүрлі жолдары болуы мүмкін (3-жаттығуды қ.), бірақ кодты жалпылаудың жоғарғы деңгейіне жетуге талпына отырып (келесі екі тарауды қ.), біз нұсқауыштарға негізделген шешімді таңдадық.

```
double* high(double* first, double* last)    //нұсқауышты
// first,last диапазонындағы ең үлкен элементке қайтарады
{
    double h = -1;
    double* high;
    for(double* p = first; p!=last; ++p)
      if (h<*p) {high = p; h=*p;}
    return high;
}
```

Енді келесі кодты жазуға болады:

```
double* jack_high=high(jack_data,jack_data+jack_count);
vector<double>& v = *jill_data;
double* jill_high = high(&v[0],&v[0]+v.size ());
```

Бұл әлдеқайда жақсы көрінеді. Біз онша көп айнымалыларды енгізген жоқпыз және тек бір цикл (`high()` функциясына) ғана жаздық. Егер біз ең үлкен элементті тапқымыз келсе, онда мынадай `*jack_high` және `*jill_high` мәндерге қарасақ болады. Мысал қарастырайық.

```
cout << " Джилл максимумы: n << *jill_high
     << "; Джек максимумы: " << *jack_high;
```

Мынаған назар аударыңыз: вектор мәліметтерін жиымда сақтайтынын `high()` функциясы пайдаланады, сондықтан біз өзіміздің ең үлкен мәнді іздейтін алгоритмімізді жиым элементтеріне сілтейтін нұсқауыштар терминдері арқылы көрсете аламыз.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Осы шағын программада біз екі әлеуетті қауіпті қатені қалдырдық. Оның біреуі апатты жағдайды туындатса, екіншісі, егер `high()` функциясы басқа программаларда қолданылатын болса, дұрыс емес жауаптарға әкеледі. Төменде көрсетілген әмбебап тәсіл осы екі қатені де айқындап, олардан қалай құтылуға болатынын көрсетеді. Әзірше оларды тауып алып, қатесін түзету үшін өз тәсілдеріңізді ұсыныңыздар.

`high()` функциясы нақты бір есепті ғана шешеді, сондықтан ол келесі шарттармен шектелген.

- Ол тек жиымдармен ғана жұмыс жасайды. Біз `vector` класының объектілері жиымда сақталынады деп есептейміз, мұнымен қоса мәліметтерді сақтаудың тізімдер және ассоциативті жиымдар тәрізді көптеген тәсілдері бар (20.4 және 21.6.1 бөлімдерін қ.).
- Оны тек ғана `vector` класының объектілеріне және `double` типіндегі жиымдарға ғана қолдануға болады, бірақта басқа типтегі векторлар мен жиым элементтеріне, мысалы, `vector<double*>` және `char[10]` сияқты типтерге қолдануға болмайды.
- Ол мәні ең үлкен элементті анықтайды, бірақ осы мәліметтермен басқа да көптеген қарапайым есептеулерді орындауға болады.

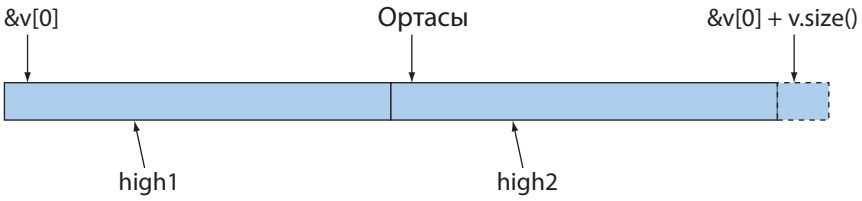
Біздегі мәліметтер топтамаларына әлдеқайда жоғары деңгейде атқарылатын ортақ есептеулерді орындауды қамтамасыз етіп көреміз.

Мынаған назар аударыңыз: ең үлкен элементті табу алгоритмін нұсқауыштар терминдерінде өрнектейтін болып шешкеннің өзінде-ақ, біз "кездейсоқ түрде" есептің шешімін олардың көптеген түрлеріне ортақ етіп (жалпылама етіп) жүзеге

асырдық: егер қаласақ, біз вектордың немесе жиымның ең үлкен элементін таба аламыз, оған қоса, сол вектордың немесе жиымның ішкі бөліктерінің де ең үлкен элементін анықтай аламыз. Мысал қарастырайық.

```
//. . .
vector<double>& v = *jill_data;
double* middle = &v[0]+v.size()/2;
double* high1 = high(&v[0], middle);
// бірінші жартысының максимумы
double* high2 = high(middle, &v[0]+v.size());
// екінші жартысының максимумы
// . . .
```

Мұнда **high1** нұсқауышы вектордың бірінші бөлігіндегі ең үлкен элементіне, ал **high2** нұсқауышы екінші бөлігіндегі ең үлкен элементіне сілтеме жасайды. Графикалық түрде оны төмендегідей етіп көрсетуге болады:



Біз **high()** функциясының аргументі ретінде нұсқауыштарды пайдаландық. Жады аймағын басқарудың осы механизмі өте төменгі деңгейге жатады және ол қателерге оңай бой алдырады.

Вектордағы ең үлкен элементті табу үшін көптеген программалаушылар төмендегіге ұқсас бір нәрсе жазуы мүмкін деп ойлаймыз:

```
double* find_highest(vector<double>& v)
{
      double h = -1;
      double* high = 0;
      for (int i=0; i<v.size(); ++i)
            if (h<v[i]) { high = &v[i]; h=v[i]; }
      return high;
}
```

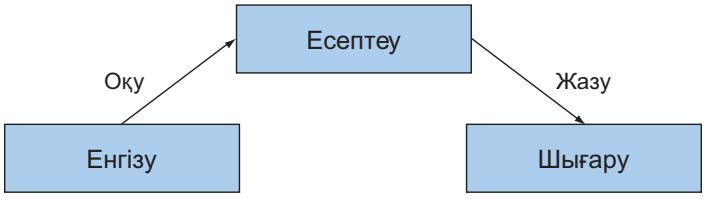
Дегенмен, бұл **high()** функциясын біз "кездейсоқ түрде" берген икемділікпен қамтамасыз етпейді, – біз вектордың ішкі бөлігіндегі ең үлкен элементті табу үшін **find_highest()** функциясын пайдалана алмаймыз. Шын мәнінде, біз

"нұсқауыштармен байланысқаннан" кейін, векторлармен де, жиымдармен де жұмыс істей алатын функцияны пайдалана отырып, тәжірибелік тиімділікке қол жеткіздік. Естеріңізде болсын: бірсыпыра есептерге ортақ шешім табу, яғни жалпылау тәсілі көптеген есептерді шығара алатын функцияларды пайдалануға жол ашады.

20.2 STL кітапханасының қағидалары

Элементтер тізбегі түрінде көрсетілген мәліметтермен жұмыстың негізін қамтамасыз ететін C++ тілінің стандарт кітапханасы STL деп аталады. Әдетте, осы қысқартылған атаудың (аббревиатураны) "шаблондардың стандартты кітапханасы" ("standard template library") деп мағынасын ашады. STL кітапханасы C++ тілі ISO стандартының бір бөлігі болып табылады. Ол контейнерлерден (`vector`, `list` және `map` кластары сияқты) және жалпыланған, яғни ортақтастырылған алгоритмдерден (`sort`, `find` және `accumulate` сияқты) тұрады. Сондықтан біз `vector` класы сияқты құралдарды стандартты кітапхананың да және STL кітапханасының да бөлігі деп айтуға құқығымыз бар. Стандартты кітапхананың басқа да құралдары – `ostream` ағымдары (10-тарауды қ.) және сөз тіркестерімен жұмыс жасайтын C тілі стиліндегі функциялар (Б.10.3 бөлімін қ.) тәрізділер STL кітапханасының ішкі бөлігі болып табылмайды. STL кітапханасын дұрыс бағалау және түсіну үшін біз алдымен мәліметтермен жұмыс жасау кезінде алып тасталуға тиіс кейбір мәселелерді қарастырамыз және оларды шешу идеяларын талдаймыз.

Есептеу аспектісінің екі негізгі түрі бар, олар: есептеулер және мәліметтер. Кейде біз есептеулерге көбірек назар аударамыз да, `if` нұсқаулары, циклдер, функциялар, қателерді өңдеу және т.с.с. туралы айтамыз. Басқа кездерде біз мәліметтерге назар аударып, жиымдар, векторлар, сөз тіркестері, файлдар және т.б. туралы айтатын боламыз. Дегенмен, тиімді жұмыс жасау үшін біз екі аспектіні де ескеруіміз қажет. Мәліметтердің үлкен көлемін талдаусыз, шолусыз (көзбен) және "қандай да бір қызықты" ізденусіз түсіну мүмкін емес. Және керісінше, біз есептеулерді өзімізге қалай қолайлы болса, солай орындасақ та болады, бірақ біздің есептеулерімізді нақты мысалдармен байланыстыратын қандай да бір мәліметтер алмайынша, мұндай көзқарас адамды өте зеріктіріп жібереді және "тым таза" болып та көрінуі мүмкін. Оған қоса, программаның есептейтін бөлігі оның "ақпараттық" бөлігімен өзара үйлесімді түрде әрекеттесуі қажет.



Осылайша мәліметтер туралы айта отырып, біз көптеген әртүрлі мәліметтерді тұспалдаймыз: ондаған **фигуралар**, температураның жүздеген мәндері, мыңдаған тіркеу жазбалары, миллиондаған нүктелер, миллиардтаған веб-парақтар және т.б.; басқаша айтқанда, біз мәліметтер контейнері, мәліметтер ағымы және т.с.с. туралы айтып отырмыз. Жекелеп айтқанда, біз комплекстік сан, температура туралы жазба немесе шеңбер сияқты шағын объектілерді ұсынатын мәліметтер топтамаларын қалай дұрыс таңдау керектігі жөніндегі сұрақтарды қарастырмаймыз. Мұндай типтер 9, 11 және 14-тарауларда сипатталған.

Көлемді мәліметтер топтамалары түсінігін бізге айқын көрсететін қарапайым мысалдар қарастырайық:

- Сөздіктегі сөздерді сұрыптау.
- Берілген аты бойынша телефон кітапшасындағы нөмірді табу.
- Ең үлкен температураны іздеу.
- 8800-ден асатын сандардың барлығын іздеу.
- Ең алғашқы кездесетін 17 санын табу.
- Құрылғылар нөмірі бойынша телеметрлік жазбаларды сұрыптау.
- Уақыт белгілері бойынша телеметрлік жазбаларды сұрыптау.
- "Petersen" сөз тіркесінен үлкен (ұзын) алғашқы мәнді іздеу.
- Ең үлкен көлемді іздеу.
- Екі тізбектің аралығындағы алғашқы сәйкессіздікті анықтау.
- Екі тізбек элементтерінің жұпталған элементтерінің өзара сәйкес көбейтіндісін есептеу.
- Айлық температураның ең үлкен мәнін табу.
- Сатылу жазбалары бойынша алғашқы үздік он сатушыны іздеу.
- Веб желісіндегі "Stroustrup" сөзінің қайталану санын анықтау.
- Элементтер қосындысын есептеу.

Осы есептердің әрқайсысын мәліметтерді сақтау тәсілін ескертпестен сипаттай алатынымызға назар аударыңыз. Әрине, біз қалайда тізімдермен, векторлармен, файлдармен, енгізу ағымдарымен және т.б. жұмыс істеуіміз керек шығар, бірақ, олармен не істейтінімізді айту үшін біз мәліметтердің қалай сақталатыны (және жиналатыны) туралы білуге міндетті емеспіз. Мәндердің немесе объектілердің типі (элементтер типі), осы мәндерге немесе объектілерге қол жеткізу тәсілі, сонымен қатар олармен біздің не істейтініміз маңызды болып табылады.

Есептердің осындай түрлері әмбебап сипатта болады. Әрине, біз осы есептерді қарапайым әрі тиімді түрде шығаратын кодты жазғымыз келеді. Осы кездерде, программалаушылар ретінде, біздің алдымызда, келесідей мәселелер тұрады:

- Мәліметтер типінің (мәліметтер түрлері) көптеген шексіз нұсқалары бар.
- Мәліметтер коллекциясын ұйымдастыру тәсілдерінің көптеген түрлері бар.
- Мәліметтер коллекциясы көмегімен шығарғымыз келетін көптеген есептер саны бар.

Осы мәселелердің әсерін азайту үшін, біз өзіміздің кодымызды көптеген есептерге қолдануға болатындай етіп жалпылағымыз келеді. Мұндайда ол мәліметтердің әртүрлі типтерімен жемісті жұмыс жасай алуы тиіс, мәліметтердің сақталу тәсілдерінің әртүрлі болғанын ескере отырып, оларды өңдеуге байланысты әртүрлі есептерді шығара алатындай болуы керек. Басқаша айтқанда, біз барлық нұсқаларды еңсере алатындай етіп, өзіміздің кодты жалпылама түрде жазғымыз келеді. Біз, әрине, әрбір есепті ең басынан бастап шығарғымыз келмейді; бұл уақытты босқа ысырап етеді.

Біздің осындай кодты жазуымыз барысында бізге қандай қолдау керектігін түсіну үшін, мәліметтермен не істей алатынымызды абстрактылы түрде қарастырып шығайық. Сонымен, келесі әрекеттерді жасауға болады:

- Мәліметтерді контейнерлерде жинау:
  - оларды, мысалы, `vector`, `list` кластары объектілерінде және жиымдарда жинау.
- Мәліметтерді ұйымдастыру:
  - баспаға шығару үшін;
  - оларға тез қол жеткізу үшін.
- Мәліметтерді іздеу:
  - индексі бойынша (мысалы, 42-элементті табу);
  - мәні бойынша (мысалы, "age" өрісіне 7 саны жазылған бірінші жазбаны табу);
  - қасиеттері бойынша (мысалы, "temperature" өрісінің мәні 32-ден үлкен және 100-ден кіші болатын барлық жазбалар).
- Контейнерді толықтыру:
  - мәліметтер қосу;
  - мәліметтерді өшіру;
  - сұрыптау (қандай да бір критерийлеріне сәйкес).
- Қарапайым математикалық операцияларды орындау (мысалы, барлық элементтерін 1,7-ге көбейту).

Біз осылардың барлығын контейнерлердің арасындағы айырмашылыққа, олардың типтері мен элементтеріне қол жеткізу тәсілдеріне қатысты ақпараттарға қарамай жасағымыз келеді. Егер осыны жасай алсақ, онда біз өзіміздің алдымызға қойған мақсатымызға жақындап, мәліметтердің үлкен көлемдерімен жұмыс жасайтын тиімді тәсілдерге қол жеткізер едік.

Алдыңғы тарауларда сипатталған программалау тәсілдері мен құралдарына қарап, біз қолданылатын мәліметтердің типінен тәуелсіз программалар жаза алатынымызды көреміз. Мұндай қорытынды келесідей фактілерге негізделген:

- `int` типін қолданудың `double` типін қолданудан айырмашылығы аз.
- `vector<int>` типін қолданудың `vector<string>` типін қолданудан айырмашылығы көп емес.
- `double` типіндегі жиым сандарын қолданудың `vector<double>` типін қолданудан айырмашылығы да көп емес.

Енді жұмысымызды, бізге алдыңғы есептерден айырмашылығы көп, мүлде басқа есеп шығару керек болғанда ғана жаңа код жазу қажет болатындай етіп ұйымдастыруға талпыну керек. Біз мәліметтерді сақтау тәсілі және олардың түсініктемесі (интерпретациясы) өзгерген сайын, программаларды қайталап жазбау үшін программалаудың әмбебап есептерін шығаратындай кодтың болуын қалар едік. Жекелеп айтқанда, келесідей шарттардың орындалуы жүзеге асырылуы тиіс:

- `vector` класының объектісіндегі мәндерді іздеу мен жиым ішіндегі мәндерді іздеудің айыпмашылығы болмауы керек.
- Регистрді есепке алмай, `string` класының объектісін іздеу әрекетінің төменгі және жоғарғы регистрлерді есепке ала отырып, string класының объектісін іздеуден айырмашылығы болмауы тиіс.
- Дәлме-дәл мәндері бар тәжірибелік (эксперименттік) мәліметтердің графикалық бейнесінің дөңгелектенген мәндері бар эксперименттік мәліметтерден айырмасы болмауы керек.
- Файлды көшіру мен `векторды` көшірудің айырмасы болмауы тиіс.

Айтылғандарды ескере отырып, келесі шарттарды қанағаттандыратын кодты жазған дұрыс болады:

- оны жеңіл оқуға;
- оңай толықтыруға болады;
- ол жүйелік сипатта болады;
- ол қысқаша жазылады;
- жылдам жұмыс істейді.

Программалаушының жұмыс көлемін азайту үшін, біз келесі міндеттерді атқаруымыз қажет.

- Мәліметтерге бірыңғай түрде қол жеткізу:
  - мәліметтерді сақтау тәсілінен тәуелсіз болу;
  - мәліметтер типінен тәуелсіз болу.
- Тип тұрғысынан қарағанда, қауіпсіз болатын мәліметтерге қол жеткізу;
- мәліметтер бойынша жеңіл орын ауыстыру;
- мәліметтерді шағын, әрі ыңғайлы түрде сақтау.
- Жұмыс жылдамдығы:
  - мәліметтерді іздеу;
  - мәліметтер қосу;
  - мәліметтерді жою.
- Кең тараған көптеген алгоритмдердің стандартты нұсқалары:
  - `copy`, `find`, `search`, `sort`, `sum`... сияқты.

STL кітапханасы тек бұл мүмкіндіктерді ғана қамтамасыз етпейді. Бұл кітапхананы біз пайдалы құралдар топтамалары бар болғандықтан ғана емес, сонымен қоса бейімделгіш және тиімді мысал ретінде де қарастырамыз. STL кітапханасын Алекс Степанов (Alex Stepanov) әртүрлі мәліметтер құрылымымен жұмыс жасайтын әмбебап, дұрыс және тиімді алгоритмдерге арналған база жасау үшін жасап шығарды. Оның мақсаты математиканың қарапайымдылығы, әмбебаптығы және әсемдігі болды.

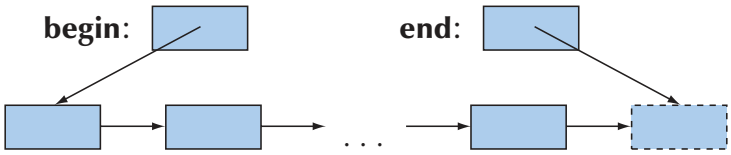
Егер де біздің қолымызда айқын идеялары мен принциптері бар кітапханалар болмағанда, онда әр программалаушы өз кодын жазу кезінде жақсы болып көрінетін негізгі тілдік конструкциялар мен идеяларға сүйеніп, әрбір программаны өзі құрастырар еді. Бұл үшін көптеген артық жұмыстар орындауға тура келеді. Осының нәтижесінде принципсіз түсінбестік те болып тұрады; көбінесе, мұндай программаларды олардың авторынан басқа ешкім түсінбейді және осындай программаларды басқа контексте (басқа кодтар ішінде) қолдану күмән тудырады.

Сонымен, себептері мен мақсаттарын қарастырып, STL кітапханасының негізгі анықтамаларын сипаттауға көшеміз, содан кейін мәліметтерді өңдеуге арналған анағұрлым жетілдірілген кодтарды қарапайым түрде құру үшін соларды қолдану мысалдарын үйренеміз.

20.3 Тізбектер мен итераторлар

STL кітапханасының негізгі түсінігі тізбек болып табылады. Осы кітапхананың авторларының ойы бойынша мәліметтердің кез келген жиынтығы (коллекциясы) тізбектен тұрады. Тізбектің басы мен соңы болады. Біз тізбек бойынша оның басынан соңына дейін, қажеттілігіне қарай, элементтер мәнін жаза немесе оқи отырып орнын ауыстыра аламыз. Тізбектің басы мен соңы итераторлар жұбымен белгіленеді. *Итератор* (iterator) – бұл тізбек элементтерін сәйкестендіріп анықтайтын объект.

Тізбекті келесі түрде көрсетуге болады:



Мұндағы, **begin** және **end** – итераторлар; олар тізбектің басы мен соңын анықтайды. STL кітапханасында тізбекті "жартылай ашық" ("half-open") деп айтады; басқаша айтқанда, **begin** итераторымен белгіленетін элемент тізбектің бөлігі болып табылады, ал **end** итераторы тізбектің соңынан кейінгі келесі ұяшыққа сілтеме жасайды. Әдетте, осындай тізбектер (диапазондар) **[begin:end)** болып белгіленеді. Бір элементтен екіншісіне бағытталған тілсызық бір элементке итератор болатын болса, онда біз оның келесісіне де итератор ала алатынымызды көрсетеді.

Итератор дегеніміз не? Бұл өте абстрактілі түсінік:

- Итератор тізбек элементін (немесе соңғы элементтен кейін тұрған ұяшықты) нұсқап тұрады, яғни оған сілтеме жасайды.
- Екі итераторды **==** және **!=** операторлары көмегімен салыстыруға болады.
- Итератор орнатылған элементтің мәнін ***** унарлық операторы ("атаусыз ету") арқылы алуға болады.
- Келесі элементке орнатылған итераторды **++** операторын қолдану арқылы ала аламыз.

Мысалы, **p** және **q** – бір тізбектің элементтеріне орнатылған итераторлар болсын делік:

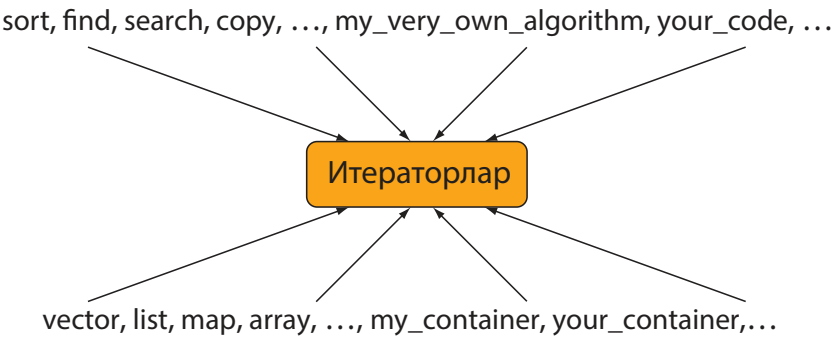
Стандартты итераторлармен орындалатын негізгі операциялар	
`p==q`	егер екі итератор да (`p` мен `q`) бір элементке сілтеме жасап тұрса немесе ол екеуі де соңғы элементтен кейінгі ұяшыққа орнатылған жағдайда ғана, ол `true` мәніне тең болады
`p!=q`	`!(p==q)`
`*p`	`p` итераторы орнатылған элементке сілтеме жасайды
`*p=val`	`p` итераторы сілтеме жасап тұрған элементке `val` мәнін меншіктейді
`val=*p`	`p` итераторы сілтеме жасап тұрған элемент мәнін `val` айнымалысына меншіктейді
`++p`	`p` итераторын тізбектің келесі элементіне немесе тізбектің соңғы элементінен кейінгі элементке орнатады

Әрине, итератор идеясы нұсқауыш идеясымен байланысты шығар (17.4 бөлімді қ.). Негізінде жиым элементіне нұсқауыш итератор болып табылады. Дегенмен, көптеген итераторлар жай ғана нұсқауыш болып табылмайды; мысалы, біз `[begin:end)` тізбегі шегінен тыс аймаққа сілтеме жасауға талпыну кезінде аластама жасайтын және берілген (мүмкін болатын) тізбектің шегінен шығып кетуді тексеретін итераторды анықтай алар едік немесе `end` итераторын атаусыз етуімізге болар еді. Итератор нақты тип емес, абстрактылы түсінік ретінде өте жоғары деңгейдегі икемділік пен әмбебаптықты қамтамасыз ете алады екен. Осы және келесі тарауларда оған тағы да бірнеше мысалдар келтіреміз.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

`[f1:e1)` тізбегімен анықталған `int` типіндегі сандар жиымы элементтерін басқа бір `[f2:f2+(e1-f1))` тізбегіне көшіретін `void copy(int* f1, int* e1, int* f2)` функциясын жазыңыз. Тек жоғарыда айтылған итераторларды ғана (индекстеу емес) қолданыңыз.

Итераторлар біздің кодымыз (алгоритмдер) бен мәліметтеріміздің арасындағы байланыс құралы ретінде пайдаланылады. Код авторы итераторлардың бар екендігін біледі (бірақ олардың мәліметтерді қалай пайдаланатынын білмейді), ал мәліметтерді беруші мәліметтердің сақталу механизмінің нақтылықтарын барлық пайдаланушыларға жария етпей итераторларды көрсетеді. Нәтижесінде бір-бірінен белгілі бір дәрежеде тәуелсіз алгоритмдер мен контейнерлерді аламыз. Алекс Степановтың сөздерінен үзінді келтірейік: "STL кітапханасының алгоритмдері мен контейнерлері бір-бірі туралы ештеңе білмегендіктен, бір-бірімен өте жақсы жұмыс істей алады". Мұның орнына алгоритмдер де, контейнерлер де итераторлар жұбымен анықталған тізбектер туралы біледі.



Басқаша айтқанда, код авторы мәліметтерді сақтаудың әртүрлі тәсілдері мен оларға қалай қол жеткізуді қамтамасыз ету туралы білуге міндетті емес; итераторлар туралы білгені жеткілікті. Және керісінше, егер мәліметтер беруші бұдан кейін әртүрлі пайдаланушылардың көптеген санына қызмет көрсету үшін код жазуға міндетті емес болса, оған мәліметтер үшін итераторды жүзеге асыру жеткілікті болып табылады. Базалық деңгейде итератор тек қана *****, **++**, **==** және **!=** операторларымен анықталады. Бұл оның қарапайымдылығы мен тез жұмыс істеуін қамтамасыз етеді.

STL кітапханасы итераторлармен байланысқан он шақты контейнерлерден және 60 алгоритмнен тұрады (21-тарауды қ.). Оған қоса, көптеген ұйымдар мен жеке тұлғалар STL кітапханасының стилінде контейнерлер мен алгоритмдер жасайды. Мүмкін, STL кітапханасы қазіргі уақыттағы жалпылама программалаудың ең көп пайдаланылатын және кең таралған мысалы болып табылатын шығар (19.3.2 бөлімін қ.). Егер сіз оның негіздері мен бірнеше мысалдарын білсеңіз, онда басқаларын да пайдалана аласыз.

20.3.1 Мысалдарға ораламыз

STL тізбектерінің көмегімен "ең үлкен элементті табу" есебін қалай шығаруға болатынын көрейік.

```
template<class Iterator>
Iterator high(Iterator first, Iterator last)
// итераторды [first:last) диапазонындағы
// ең үлкен элементке қайтарады
{
      Iterator high = first;
      for (Iterator p = first; p!=last; ++p)
            if (*high<*p) high = p;
      return high;
}
```

Осы уақытқа дейін ең үлкен элементті сақтау үшін пайдаланылған жергілікті айнымалы h-ты алып тастағанымызға назар аударыңыз. Егер сізге тізбек элементтерінің нақты типтері белгісіз болса, онда -1 инициалдауы кездейсоқ және таңғаларлық болып көрінеді. Ол, шынымен де кез келген сан бола береді және таңғалуға да болады! Сонымен қатар, мұндай инициалдау қате болып табылады: біздің мысалдағы 1 санының өзін-өзі ақтайтын себебі – ол теріс жылдамдықтың болмайтынынан шығады. Біз -1 сияқты "сиқырлы тұрақтылардың" кодты сүйемелдеуге кедергісін тигізетінін білеміз (4.3.1, 7.6.1, 10.11.1 және т.б. бөлімдерді қ.). Біз мұндай тұрақтылар функцияның тиімділігін төмендетіп, шешімнің толық еместігін білдіретінін көреміз; басқаша айтқанда, "сиқырлы тұрақтылар" көбінесе, олақтықтың куәсі болып саналады.

high() шаблондық функциясын < операциясының көмегімен салыстыруға болатын элементтердің кез келген типіне қолдануға болады. Мысалы, біз high() функциясын vector<string> контейнеріндегі соңғы жолды лексикографикалық түрде іздеу үшін қолдануға болар еді (7- жаттығуды қ.).

high() шаблондық функциясын итераторлардың жұбымен анықталған кез келген тізбекке қолдануға болады. Мысалы, біз өзіміздің программамызды дәлме-дәл түрде нақтылап қайта көрсете аламыз.

```
double* get_from_jack(int* count);
// Джек double типіндегі сандарды жиымға енгізеді де,
// *count айнымалысындағы элементтер санын қайтарады
vector<double>* get_from_jill();
// Джилл векторды толтырады

void fct()
{
    int jack_count =0;
    double* jack_data = get_from_jack(&jack_count);
    vector<double>* jill_data = get_from_jill();

    double*jack_high=high(jack_data,jack_data+jack_count);
    vector<double>& v=*jill_data;
    double* jill_high = high(&v[0],&v[0]+v.size());
    cout << "Джилл максимумы " << *jill_high
    << "; Джектің максимумы " << *jack_high;
    // . . .
    delete[] jack_data;
    delete jill_data;
}
```

Мұнда high() функциясын екі рет шақырғанда да, double* типі аргументтің шаблондық типі болып табылады. Бұның біздің алдыңғы шешімімізден

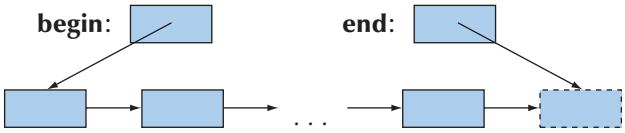
ешқандай да айырмашылығы жоқ. Дәлірек айтқанда, бұл кодтардың жалпылану деңгейінің айырмашылығы елеулі болғанмен, олардың орындалу кодтарының бір-бірінен ешқандай айырмасы жоқ. `high()` функциясының шаблондық нұсқасын **итераторлар** жұбымен анықталған тізбектердің кез келген түріне қолдануға болады. Осы принциптерді іске асыратын STL кітапханасының қағидалары мен пайдалы стандартты алгоритмдерді толығырақ ұғыну үшін және де күрделі кодтарды жазудан қашығырақ болу үшін мәліметтер коллекциясын (жиынтығын) сақтаудың бірнеше тәсілдерін қарастырайық.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

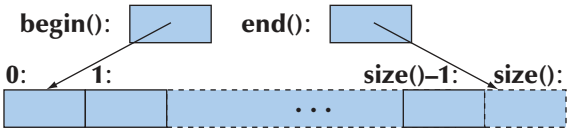
Бұл программада тағы да үп-үлкен қате жіберілген. Оны тауып алып, түзетіңіз және осындай туындаған мәселелерді алып тастайтын әмбебап тәсілдер ұсыныңыз.

20.4. Байланысқан тізімдер

Тізбектің графикалық бейнелену түрін тағы да бір қарастырайық:



Оны компьютер жадында сақталатын **вектор** бейнесімен салыстырамыз:

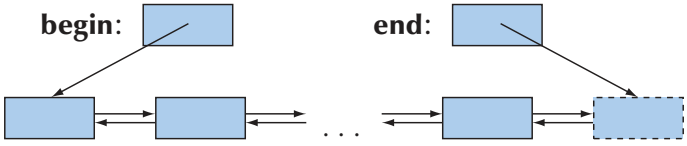


Шын мәнісінде, `0` индексі мен `v.begin()` итераторы бір элементті, ал `v.size()` функциясы ең соңғы элементтен кейін орналасқан элементті білдіреді, оны `v.end()` итераторы арқылы да көрсетуге болады.

Вектордағы элементтер компьютер жадында тізбектеле орналасқан. STL кітапханасындағы тізбек ұғымы оны қажет етпейді. Бұл көптеген алгоритмдерге бұрынғы элементтер арасына элементтер енгізіп, олардың орнын ауыстырмай-ақ қоя беруге мүмкіндік береді. Тізбектің абстрактылық түсінігін графикалық түрде көрсетуде бұрынғы элементтерді ығыстырмай-ақ жаңа элементтер енгізу (және

жою) мүмкіндігі бар деп есептеледі. STL кітапханасындағы итераторлар түсінігі осы тұжырымдаманы (концепцияны) сүйемелдейді.

STL кітапханасындағы тізбектер диаграммасына басқаларынан гөрі дәлірек сәйкес келетін мәліметтер құрылымын *байланысқан тізімдер* (linked list) деп атайды. Абстрактілі модельдегі бағыттауыш тілсызықтар көбінесе нұсқауыштар түрінде жүзеге асады. Байланысқан тізімнің элементі – бұл бір элементтен және бір немесе бірнеше нұсқауыштардан тұратын түйінннің бөлігі. Түйіні бір ғана (келесі түйінге) нұсқауыштан тұратын байланысқан тізім *бірбайланысты тізім* (singly-linked list) деп аталады, ал бір түйіні алдыңғы және келесі түйінге сілтеме жасап тұратын тізім – *екібайланысты тізім* (doubly-linked list) болып табылады. Біз C++ тілінің стандартты кітапханасындағы **list** деп аталатын екібайланысты тізімнің жасалуын сұлба (схема) түрінде қарастырамыз. Графикалық түрде тізімді келесідей түрде көрсетуге болады:

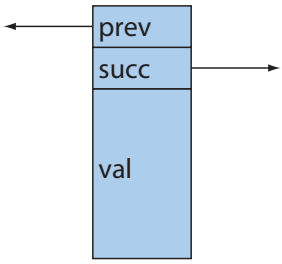


Код түрінде ол былай болады:

```
template<class Elem> struct Link {
      Link* prev; // алдыңғы түйін
      Link* succ; // соңғы түйін
Elem val; // мәні
};

template<class Elem> struct list {
      Link<Elem>* first;
      Link<Elem>* last;
      // ең соңғы түйіннен кейін орналасқан түйін
};
```

Link класының сұлбасы төменде көрсетілген.



Байланысқан тізімдерді іске асыратын және көрсететін тәсілдер көп. Стандартты кітапханада жүзеге асырылған тізімнің сипаттамасы Б қосымшасында көрсетілген. Мұнда біз тек тізімдердің негізгі қасиеттерін – бұрыннан бар элементтеріне тиіспей, элементтерді кірістіру және жою мүмкіндігін, сонымен қоса итератордың көмегімен тізімде қалай орын ауыстыруға болатындығын көрсетеміз де, оны пайдалану мысалын келтіреміз.

Біз сіздерге тізімдер туралы ойлана отырып, өзіңіз қарастыратын операцияларды бейнелейтін диаграммалар суретін салуды табандылықпен атқаруды ұсынамыз. Байланысқан тізімдермен іс-әрекеттер орындау – бұл бір суреттің мыңдаған сөздерді ауыстыра алатынын білдіретін тақырып.

20.4.1 Тізімдермен орындалатын операциялар

Тізім үшін қандай операциялар қажет?

- Векторларға қолданылатын операцияларға тепе-тең операциялар (құру, өлшемін анықтау және т.б.), индекстеуден басқа.
- Кірістіру (элементті қосу) және өшіру (элементті жою).
- Элементтерге сілтеме жасау және тізім бойынша орын ауыстыру үшін пайдалануға болатын нәрсе: итератор.

STL кітапханасында итератордың типі оның өз класының мүшесі болып табылады, сондықтан біз де солай жасаймыз:

```
template<class Elem> class list {
//  бейнелеу және жүзеге асыру нақтылықтары
public:
    class iterator;   // типі – класс мүшесі: iterator

    iterator begin();
    // бірінші элементке сілтеме жасайтын итератор
    iterator end( );
    // соңғы элементке сілтеме жасайтын итератор

    iterator insert(iterator p, const Elem& v);
    // v-ны тізімге p итераторы орнатылған
    // элементтен кейін кірістіру
    iterator erase(iterator p);
    // p итераторы орнатылған элементті тізімнен өшіру

    void pushback(const Elem& v);
    // v-ны тізім соңына енгізу
```

```
    void push_front(const Elem& v);
    // v-ны тізім басына енгізу
    void pop_front(); // бірінші элементті жою
    void pop_back();  // соңғы элементті жою

    Elemfc front();   // бірінші  элемент
    Elem& back();     // соңғы элемент
    //…
};
```

Біздің **vector** класымыз стандартты вектордың толық нұсқасымен сәйкес келмегені сияқты **list** класы да – стандартты тізімнің толық анықтамасы емес. Бұл анықтамада бәрі дұрыс; бірақ ол толық емес. "Біздің" **list** класымыздың мақсаты – байланысқан тізімдердің құрылымын түсіндіру, олардың жүзеге асырылғанын көрсету және олардың негізгі мүмкіндіктерін қолданудың тәсілдерін көрсету. Сарапшыларға (эксперттерге) арналған бұдан толығырақ ақпарат B қосымшасында және де C++ тілі туралы кітаптарда келтірілген.

STL кітапханасындағы **list** класын анықтауда итератор басты рөл атқарады. Итераторлар элементтерді кірістіру немесе жою орындарын белгілеу үшін қолданылады. Сонымен қоса, оларды индекстеу операторының орнына тізім бойынша "навигациялау" үшін пайдаланады. Итераторларды осындай түрде пайдалану жиымдар және векторлар бойынша орын ауыстыру (жылжу) кезінде нұсқауыштарды қолдануға ұқсайды, олар 20.1 және 20.3.1 бөлімдерде сипатталған. Стандартты алгоритмдерде итераторлардың осындай түрі негізгі болып табылады (21.1-21.3 бөлімдер).

list класында неліктен индекстеу қолданылмайды? Біз түйіндерді де индекстер едік, бірақ бұл операция өте баяу орындалады: **lst[1000]** элементіне жету үшін бізге бірінші элементтен бастап, біртіндеп барлық элементтерден өтіп, сол **1000** нөмірлі элементке жеткенге дейін жүру керек еді. Егер сіз осылай жасағыңыз келсе, онда ол операцияны өздеріңіз іске асырыңыздар (немесе **advance();** алгоритмін пайдаланыңыз, 20.6.2 бөлімін қ.). Осы себептерге байланысты **list** стандартты класында индекстеу операциясы жоқ.

Біз тізім үшін итератор типін класс мүшесі (ішкі класс) етіп жасадық, өйткені оны ауқымды етудің ешқандай себебі жоқ. Ол тек тізімдерде ғана қолданылады. Оған қоса, бұл бізге контейнердегі әрбір типті **iterator** атымен атауға мүмкіндік береді. Стандартты кітапханада **list<T>::iterator, vector<T>::iterator, map<K,V>::iterator** және т.б. бар.

20.4.2 Итерация

Тізім итераторы ***, ++, ==** және **!=** операцияларын орындауды қамтамасыз етуі қажет. Өйткені стандартты тізім екібайланысты болғандықтан, онда тізімнің басына шығаратын артқа орын ауыстыру – операциясы бар:

```
template<class Elem> class list<Elem>::iterator {
    Link<Elem>* curr;   // ағымдағы түйін
public:
    iterator(Link* p) :curr(p) { }

    iterator&operator++(){curr=curr->succ;return*this;}
    //алға
    iterator&operator--(){curr=curr->prev;return*this;}
    //артқа
    Elem& operator*() { return curr->val; }
    // мәнін алу (атаусыз ету)

    booloperator==(constiterator&b)const{returncurr==b.curr;}
    booloperator!=(constiterator&b)const{returncurr!=b.curr;}
};
```

Бұл функциялар қысқа, қарапайым әрі тиімді, өйткені оларда цикл жоқ, күрделі өрнектер және күмән келтіретін функцияларды шақыру да жоқ. Егер осылай жүзеге асыру сізге түсінікті болмаса, онда жоғарыда келтірілген диаграммаларға қараңыз. Осы тізімнің итераторы қажетті операциялары бар түйінге нұсқауышты көрсетеді. `list<Elem>::iterator` класы үшін кодты жүзеге асырудың векторлар мен жиымдар үшін итератор ретінде қолданылған қарапайым нұсқауыштан айырмашылығы көп болғанына қарамастан, олардың семантикасы бірдей. Негізінде, тізім итераторы түйінге жасалған нұсқауыш үшін `++`, `--`, `*`, `==` және `!=` тәрізді ыңғайлы операцияларды орындауды қамтамасыз етеді.

`high()` функциясына тағы да назар аударайық:

```
template<class Iterator >
Iterator high(Iterator first. Iterator last)
// итераторды [first,last) диапазонындағы ең үлкен
// элементке қайтарады
{
     Iterator high = first;
     for (Iterator p = first; p!=last; ++p)
     if (*high<*p) high = p;
     return high;
}
```

Біз мұны `list` класының объектісіне қолдана аламыз:

```
void f()
{
  list<int> lst;
```

```
    int x;
    while (cin >> x) lst.push_front(x);

    list<int>::iterator p = high (lst .begin (),lst.end());
    cout << "ең үлкен мәні = " << *p << endl;
}
```

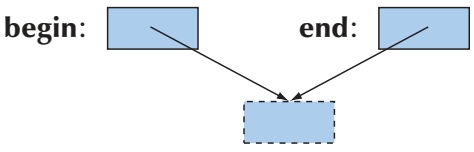
Мұнда `iterator argument` класы аргументінің мәні `list<int>::iterator` класы болып табылады, ал `++`, `*` және `!=` операцияларын жүзеге асыру, олардың мағынасы өзгеріссіз қалғанмен, жиымнан тіпті басқаша болады. `high()` шаблондық функциясы бұрынғыдай мәліметтермен бойынша (мұнда `list` класының объектісі бойынша) біртіндеп жылжи отырып, ең үлкен мәнін табады. Біз тізімнің кез келген жеріне элементтерді кірістіріп қоя аламыз, мұнда біз қарапайым түрде көрсету (иллюстрация ретінде) мақсатында `push_front()` функциясын элементтерді тізімнің басына қосу үшін пайдаландық. Осындай жетістікпен біз `vector` класының объектілері үшін жасағанымыздай етіп `push_back()` функциясын да қолдануымызға болар еді.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Стандартты `vector` класында `push_front()` функциясы жоқ. Неліктен? **vector** класы үшін `push_front()` функциясын жүзеге асырып, оны `push_back()` функциясымен салыстырыңыз.

Сонымен, мынаны сұрайтын уақыт жетті: "Егер `list` класының объектісі бос болса не болар еді? " Басқаша айтқанда, "егер `lst.begin()==lst.end()` болса ше?" Мұндай жағдайда `*p` нұсқауының орындалуы соңғы элементтен кейін тұрған элементті, яғни `lst.end()` элементін атаусыз етуге мүмкіндік береді. Бұл апаттық жағдай! Немесе, одан да сорақысы, мұның нәтижесінде дұрыс жауапты өзгертіп жіберетін кездейсоқ шама алуға болады.

Сұрақтың соңғы айтылымы тікелей көмек бере алатын мәліметтен тұрады: `begin()` және `end()` итераторларын салыстыра отырып, біз тізімнің бос екендігін тексере аламыз, негізінде біз тізбектің басы мен соңын салыстыра отырып, оның бос болатынын тексеріп біле аламыз:



`end` итераторының соңғы элементке емес, соңғы элементтен кейін тұрған элементке орнатылатынының негізгі себебі бар: бос тізбек – ерекше жағдай емес.

Біз ерекше жағдайларды жақсы көрмейміз, сондықтан анықтама бойынша, олардың әрқайсысы үшін жеке-жеке код жазуға тура келеді.

Біздің мысалды төмендегідей түрде жасауға болады:

```
list<int>::iterator p = high(lst.begin(),lst.end());
if (p==lst.end()) // біз соңына жеттік пе?
     cout << "Тізім бос";
else
     cout << "ең үлкен мәні = " << *p << endl;
```

STL кітапханасының алгоритмдерімен жұмыс жасай отырып, біз бұл тексеруді жүйелі түрде пайдаланамыз. Стандартты кітапханада тізім қарастырылғандықтан, оның жүзеге асу нақтылықтарына тереңдемейміз. Оның орнына осы тізімдердің немен қолайлы екенін қысқаша көрсетеміз (егер сізді тізімдерді жүзеге асыру нақтылықтары қызықтырса, онда 12-14-жаттығуларды орындап көріңіз).

20.5 `vector` класының тағы бір жалпыламасы

20.3 және 20.4 бөлімдерде көрсетілген мысалдардан шығатыны, стандартты вектордың iterator класы болып табылатын класс мүшесі бар, сондай-ақ, онда `begin()` және `end()` функция-мүшелері (`std::list` класы сияқты) де бар. Бірақ біз оларды 19-тараудағы өзіміз жасаған vector класында көрсетпедік. 20.3 бөлімінде сипатталған жалпылама программалауда әртүрлі контейнерлер бір-бірін азды-көпті түрде алмастыра отырып, нелерге байланысты қолданылуы мүмкін? Алдымен шешу сұлбасын сипаттайық та (қарапайым болуы үшін жады аймағын бөлуді ескеремейміз), оны содан кейін түсіндірейік.

```
template<class T> class vector {
public:
     typedef unsigned long size_type;
     typedef T value_type;
     typedef T* iterator;
     typedef const T* const_iterator;
     // . . .
     iterator begin();
     const_iterator begin() const;
     iterator end();
     const_iterator end() const;

     size_type size O;
     // . . .
};
```

Typedef операторы типтің синонимін құрады; басқаша айтқанда, біздің **vector** класымыз үшін **iterator** аты – бұл синоним, яғни біз итератор ретінде қолданамыз деп шешкен типтің басқа аты: **T***. Енді **vector** класының **v** объектісі үшін келесі нұсқауларды жазуға болады:

```
vector<int>::iterator p = find(v.begin(),v.end(),32);
```

және

```
for(vector<int>::size_type i=0;i<v.size();++i)
                    cout << v[i] << '\n';
```

Осы нұсқауларды жазу үшін, шын мәнінде, бізге қандай типтер **iterator** және **size_type** деп аталатынын білу қажет емес. Жекелеп алғанда, **iterator** және **size_type** типтері арқылы жоғарыда көрсетілген кодта біз векторлармен жұмыс жасаймыз, **size_type** типі – бұл **unsigned long** емес (көптеген құрамдас жүйелердің процессорындағы сияқты), ал **iterator** типі – жай ғана нұсқауыш емес, класс болып табылады (C++ тілінің көптеген кең таралған нұсқаларындағы сияқты).

Стандартта **List** класы және басқа да стандартты контейнерлер осыған ұқсас түрде анықталған. Мысал қарастырайық.:

```
template<class Elem> class list {
public:
      class Link;
      typedef unsigned long size_type;
      typedef Elem value_type;
      class iterator;       // 20.4.2 бөлімін қараңыз
      class const_iterator; // iterator сияқты, бірақ,
      // элементтердің өзгеруін де жүзеге асыра алады
      // . . .
      iterator begin();
      const_iterator begin() const;
      iterator end();
      const iterator end () const;

      size_type size();
      // . . .
};
```

Сонымен осылайша, программаның нені пайдаланатынын – **list** класын немесе **vector** класын – ойламай-ақ кодты жазуға болады екен. Барлық стандартты алгоритмдер **iterator** және **size_type** сияқты типтердің атымен көрсетілген терминдермен анықталған, сондықтан олар контейнерлердің жүзеге асырылуына немесе олардың түріне тәуелді болмайды (осы туралы толығырақ – 21-тарауда).

20.6 Қарапайым мәтіндік редактор мысалы

Тізімнің маңызды қасиеті болып тізімнің басқа элементтерінің орнын ауыстырмай-ақ элементтерді кірістіру немесе алып тастау есептеледі. Осы әрекетті көрсететін қарапайым мысалды зерттейік. Қарапайым мәтіндік редактордағы мәтіндік құжаттың символдарды қалай бейнелене алатынын қарастырамыз. Мұндай бейнелену түріндегі құжаттармен орындалатын операциялар қарапайым және мүмкіндігінше тиімді болуы қажет.

Қандай операциялар? Құжат компьютердің негізгі жадында сақталынады делік. Сол себепті кез келген қолайлы бейнелеу, яғни көрсету түрін таңдап алып, оны файлда сақтағымыз келген байттар ағынына айналдырамыз. Осылай етіп біз файлдан байттар ағынын оқып, оларды компьютер жадындағы сәйкес көрсетілімдерге айналдырамыз. Осы сұрақты шешіп алған соң, компьютер жадындағы құжаттың өзіне сәйкес келетін көрсетіліміне тоқталамыз. Бұл көрсетілім:

- Енгізу ағынынан түсетін байттар ағынынан құжат жасау.
- Бір немесе бірнеше символдарды кірістіріп қою.
- Бір немесе бірнеше символдарды жою.
- Сөз тіркесін іздеу.
- Байттар ағынын файлға немесе экранға шығару үшін генерациялау сияқты бес операцияны сүйемелдеуі керек.

Қарапайым көрсетілім ретінде `vector<char>` класын таңдауға болады. Бірақ векторға символды қосу немесе жою үшін бізге құжаттағы барлық келесі символдарды ары (бері) қарай ығыстыру керек болар еді. Мысал қарастырайық:

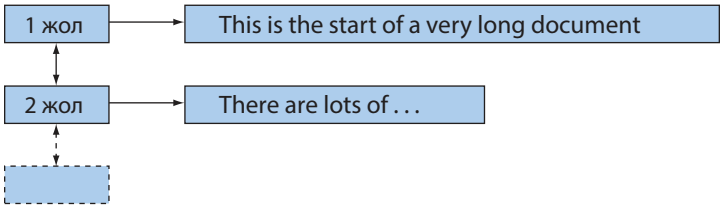
```
This is he start of a very long document.
There are lots of ...
```

Біз жетпей тұрған `t` символын қосып, келесі мәтінді алар едік:

```
This is the start of a very long document.
There are lots of ...
```

Дегенмен, егер осы символдар `vector<char>` класының жеке объектісінде сақталса, біз барлық символдарды `h` әріпінен бастап бір позицияға оңға қарай жылжытуымыз керек еді. Ол үшін көп символдарды көшіру керек болады. Негізінде, 70 мың символдан тұратын құжат үшін (босорындар санын ескеріп есептегенде осы тарау сияқты), символды кірістіріп қою немесе өшіру кезінде бізге орташа шамамен 35 мың символды жылжыту қажет. Нәтижесінде пайдаланушылар үшін уақыттың кідірісі байқалып, өкінішті болар еді. Осының салдарынан біз өз көрсетілімімізді "порцияларға" (бөліктерге) бөліп, үлкен символдар жиымын

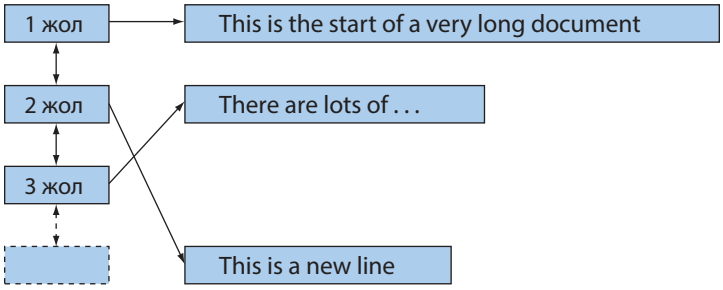
ығыстырмай, құжаттың ішкі шағын бөліктерін ғана қарастырамыз. Біз құжатты сөз тіркестері тізімі ретінде `list<Line>` класы арқылы көрсетеміз, мұндағы `Line` шаблондық параметрі – бұл `vector<char>` класы. Мысал қарастырайық:



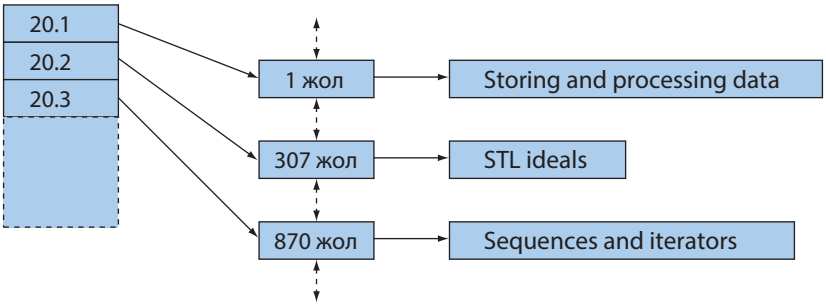
Енді `t` символын кірістіріп қою үшін осы сөз тіркесінің ғана қалған символдарын жылжыту жеткілікті. Оған қоса айтарымыз, қажет болған жағдайда қандай да бір символдарды жылжытпай-ақ жаңа жолды да қоса аламыз. Мысал үшін "`This is a new line.`" жолын "`document.`" сөзінен кейін кірістіріп қоюды қарастырайық:

```
This is the start of a very long document.
This is a new line.
There are lots of ...
```

Осы үшін бізге керегі – жаңа жолды бұрынғылардың ортасына қосу:



Жаңа түйіндерді бұрыннан бар түйіндердің орнын ауыстырмай-ақ кірістіріп қою мүмкіндігі былай түсіндіріледі: осы түйіндерге сілтеме жасайтын итераторларды немесе осы түйіндердің объектілеріне орнатылған нұсқауыштарды (немесе сілтемелерді) пайдаланамыз. Осындай итераторлар мен нұсқауыштар жолдарды кірістіріп қоюға немесе алып тастауға тәуелді болмайды. Мысалы, итераторлар сақталатын мәтіндік процессорда `vector<list<Line>::iterator>` класының объектісі қолданылуы мүмкін, онда `Document` класының ағымдағы объектісінен алынған әр тақырыптың немесе тақырыпшаның (ішкі тақырыптың) басына орнатылған итераторлар сақталады:



Біз "paragraph 20.3" арқылы орнатылған итератордың біртұтастығын бұзбай, "paragraph 20.2" арқылы жаңа жолдар қоса аламыз.

Қорытындылай келе айтарымыз, жолдар тізімімен қатар барлық символдардың векторын да пайдаланудың логикалық және де тәжірибелік (практикалық) себептері бар. Дегенмен, осы себептердің маңызды болатын кездері өте сирек болатындықтан, "`vector` класын үнсіз келісім бойынша пайдалану" ережесі бұрынғысынша өз күшінде қала беретініне көңіл аудару керек. Егер сіз өз мәліметтеріңізді тек тізім түрінде ғана көрсетсеңіз де, `vector` класына қарағанда, `list` класына басымдылық беру үшін ерекше себеп керек! (20.7 бөлімін қ.). Тізім – бұл логикалық түсінік, біздің программамызда оны `list` класының да (байланысқан тізім) және `vector` класының да көмегімен көрсетуге болады. STL кітапханасында тізім туралы біздің тұрмыстық көрсетілімнің (мысалы, жұмыстар, істер, тауарлар тізімі немесе кесте) ең жақын аналогы болып тізбек саналады, ал көптеген тізбектерді `vector` класы арқылы көрсеткен дұрыс болып табылады.

20.6.1 Жолдар

Біздің құжаттағы жолдар дегеніміз не екенін қалай шешуге болады? Мұның қалыптағы үш баламасы бар:

1. Енгізу жолындағы жаңа жолдар (мысалы, `'\n'`) индикаторын қолдану.
2. Құжатты белгілі бір тәртіппен бөлу және қарапайым пунктуацияларды пайдалану (мысалы, .).
3. Ұзындығы берілген бір мәннен асатын (мысалы, 50 символ) жолды екіге бөлу.

Бұдан басқа, қалыптағыдан бөлек те нұсқалар бар. Қарапайымдылық үшін бірінші баламаны таңдап көрейік.

Біздің редакторда құжатты `Document` класының объектісі түрінде көрсетеміз. Шамамен алғанда, сұлбалық түрде біздің тип мынадай болуы қажет:

```
typedef vector<char> Line; // жол – бұл символдар векторы
```

```
struct Document {
      list<Line> line;    // құжат – жолдар тізімі
                          // line[i] – бұл i-ші жол
      Document() { line.push_ back(Line()); }
};
```

`Document` класының әрбір объектісі бос жолдан басталады: `Document` класының конструкторы алдымен бос жолды құрайды да, сонан соң тізімді жолдан жолға өте отырып толтырып шығады.

Жолдарға бөлуді және оны оқуды былайша көрсетуге болады:

```
istream& operator>>(istream& is, Document& d)
{
    char ch;
    while (is>>ch) {
        d.line.back().push_back(ch); // символ қосамыз
        if (ch=='\n')
            d.line.push_back(Line()); // жаңа жол қосамыз
    }
    if (d.line.back().size()) d.line.push_back(Line());
                                        // соңғы жолды қосу
    return is;
}
```

`vector` және `list` кластарының соңғы элементке сілтеме қайтаратын `back()` функция-мүшесі бар. Оны пайдалану үшін, сіз оның соңғы элементке сілтеме жасайтынына сенімді болуыңыз керек, `back()` функциясын бос контейнерге қолдануға болмайды. Міне, осы себеппен анықтамаға сәйкес `Document` класының әрбір объектісі `Line` класының бос объектісінен тұруы тиіс. Біз әрбір енгізілген символды, тіпті жаңа жолға көшу символдарын да (`'/n'`) есте сақтайтынымызға назар аударыңыз. Жаңа жолға көшу символдарын есте сақтау жұмысты оңайлатады, бірақ символдарды санау кезінде сақ болған жөн (символдарды қарапайым санау тәсілі босорындар мен жаңа жолға көшу символдарын да есепке алады).

20.6.2 Итерация

Егер құжат `vector<char>` класының объектісі сияқты сақталса, оның элементтері бойынша жылжу қарапайым болар еді. Итераторды жолдар тізімі бойынша қалай жылжытуға болады? Әрине, тізім бойынша жылжуды, яғни орын ауыстыруды `list<Line>::iterator` класы арқылы жасауға болады. Дегенмен, егер біз жолдарды бөлуді ойламай, символдардың бірінен кейін бірін қарастыра отырып, жүріп өтетін болсақ не болады? Біздің `Document` класы үшін арнайы жасалған итераторды пайдалануымызға болады:

```
class Text_iterator {
                // жолдағы символдың позициясын қадағалайды
    list<Line>::iterator ln;
    Line::iterator pos;
public:
    // итераторды 11-жолдың pp позициясына орнатады
    Text_iterator(list<Line>::iterator ll, Line::iterator pp)
      :ln(ll), pos(pp) { }

    char& operator*() { return *pos; }
    Text_iterator& operator++();

    bool operator==(const Text_iterator& other) const
            { return ln==other.ln && pos==other.pos; }
    bool operator!=(const Text_iterator& other) const
            { return !(*this==other) ; }
};

Text_iterator& Text_iterator::operator++()
{
    if (pos==(*ln).end()) {
      ++ln;                  // жаңа жолға көшу
      pos = (*ln).begin();
    }
    ++pos;                   // жаңа символға көшу
    return *this;
}
```

`Text_iterator` класы пайдалы болуы үшін, `Document` класын `begin()` және `end()` сияқты дәстүрлі функциялармен қамтамасыз ету қажет.

```
struct Document {
    list<Line> line;

    Text_iterator begin()  //бірінші жолдың бірінші символы
         { return Textiterator(line.begin() ,
         (*line.begin()) .begin()); }
    Text_iterator end()    //"соңғы жолдан кейін"
    {
        return last<Line>::iterator last=line.end();
        Textiterator (line.end(), (*line.end()).end));
    }
};
```

Біз `(*line.begin()).begin()` әуесқой конструкцияны пайдаландық, себебі орын ауыстыруды `line.begin()` итераторы сілтеме жасап тұрған итератордың позициясынан бастағымыз келеді; оның баламасы ретінде `line.begin()->begin()` функциясын пайдалануға болар еді, өйткені стандартты итераторлар `->` операциясын сүйемелдейді. Енді құжаттың символдары бойынша жылжи отырып, орын ауыстыруға болады.

```
void print (Document& d)
{
  for(Text_iteratorp=d.begin();p!=d.end();++p)cout<<*p;
}

print (my_doc);
```

Құжатты символдар тізбегі түрінде көрсету көп себептерге байланысты пайдалы болып табылады, бірақ әдетте біз символға қарағанда анағұрлым спецификалық ақпаратты қарастырып, құжаттар арқылы жылжимыз. Мысалы, `n` жолын жоятын код фрагментін қарастырайық.

```
void erase_line(Document& d, int n)
{
     if (n<0 || d.line.size()-1<=n) return;
     list<Line>::iterator p = d.line.begin();
     advance(p,n)
     // диапазон шегінен тысқары тұрған жолдарды қарастырмаймыз
     d.line.erase(p);
}
```

`advance(n)` функциясын шақыру итераторды `n` элементке алға жылжытады, `advance()` функциясы – бұл стандартты функция, бірақ біз осыған ұқсас кодты өзіміз де жаза аламыз:

```
template<class Iter> Iter advance (Iter p, int n)
{
     while (n>0) { ++p;--n; }
}
```

`advance()` функциясын индекстеудің имитациясы үшін пайдалануға болатынына назар аударыңыз. Негізінде, `vector` класының `v` атты объектісі үшін `*advance(v.begin(),n)` өрнегі `v[n]` конструкциясына парапарлыққа (эквиваленттілікке) жақын деп айтуға болады. Мұндағы "жақын" сөзі `advance()` функциясының алғашқы `n-1` элементке әрбір қадам бойынша сәйкес келетінін білдіреді, ал мұндағы индекстеу операциясы бірден `n`-ші элементке алып барады. `list` класы үшін біз осы тиімсіздеу тәсілді пайдалануға мәжбүр боламыз. Мұны тізімнің икемділігі үшін төленетін баға деп санау керек.

Егер итератор алға және артқа жылжып орын ауыстыра алатын болса, мысалы `list` класында, онда стандартты `advance()` кітапханалық функциясының теріс аргументі артқа қарай жылжу болатынын білдіреді. Егер итератор индекстеуді жүзеге асыратын болса, мысалы, `vector` класында, онда стандартты `advance()` кітапханалық функциясы оны бірден дұрыс элементке орнатады да, ол `++` операторы арқылы барлық элементтерді біртіндеп аралап, баяу орын ауыстырып отырмайды. Әрине, мұнда стандартты `advance()` функциясы біздің функциямызға қарағанда аздап "ақылдылық" көрсетіп отыр. Мұны айтып өткен дұрыс: өйткені стандартты құралдар өте мұқият жасалады да, біз өзіміз құрғаннан гөрі оларға көп уақыт жұмсалған, сондықтан өзіміз "қолдан" жасағаннан гөрі стандарт құралдарды пайдаланған дұрысырақ болып табылады.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

`advance()` функциясын, ол теріс аргумент алғанда, артқа қарай жылжитындай етіп, қайтадан жазып шығыңыз.

Іздеу – бұл итерацияның ең қалыпты түрі болуы мүмкін шығар. Біз жеке сөздерді, (мысалы, `milkshake` немесе `Gavin`), әріптер тізбегін (мысалы, `secret\nhomestead` – яғни `secret` сөзімен аяқталып, `homestead` сөзімен басталатын жолдарды), регулярлы өрнектерді (мысалы, `[bB]\w*ne` – яғни `B` әріпінің жоғарғы немес төменгі регистрде жазылып, артында `0` немесе бірсыпыра әріптер тұрса және олар `ne` әріптерімен аяқталса, 23-тарауды қ.) және т.с.с. іздейміз. Екінші есепті қалай шығаратынымызды көрсетейік: `Document` класының объектісін сақтау сұлбасын пайдалана отырып жолдарын табамыз. Қарапайым, бірақ тиімсіз алгоритмді пайдаланамыз:

- Құжаттағы ізделінетін жолдың бірінші символын табамыз.
- Осы және келесі символдары ізделінетін жолдардың символдарымен сәйкес келе ме, жоқ па, соны тексереміз.
- Егер сәйкес келсе, онда есеп шығарылды; егер олай болмаса, онда бірінші символдың келесі кездесуін іздейміз.

Қарапайым болуы үшін STL кітапханасындағы мәтіндерді көрсету ережесін, итераторлар жұбымен анықталған тізбектер түрінде көрсетеміз. Бұл бізге іздеу функциясын бүкіл құжатқа емес, оның кез келген бөлігіне қолдануға мүмкіндік береді. Егер біз құжаттан керекті тіркесті тапсақ, онда оның бірінші символына орнатылған итераторды қайтарамыз; егер таппасақ, тізбектің соңында орнатылған итераторды қайтарамыз:

```
Text_iterator find_txt(Text_iterator first, Text_iterator
last, const strings s)
{
    if(s.size()==0)returnlast;//босжолдыіздеугеболмайды
    char first_char = s[0];

    while (true) {
        Text_iterator p = find (first, last, first_char);
        if (p==last || match(p, last, s)) return p;
        first=++p; // келесі символға қараңыз
     }
}
```

Іздеу нәтижесі сәтсіз болғандығының белгісі ретінде сөз тіркесінің соңын қайтару тәсілі STL кітапханасында қабылданған маңызды шарт болып табылады. **match()** функциясы өте қарапайым (тривиалды), ол жай ғана екі символдар тізбегін салыстырады. Оны өзіңіз жазып көріңіз. Тізбектегі символды іздеуге арналған **find()** функциясы, мүмкін, ең қарапайым стандартты алгоритм болып табылатын шығар (21.2 бөлім). Біз өзіміздің **find_txt()** функциямызды, шамамен былай пайдалануымызға болады:

```
Text_iterator p =
   find_txt(my_doc.begin(),my_doc.end(),"secret\nhomestead");
if (p==my_doc.end())
   cout << "табылмады";
else {
   // қандай да бір әрекеттер
}
```

Біздің мәтіндік процессор және оның операциялары өте қарапайым. Біз мүмкіндігі мол "керемет" редактор емес, тиімді түрде жұмыс істей алатын қарапайым редактор құрғымыз келеді.

Бірақ тиімді түрде кірістіру, өшіру және символды іздеу әрекеттерін – қарапайым есеп деп қателеспеу керек. Біз бұл мысалды STL кітапханасында қабылданған программалау ережелеріне сәйкес тізбектер, итераторлар және контейнерлер (**list** және **vector** сияқты) концепцияларының әмбебаптығы мен қуатын көрсету үшін таңдап алдық. Осыларға сәйкес тізбектің соңына орнатылған итераторды қайтару сәтсіздіктің белгісі болып табылады. Егер біз қаласақ, мұндағы, **Document** класын **Text_iterator** итераторымен жабдықтай отырып, STL контейнеріне айналдыруымызға болатынына назар аударыңыз. Ең бастысы, біз **Document** класының объектісін мәндер тізбегі түрінде көрсетуді жасап шықтық.

20.7 `vector`, `list` және `string` кластары

Неліктен біз жолдарды сақтау үшін `list` класын, ал символдар үшін – `vector` класын пайдаланамыз. Дәлірек айтсақ, неліктен ЖОЛДАР тізбегін сақтау үшін `list` класын, символдар тізбегін сақтау үшін - `vector` класын пайдаланамыз? Оған қоса, неліктен жолдарды сақтау үшін біз `string` класын пайдаланбаймыз?

Осы сұрақтың аздап жалпы нұсқасын жасап көрейік. Бізде символдар тізбегін сақтауға арналған төрт тәсіл бар:

- `chart[]` (символдар жиымы)
- `vector<char>`
- `string`
- `list<char>`

Нақты бір есепті шығару үшін осы нұсқалардың қайсысын таңдаймыз? Жай ғана қарапайым есеп үшін бұл нұсқалардың барлығы да бір-бірін өзара алмастыра алады; басқаша айтқанда, олардың интерфейсі өте ұқсас. Мысалы, итератор болса, біз `++` операциясы арқылы элементтер бойынша жылжи аламыз және символдарға қол жеткізу үшін `*` операторын пайдалана аламыз. Егер `Document` класымен байланысқан кодтар мысалдарына қарайтын болсақ, онда біз айқын түрде `vector<char>` класын `list<char>` немесе `string` класымен ешқандай қиындықсыз алмастыра аламыз. Мұндай өзара алмасу мүмкіндігі іргелі артықшылық болып табылады, өйткені ол бізге тиімділікке қарай таңдау жасауға мүмкіндік береді. Бірақ тиімділік мәселелерін қарастырмастан бұрын, біз осы типтердің логикалық мүмкіндіктерін қарастыруымыз керек: олардың әрқайсысы басқалары жасай алмайтын қандай әрекетті орындауы мүмкін?

- `Elem[]`. Өзінің өлшемін білмейді. `begin()`, `end()` функциялары және басқа да контейнерлік функция-мүшелері жоқ. Мүмкін болатын диапазоннан шығып кетуді жүйелі түрде тексере алмайды. C тілінің стилінде немесе C тілінде жазылған функцияларға берілуі мүмкін. Элементтері компьютер жадындағы сыбайлас (көршілес) ұяшықтарда тізбектеле орналасқан. Жиымның өлшемі компиляция кезеңінде анықталады. (`==` және `!=`) тәрізді салыстыру операциялары мен шығару (`<<`) операциясы барлық элементтерге емес, жиымның бірінші элементіне нұсқауышты қолданады.
- `vector<Elem>`. Іс жүзінде, `insert()` және `erase()` функцияларын қосып есептегенде, бәрін де орындай алады. Индекстеу де қарастырылған. `insert()` және `erase()` сияқты тізімдермен орындалатын операциялар символдардың орын ауыстыруымен байланысқан (ірі элементтер үшін және элементтерінің саны көп болған кездерде тиімді болмауы мүмкін). Берілген диапазон шегінен шығып кетуді тексере алады. Элементтері компьютер жадындағы көршілес ұяшықтарда тізбектеле орналасқан. `Vector`

класының объектісі ұлғаюы мүмкін (мысалы, `push_back()` функциясын қолданады). Вектордың элементтері жиымда сақталады (үздіксіз түрде). Элементтерді салыстыру `==`, `!=`, `<`, `<=`, `>` және `>=` операторларының көмегімен жасалады.

- `string`. Қарапайым және пайдалы операцияларды, сонымен қоса, конкатенация сияқты (`+` және `+=`) мәтіндермен орындалатын спецификалық іс-әрекеттер қарастырылады. Элементтері компьютер жадындағы көршілес ұяшықтарда сақталуы міндетті емес. `String` класының объектісін үлкейтуге болады. Элементтерді салыстыру `==`, `!=`, `<`, `<=`, `>` және `>=` операторларының көмегімен жасалады.

- `list<Elem>`. Индекстеуден басқа қалыпты және пайдалы операцияларды қарастырады. `insert()` және `delete()` операцияларын басқа элементтердің орнын ауыстырмай-ақ жасауға болады. Әрбір элементті сақтау үшін екі қосымша сөз қажет (түйіндерге нұсқауыштар үшін). `list` класының объектісін үлкейтуге болады. Элементтерді салыстыру `==`, `!=`, `<`, `<=`, `>` және `>=` операторларының көмегімен жасалады.

Біз бұрын көргендей (17.2 және 20.5 бөлімдерін қ.), жиымдар ең төменгі деңгейде компьютер жадын басқару үшін пайдалы әрі қажетті болып табылады және де олар C тілінде жазылған программалармен өзара әрекеттесуді қамтамасыз ету үшін де керек (бұл туралы толығырақ 27.1.2 және 27.5 бөлімдерде). Бұған қарағанда, `vector` класы қолдануға ыңғайлырақ, себебі оны пайдалану жеңіл орындалады, ол әрі икемді және қауіпсіз болып саналады.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Мұндай айырмашылықтар тізімі нақты кодта нені білдіреді? `char`, `vector<char>`, `list<char>` және `string` типтерінде `"Hello"` мәні арқылы берілген объектілер жиымдарын анықтаңыз да, оны функцияға аргумент ретінде беріңіз. Берілетін сөз тіркесіндегі символдардың санын жазыңыз, оны функциядағы `"Hello"` сөз тіркесімен салыстырып көріңіз (шынымен де `"Hello"` тіркесін бергеніңізге көз жеткізу үшін), сонан соң сөздікте осы сөздердің қайсысы бірінші кездесетінін анықтау үшін, аргументті `"Howdy"` тіркесімен салыстырыңыз. Аргументті өзі типтес басқа айнымалыға көшіріңіз.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Алдыңғы **Мынаны жасап көріңіз** тапсырмасын мәндері {1,2,3,4,5} болып келетін `int`, `vector<int>` және `list<int>` типтеріндегі объектілер жиымы үшін орындаңыз.

20.7.1 `insert` және `erase` операциялары

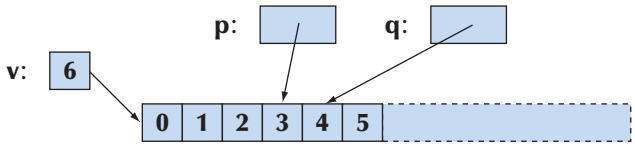
Үнсіз келісім бойынша контейнер ретінде **`vector`** стандартты класы пайдаланылады. Оның көптеген керекті қасиеттері бар, сондықтан баламасын тек қажет болған жағдайда ғана пайдаланған жөн. Оның негізгі кемшілігі болып, (**`(insert()`** және **`erase()`**) тізімдерінің сипаттарына сәйкес операцияларды қолданғанда, векторда қалған элементтерінің орын ауыстыруы орындалады; егер вектор көп элементтерден тұратын болса немесе вектордың элементтерінің өзі ірі объектілерден тұрса, мұнда жұмсалатын шығындар көлемі ұлғаяды. Алайда, бұған онша алаңдаудың қажеті жоқ. Біз ешбір қиындықсыз жарты миллион жылжымалы нүктелі мәндерді **`push_back()`** функциясы арқылы векторға оқып алдық.

Өлшеулер компьютер жадын алдын ала бөлу белгілі бір келеңсіздіктерге әкелмейтіндігін растады. Тиімділікке ұмтылып, елеулі өзгерістер енгізбес бұрын, өлшеулер жасап алған дұрыс (кодтың тиімділік деңгейін болжау тіпті сарапшыларға да қиынға соғады).

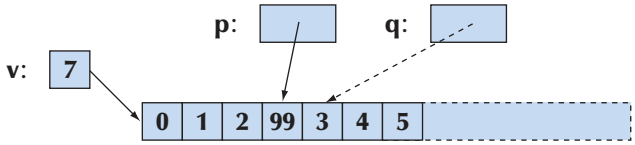
20.6 бөлімінде көрсетілгендей, элементтердің орнын ауыстыру логикалық шектеулерге байланысты болады: тізімдер үшін (**`insert()`**, **`erase()`** және **`pushback()`** сияқты) қалыпты операцияларды орындай отырып, итераторлар немесе вектор элементтеріне нұсқауыштарды сақтаудың қажеті жоқ. Егер элемент орнын ауыстырса, онда сіздің итераторыңыз немесе нұсқауышыңыз керекті элементке орнатылмайды немесе тіпті вектор элементіне сілтеме де жасамауы мүмкін. **`list`** класының (**`map`** класының да, 21.6 бөлімін қ.) **`vector`** класынан негізгі артықшылығы осында жатыр. Егер сізге ірі объектілер топтамасы керек болса немесе программаның көптеген бөліктерінде объектілерге сілтеме жасау қажет болса, **`list`** класын пайдалану мүмкіншіліктерін қарастырыңыз.

`vector` және **`list`** кластарындағы **`insert()`** және **`erase()`** функцияларын салыстырамыз. Алдымен, осының қағидалы (принципиалды) сәттерін көрсету үшін арнайы құрылған мысал қарастырайық:

```
vector<int>::iterator p = v.begin(); //векторды аламыз
++p; ++p; ++p;  // итераторды 3-элементке орнатамыз
vector<int>::iterator q = p;
++q;            // итераторды 4-элементке орнатамыз
```

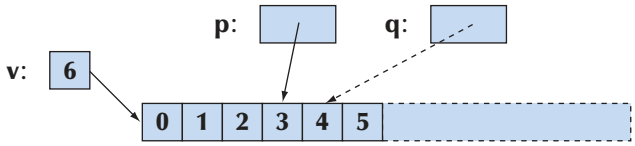



```
p=v.insert(p,99);
// p итераторы кірістірілген элементке сілтеме жасайды
```



Енді `q` итераторы дұрыс болмай шығады. Вектордың өлшемін үлкейту кезінде элементтер басқа жерге орын ауыстыруы мүмкін. Егер компьютер жадында `v` векторының қосымша орны болса, онда ол дәл сол жерде үлкейтіледі, ал `q` итераторы 4-элементке емес, 3-элементке сілтеме жасап тұрады, бірақ мұнан пайда табамын деп ойлауға болмайды.

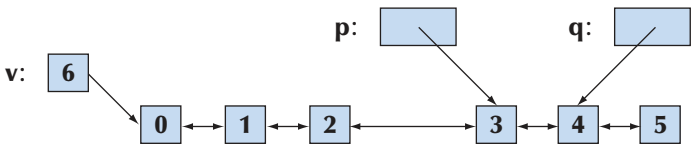
```
p = v.erase(p);        // p итераторы өшірілген элементтен
                       // кейінгі элементе сілтеме жасайды
```



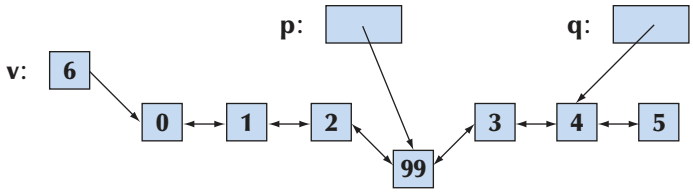
Басқаша айтқанда, `insert()` функциясынан кейін `erase()` функциясы тұрса, онда вектордың мазмұны өзгермейді, бірақ `q` итераторы дұрыс болмай шығады. Алайда, олардың арасындағы барлық элементтерді енгізу нүктесінен оңға қарай жылжытып орын ауыстырсақ, онда біз `v` векторының көлемін үлкейткен кезде оның барлық элементтері енгізу нүктесінен бастап, компьютер жадына қайтадан жаңаша орналасуы әбден мүмкін.

Салыстыру үшін біз дәл осы әрекеттерді `list` класының объектісімен де жасап шықтық:

```
list<int>::iterator p = v.begin();  // тізім аламыз
++p; ++p; ++p;      // итераторды  3-элементке орнатамыз
list<int>::iterator q = p;
++q;                // итераторды  4-элементке орнатамыз
```

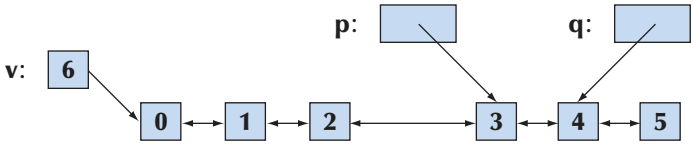


```
p = v.insert(p,99);
// p итераторы енгізілген элементке сілтеме жасап тұр
```



q итераторы бұрынғыша 4-элементке сілтеме жасап тұрғанына назар аударыңыз.

```
p = v.erase(p);     //  p итераторы өшірілген элементтен
                    // кейінгіге сілтеме жасап тұр
```



Біз тағы да, қай жерден бастасақ, сол жерге келдік. Алайда, мұның **vector** класынан айырмашылығы, **list** класымен жұмыс жасай отырып, біз элементтердің орнын ауыстырмадық және **q** итераторы әрқашанда дұрыс болып қалып отырды.

list<char> класының объектісі басқа үш баламаға қарағанда, компьютер жадында кемінде үш есе көп орын алып тұр, – компьютерде **list<char>** класының объектісі әрбір элементке 12 байт, **vector<char>** класының объектісі бір элементке 1 байт орын алып тұр. Символдардың саны көбейген жағдайда бұл жайт маңызды болуы мүмкін.

String класынан **vector** класының артықшылығы қандай? Бір қарағанда, олардың мүмкіндіктер тізімі **string** класы **vector** класы сияқты барлығын да немесе одан да көп жасауы мүмкін екендігін көрсетеді. Бұдан мәселе туындайды: **string** класы көп нәрсе жасай алатындықтан, оны оңтайландыру (оптималдау) қиынға соғады. Алайда, **vector** класын **pushback()** сияқты компьютер жады операцияларының көмегімен оңтайландыруға болады, ал **string** класымен олай жасауға болмайды. Сонымен қатар **string** класында көшіру операциясын қысқа сөз тіркестерімен және C тілі стиліндегі сөз тіркестерімен жұмыс істеу кезінде оңтайландыруға болады. Мәтіндік редакторға арналған мысалда, **insert()** және **delete()** функцияларын пайдаланғандықтан, біз **vector** класын таңдадық. Бұл шешім тиімділік тұрғысына байланысты түсіндіріледі. Мұндағы негізгі логикалық айырмашылық, біз кез келген типтегі элементтері бар вектор құра аламыз. Бізде тек

символдармен жұмыс жасаған кезде, таңдау мүмкіндігі болады. Қорытындылай келе айтарымыз, егер бізге конкатенация сияқты немесе босорындармен бөлінген сөздерді оқу тәрізді тіркестермен орындалатын операциялар қажет болса, біз **string** класын емес, **vector** класын пайдалануды ұсынамыз.

20.8 Біздің vector класымызды STL кітапханасына бейімдеу

Енді **vector** класына 20.5 бөліміндегі **begin()**, **end()** функцияларын және **typedef** нұсқауын қосқаннан кейін, оның **std::vector** класының ең жақын аналогы болуы үшін, тек **insert()** және **erase()** функциялары ғана жетіспейді.

```
template<class T,class A = allocator<T> > class vector
{
    int sz;    // өлшемі
    T* elem;   // элементтерге нұсқауыш
    int space; // элементтер саны плюс бос ұяшықтар саны
    A alloc;   // элементтер үшін жады аймағын
               // бөлгішті пайдаланады
public:
    //...қалғандарының барлығы 19-тарауда және
    // 20.5 бөлімде сипатталған...
    typedef T* iterator;
    // Elem* - барынша қарапайым итератор

    iterator insert(iterator p,const T& val);
    iterator erase(iterator p);
};
```

Мұнда біз тағы итератордың типі ретінде **T*** типіндегі элементке нұсқауышты пайдаландық. Бұл – барлық мүмкін шешімдердің ішіндегі ең қарапайымы. Берілген диапазон шегінен шығып кетуді тексеретін итераторды өңдеуді оқырмандар жаттығу ретінде өздері жасай алады (20-жаттығу).

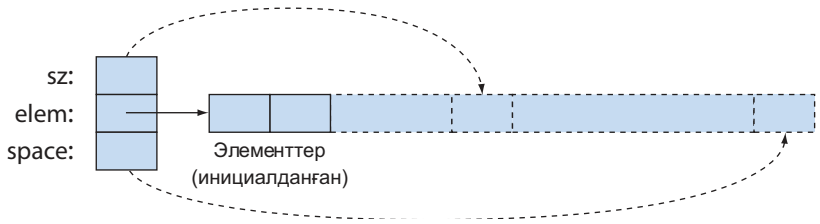
vector класы сияқты компьютер жадының көршілес ұяшықтарында сақталатын мәліметтер типі үшін **insert()** және **erase()** сияқты тізімдермен орындалатын операцияларды элементтер саны аз векторлармен жұмыс кезінде тиімді екенін адамдар көбінесе, жазбайды. Алайда, **insert()** және **erase()** сияқты тізімдермен орындалатын операциялар шағын векторлармен немесе элементтер саны аз векторлармен жұмыс істеу кезінде өте тиімді әрі пайдалы болып шықты. Біз тізімдердің басқа да дәстүрлі операциялары сияқты **pushback()** функциясының да пайдалы екенін байқап жүрміз.

Негізінде, біз жойылатын элементтен (жылжыту және жою) кейін тұрған элементтердің барлығын да көшіре отырып, **vector<T>::erase()** функциясын жүзеге асырдық.

19.3.6 бөліміндегі **vector** класының анықтамасын көрсетілген қосымшалармен бірге пайдалана отырып, келесі кодты аламыз:

```
template<clase T, class A>
vector<T#A>::iterator vector<T,A>::erase(iterator p)
{
    if (p==end()) return p;
    for (iterator pos = p+1; pos!=end(); ++pos)
      *(pos-1) = *pos; // элементті бір орын солға жылжыту
    alloc.destroy(end()-1);
    // соңғы элементтің артық көшірмесін өшіру
    --sz;
    return p;
}
```

Бұл кодты графикалық түрде көрсеткенде, оны оңай түсінуге болады:



erase() функциясының коды өте қарапайым, бірақ мүмкін, қағазда бірнеше мысалды қарастырған жеңілірек болатын ба еді. **vector** класының бос объектісі дұрыс өңделе ме? **p==end()** тексеруі не үшін қажет? Вектордағы соңғы элементті жойғаннан кейін не болады? Егер біз индекстеуді пайдалансақ, осы кодты оқу жеңілірек болар ма еді?

vector<T,A>::insert() функциясын жүзеге асыру біраз қиындау болады:

```
template<class T, class A>
vector<T,A>::iterator vector<T,A>::insert(iterator p,
                                          const T& val)
{
    int index = p-begin();
    if(size()==capacity()) reserve(2*size());
    // орын бар екеніне көз жеткізу
```

```
    // алдымен соңғы элементті инициалданбаған
    // ұяшыққа көшіреміз:
    alloc.construct(elem+sz,*back());
    ++sz;
    iterator pp = begin()+index;
    // val мәнін жазуға арналған орын
    for(iterator pos = end()-1; pos!=pp; --pos)
        *pos = *(pos-1); // элементті оңға бір орынға жылжыту
    *(begin()+ index) = val;  // "insert" val
    return pp;
}
```

Келесі фактілерге назар аударыңыз:

- Итератор тізбектің шегінен тыс орналасқан ұяшықтарға сілтеме жасай алмайды, сондықтан біз **elem+space** сияқты нұсқауыштарды пайдаланамыз. Бұл – компьютер жадын бөлгіштерді итераторлардың емес, нұсқауыштардың негізінде жүзеге асуының бір себебі.
- Біз **reserve()** функциясын пайдаланғанда, элементтер компьютердің жаңа жады аймағына көшірілуі мүмкін. Сондықтан біз оған орнатылған итераторды емес, өшірілетін элементтің индексін есте сақтауымыз керек. Вектордың элементтері компьютер жадына қайта жазылып орналасқан кезде, оларға орнатылған итераторлар дұрыс болмай қалады – оларды ескі адрестерге сілтеме ретінде қарастыруға болады.
- Біздің A компьютер жадын бөлгішті пайдалануымыз интуитивті түрде жүргізілген, ол дәл емес болып табылады. Егер сізге контейнерді жүзеге асыру керек болса, онда стандартты мұқият қарап шығу керек.
- Осыған ұқсас нақтылықтар, компьютер жадымен төменгі деңгейде тікелей жұмыс жасаудан құтылуға мүмкіндік береді. Әрине, стандартты **vector** класы басқа да стандартты контейнерлер сияқты, осындай маңызды семантикалық нақтылықтарды дұрыс іске асырады. Бұл – "қолдан жасалған" шешімдерді емес, стандартты кітапханаларды пайдалануды ұсыну себептерінің бірі.

Тиімділікке қатысты себептерге байланысты, біз **insert()** және **erase()** функцияларын 100 мың элементтен тұратын орташа векторлар элементтеріне пайдаланбауымыз керек; ол үшін **list** класын пайдаланған жөн (және **map** класын, 21.6 бөлімін қ.). Дегенмен, **insert()** және **erase()** операцияларын барлық векторларға да қолдануға болады, ал шағын мәліметтердің орнын ауыстырғанда олардың жұмыс өнімділігі өте жоғары болып табылады, себебі қазіргі компьютерлер мұндай көшірулерді өте тез орындайды (20-жаттығуды қ.). Аз ғана саны бар шағын элементтерден тұратын тізімдерден (байланысқан) аулақ болыңыз.

20.9 Құрамдас жиымдарды STL кітапханасына бейімдеу

Біз құрамдас жиымдардың кемшіліктерін көп рет көрсеттік: олар кішкене сылтау болса нұсқауыштарды жанамалы түрде түрлендіреді, оларды меншіктеу арқылы көшіруге болмайды, олар өз өлшемін білмейді (20.5.2 бөлімін қ.) және т.б. Оған қоса, біз олардың артықшылықтарын да айттық: олар физикалық жадыны өте жақсы модельдейді.

Жиымдар мен контейнерлердің артықшылықтарын пайдалану үшін, біз жиымдардың жақсы қасиеттерін алып, бірақ олардың кемшіліктерінен алмайтын, **array** типтес контейнерді құра аламыз. **array** класының нұсқасы C++ тілінің Стандарттау комитетінің техникалық есеп беруінің бөлігі ретінде стандартқа енгізілген. Оның осы есеп беруге енгізілген қасиеттері барлық компиляторларда іске асырылуы міндетті емес, сондықтан **array** класы сіздің стандартты кітапханаңызда болмауы да мүмкін. Дегенмен, оның идеясы қарапайым және пайдалы.

```
template <class T, int N> // онша толық стандартты жиым емес
struct array {
    typedef T value_type;
    typedef T* iterator;
    typedef T* const_iterator;
    typedef unsigned int size_type; // индекс типі

    T elems[N];
    // тікелей жасау/көшіру/жою талап етілмейді

    iterator begin() { return elems; }
    const_iterator begin() const { return elems; }
    iterator end() { return elems+N; }
    const_iterator end() const { return elems+N; }

    size_type size() const;

    T& operator[](int n) { return elems[n]; }
    const T& operator[](int n) const { return elems[n]; }

    const T&at(intn)const;//диапазондытексеріпқолжеткізу
    T& at(int n);         //диапазондытексеріпқолжеткізу

    T * data() { return elems; }
    const T * data() const { return elems; }
};
```

Бұл анықтама толық емес және толығымен стандартқа сәйкес келмейді, бірақ ол негізгі идеяны жақсы бейнелейді. Оған қоса, егер де сіздің стандартты кітапханаңызда ол болмаса да, `array` класын пайдалануға мүмкіндік береді. Егерде ол бар болса, онда оны `<array>` тақырыбынан іздеу керек. Мынаған назар аударыңыз: `array<T,N>` класының объектісіне оның өлшемі `N` белгілі болғандықтан, біз мұнда `vector` класындағы сияқты, `=`, `==`, `!=` операторларын қарастыра аламыз (міндеттіміз).

Мысалы, 20.4.2 бөліміндегі `high()` стандартты функциясы бар жиымды пайдаланамыз:

```
void f()
{
    array<double,6> a = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5 };
    array<double,6>::iterator p=high(a.begin(),a.end());
    cout << "ең үлкен мәні " << *p << endl;
}
```

Мынаған назар аударыңыз, `high()` функциясын жазған кезде біз array класы туралы ойлаған жоқпыз. `high()` функциясын `array` класының объектісіне қолдану мүмкіндігі мұндағы, екі жағдайда да біздің стандартты келісімдерді ұстанғанымыздың қарапайым салдары болып табылады.

20.10 Контейнерлерге шолу

STL кітапханасында бірнеше контейнерлер бар:

Стандартты контейнерлер	
`vector`	Элементтердің үздіксіз тізбегі. Келісім бойынша контейнер ретінде қолдануды ұсынамыз.
`list`	Екібайланысты тізім. Бұрыннан бар элементтердің орнын ауыстырмай енгізіп қою және өшіру қажет болған жағдайда ғана қолдануды ұсынамыз.
`deque`	Тізім мен вектордың араласуы. Өзіңіз алгоритмдер мен машиналық архитектураның сарапшысы болмайынша пайдаланбауды ұсынамыз.
`map`	Теңгеріліп реттелген бұтақ. Элементтермен мәні бойынша қатынас құру қажет болғанда ғана қолдануды ұсынамыз (21.6.1-21.6.3 бөлімдерін қ.).
`multimap`	Кілттің бірнеше көшірмесі сақталатын теңгеріліп реттелген бұтақ. Элементтермен мәні бойынша қатынас құру қажет болғанда ғана қолдануды ұсынамыз (21.6.1-21.6.3 бөлімдерін қ.).

Стандартты контейнерлер (*жалғасы*)	
`unordered_map`	Хеш-кесте; `map` класының оңтайландырылған нұсқасы. Егер жоғары жұмыс өнімділігі қажет болатын болса және хештеудің жақсы функциясын жасай алсаңыз, қолдануды ұсынамыз (21.6.4 бөлімін қ.).
`unordered_multimap`	Кілттің бірнеше көшірмесі сақталатын хештеу кестесі; `multimap` класының оңтайландырылған нұсқасы. Жоғары жұмыс өнімділігі қажет болатын болса және хештеудің жақсы функциясын жасай алсаңыз, `map` класының үлкен объектілері үшін қолдануды ұсынамыз (1.6.4 бөлімін қараңыз).
`set`	Теңгеріліп реттелген бұтақ. Егер жеке мәндерді қадағалау қажет болған жағдайда қолдануды ұсынамыз (21.6 5 бөлімін қ.).
`multiset`	Кілттің бірнеше көшірмесі сақталатын теңгеріліп реттелген бұтақ. Жеке мәндерді қадағалау қажет болған жағдайда қолдануды ұсынамыз (21.6. 5 бөлімін қ.).
`unordered_set`	`unordered_map` класына ұқсас, бірақ жұптары үшін емес, мәндері үшін (кілт, мән).
`unordered_multiset`	`unordered_multimap` класына ұқсас, бірақ мәндері үшін, жұптары үшін емес (кілт, мән).
`array`	Құрамдас жиымдардың көптеген кемшіліктерінен арылған, өлшемі бекітілген жиым (20.6 бөлімін қ.).

Осындай контейнерлер жайлы және оларды пайдалану туралы көптеген қосымша ақпараттар топтамаларын Интернетте орналасқан кітаптар мен құжаттардан алуға болады. Сенім артуға болатын бірнеше ақпарат көздерін ұсынамыз:

Austern, Matt, ed. "Technical Report on C++ Standard Library Extensions," ISO/IEC PDTR 19768. (Colloquially known as TR1.)

Austern, Matthew H. *Generic Programming and the STL*. Addison-Wesley, 1999. ISBN 0201309564. Koenig, Andrew, ed. *The C++ Standard*. Wiley, 2003. ISBN 0470846747. (Not suitable for novices.)

Koeing, Andrew, ed. The C++ Standard. Wiley, 2003. ISBN 0470846747.

Lippman, Stanley B.,Josée Lajoie, and Barbara E. Moo. *The C++ Primer*. Addison-Wesley, 2005. ISBN 0201721481. (Use only the 4th edition.)

Musser, David R., Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition*. Addison-Wesley, 2001. ISBN 0201379236.

Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, 2000. ISBN 0201700735.

STL кітапханасын жүзеге асыру туралы құжатты және SGI (Silicon Graphics International) компаниясының енгізу-шығару ағындарының кітапханасын мына *www.sgi.com/tech/stl* веб-парақтан табуға болады. Осы веб-парақта аяқталған программалардың келтірілгеніне назар аударыңыз.

Dinkumware компаниясының STL кітапханасын жүзеге асырылуы туралы құжаттарды *www.dinkumware.com/manuals/default.aspx* веб-парақтан табуға болады (бұл кітапхананың бірнеше нұсқасы бар екенін назарға алыңыз.)

Rogue Wave компаниясының STL кітапханасын жүзеге асырылуы туралы құжаттарды *www2.roguewave.com/support/docs/index.cfm* веб-парақтан табуға болады.

Сіз өзіңізді алданғандай сезінесіз бе? Біз барлық контейнерлерді сипаттап, оларды қалай пайдалану керектігін көрсетеді деп ойлайсыз ба? Бұл мүмкін емес. Олардың барлығын бір кітапта сипаттау үшін, өте көптеген стандартты мүмкіншіліктерді, кітапханаларды және пайдалы тәсілдерді қарастыру керек. Программалаудың бір адам игеретін өте көп мүмкіндіктері бар. Бұған қоса, көбінесе, программалау – бұл өнер деп айтылады. Программалаушы ретінде сіз тілдің мүмкіндіктері, кітапханасы және технологиясы туралы ақпараттарды іздеуге дағдылануыңыз керек. Программалау – динамикалық және тез өркендейтін сала, сондықтан өзіңіз білетіндеріңізбен қанағаттанып және сіз білмейтін заттардың бар екеніне сабырлылық танытыңыз. "Анықтамалықтан іздеу" – бұл көптеген сұрақтардың зерделі жауабы. Өзіңіздің тәжірибеңіздің артуына байланысты сіз көбінесе тап осылай жасайсыз.

Басқа жағынан, `vector`, `list` және `map` кластарын, 21-тарауда сипатталған стандартты алгоритмдерді меңгеріп, сіз STL кітапханасының басқа да контейнерлерімен жеңіл жұмыс жасап үйренесіз. Стандартты емес контейнерлермен жұмыс істеу үшін не қажет екенінің бәрін біліп алып, оларды өзіңіз де программалай аласыз.

Контейнер деген не? Осы ұғымның анықтамасын жоғарыда көрсетілген кез келген мәліметтер көзінен табуыңызға болады. Мұнда біз жасанды емес анықтама ғана береміз. Сонымен, STL кітапханасы контейнерінің келесідей қасиеттері бар:

- `[begin():end())` элементтер тізбегін көрсетеді.
- Контейнермен орындалатын операциялар элементтерді көшіреді. Көшіруді меншіктеу арқылы немесе көшіру конструкторы арқылы орындауға болады.
- Элементтер типі `value_type` деп аталады.
- Контейнер аттары `iterator` және `const_iterator` болып келетін итератор типтерінен тұрады. Итераторлар `*`, `++` (префиксті және постфиксті түрлері), семантикасына сәйкес `==` және `!=` операцияларын қамтамасыз етеді. `list` класына арналған итераторлар тізбек бойынша кері бағытта орын ауыстыруға керекті операторды қарастырады; мұндай

итераторларды қосбағытты (**bidirectional iterator**) итераторлар деп атайды. **vector** класына арналған итераторлар **--**, **[ ]**, **+** және **-** операторларын қарастырады. Бұл итераторларды еркін бағытты (**random-access iterators**) итераторлар деп атайды (20.10.1 бөлімін қ.).

- Контейнерлердің **insert()** және **erase()**, **front()** және **back()**, **push_back()** және **pop_back()**, **size()** және т.с.с. функциялары бар; **vector** және **map** кластары индекстеу операциясын қамтамасыз етеді (мысалы, **[ ]** операторы).

- Контейнерлер элементтерді салыстыру (**==**, **!=**, **<**, **<=**, **>** және **>=**) операторларын қамтамасыз етеді. Контейнерлер **<**, **<=**, **>** және **>=** операциялары үшін лексикографикалық ретке келтіруді қамтамасыз етеді; басқаша айтқанда, орын ауыстыруды бірінші элементтен бастау үшін, элементтерді салыстырады.

Осы тізімнің мақсаты – оқырмандарға белгілі бір шолу жасап беру. Толық ақпарат Б қосымшасында келтірілген. Бұлардан дәлірек спецификациясы мен операциялардың толық тізімі *The C++ Programming Language* кітабында немесе стандартта келтірілген.

Мәліметтердің кейбір типтерінде стандартты контейнерлердің көптеген қасиеттері бар, бірақ барлығы емес. Кейде біз оларды "контейнер тәріздестер" деп атаймыз. Олардың ішіндегі қызықтыларының бірсыпырасы төменде келтірілген:

"контейнер тәріздестер"	
T[n] built-in array	**size()** функциясы және басқа да функция-мүшелері жоқ; егер таңдау болса, **vector**, **string** немесе **array** тәрізді контейнерді пайдалануды ұсынамыз, бірақ құрамдас жиымдарды емес.
string	Тек символдарды ғана сақтайды, бірақ, конкатенация сияқты (**+** және **+=**) мәтінді өңдеу, түрлендіру (манипуляциялау) үшін пайдалы операцияларды орындауды қамтамасыз етеді. Стандартты **string** класын пайдалануды ұсынамыз.
valarray	Векторлық операциялары бар сандар векторы, бірақ оның өнімділікті арттыруға бағытталған көптеген шектеулері бар. Тек векторлық есептеулер саны көп болған кезде ғана пайдалануды ұсынамыз.

Оған қоса, көптеген адамдар және ұйымдар стандарттың талаптарын қанағаттандыратын (немесе соған жақын) өздерінің жеке контейнерлерін құрастырады.

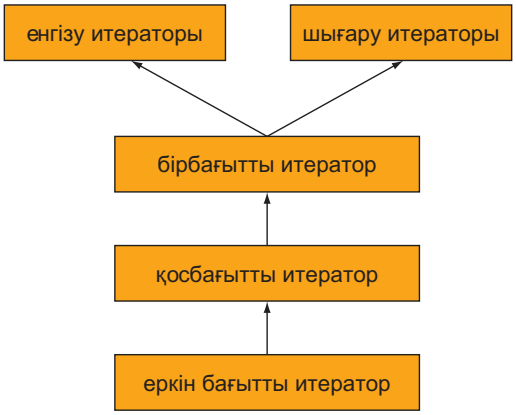
Егер сіздің күмәніңіз болса, онда **vector** класын пайдаланыңыз. Егер сіздің оны жасамауға салмақты себептеріңіз болмаса, онда **vector** класын пайдаланыңыз.

20.10.1 Итераторлардың санаттары

Біз итераторлар туралы олардың барлығы да бірін-бірі өзара алмастыра алады деп айтқан болатынбыз. Бірақ олар тізбектегі әрбір элементті тек бір рет қана оқып, ары жылжытатындай сияқты қарапайым операцияларды орындау кездерінде ғана эквивиалентті болып табылады. Егер сіз одан да кеңірек әрекеттер орындауды қаласаңыз, мысалы, кері бағытта орын ауыстыру немесе элементке еркін қол жеткізу, онда сіздерге жетілдірілген итераторлар қажет болады.

Итераторлар санаттары (категориялары)	
оқу үшін итератор	Біз ++ операторын пайдаланып, алға қарай жылжуымызға болады және * операторы арқылы элементтердің мәнін оқимыз. Итераторлардың осындай түрін istream енгізу ағыны ұсынады (21.7.2 бөлімін қ.). Егер `(*p).m` мәні дұрыс болса, онда қысқарту ретінде `p->m` конструкциясын пайдалануға болады.
жазу үшін итератор	Біз `++` операторын пайдаланып, алға қарай жылжуымызға болады және `*` операторы арқылы элементтердің мәнін жаза аламыз. Итераторлардың осындай түрлері `ostream` шығару ағыны ұсынады (21.7.2 бөлімін қ.).
бірбағытты итератор	Біз `++` операторын пайдаланып, алға қарай жылжи отырып, `*` операторы арқылы (әрине, элементтер константалық болмаса) элементтердің мәнін оқимыз немесе жазамыз. Егер `(*p).m` мәні дұрыс болса, онда қысқарту ретінде `p->m` конструкциясын пайдалануға болады.
қосбағытты итератор	Біз `*` операторы арқылы (әрине, егер элементтері константалық болмаса) элементтердің мәнін оқи немесе жаза отырып, алға (`++` операторы арқылы) және артқа (`--` операторы арқылы) қарай жылжуымызға болады. Егер `(*p).m` мәні дұрыс болса, онда қысқарту ретінде `p->m` конструкциясын пайдалануға болады.
еркін бағытты итератор	Біз `*` немесе `[]` операторы арқылы (әрине, егер элементтері константалық болмаса) элементтердің мәнін оқи немесе жаза отырып, алға (`++` операторы арқылы) және артқа (`--` операторы арқылы) қарай жылжуымызға болады. Біз еркін бағытты итераторды индекстей аламыз, сондай-ақ, `+` және `-` операторларын қолданып оған бүтін сан қосуға немесе азайтуға болады. Біз бірінен-бірін азайту арқылы бір ғана тізбекке орнатылған итераторлардың (еркін бағытты) арақашықтығын есептей аламыз. `vector` класы дәл осындай итератор түрлерін қамтамасыз етеді. Егер `(*p).m` мәні дұрыс болса, онда қысқарту ретінде `p->m` конструкциясын пайдалануға болады.

Қарастырылған операцияларды қадағалай отырып, жазу немесе оқу итераторлары үшін қосбағытты итераторды пайдалануға болатындығына көз жеткізуге болады. Сондай-ақ, қосбағытты итератор бірбағытты итератор болып, ал еркін бағытты итератор – қосбағытты болып табылады. Графикалық түрде итераторлардың санаттарын былай көрсетуге болады:



Итераторлар санаттары кластар болып табылмайтындығына назар аударыңыздар. Бұл мұралау арқылы іске асатын кластар иерархиясы емес.

ТАПСЫРМА

1. `int` типтес он элементі бар сандық жиымды анықтаңыз {0,1,2,3,4,5,6,7,8,9}.
2. Осы он элементпен `vector<int>` класының объектісін анықтаңыз.
3. Осы он элементпен `list<int>` класының объектісін анықтаңыз.
4. Екінші жиым, вектор және тізімді анықтаңыз, олардың әрқайсысы сәйкесінше бірінші жиыммен, вектормен және тізіммен инициалданады.
5. Жиымдағы әрбір элементтің мәнін екіге арттырыңыз; жиымдағы әрбір элементтің мәнін үшке арттырыңыз; жиымдағы әрбір элементтің мәнін беске арттырыңыз;
6. Қарапайым `copy()` операциясын жазыңыз:

   template<class Iter1, class Iter2> copy (Iter f1, Iter1 el, Iter2 f2);

   Дәл стандартты кітапхананың көшіру функциясындай етіп `[f1,e1)` көшіру тізбегін `[f2,f2+(e1-f1))` тізбегіне көшіретін операцияны жазыңыз. Егер `f1==e1` болса, онда тізбек бос болып, көшіретін ештеңенің жоқ екеніне назар аударыңыз.
7. Өзіңіз жасаған `copy( )` функциясын жиымды векторға көшіру үшін немесе тізімді жиымға көшіру үшін пайдаланыңыз.
8. Стандартты кітапхананың `find()` функциясын вектордың 3-ке тең мәні бар екенін тексеру үшін қолданып, егер векторда осы сан болса, оның вектордағы позициясын экранға шығарыңыз. Стандартты кітапхананың `find()` функциясын тізімде 27-ге тең мәннің бар екенін анықтау үшін қолдану керек. Егер тізімде осы сан болса, онда оның тізімдегі позициясын экранға шығарыңыз. Бірінші элементтің позициясы нөлге тең, екінші

элементтің позициясы бірге тең және т.с.с. Егер `find()` функциясы тізбектің соңына орнатылған итераторды қайтарса, онда ол мәннің тізімнен табылмағаны деп біліңіз. Әрбір кезеңнен кейін программаны тесттен өткізуді ұмытпаңыз.

БАҚЫЛАУ СҰРАҚТАРЫ

1. Әртүрлі адамдар жазған программалар неге әртүрлі болады? Мысал келтіріңіз.
2. Әдетте мәліметтер туралы ойлап отырып, біз қандай қарапайым сұрақтар қоямыз?
3. Мәліметтерді сақтаудың әртүрлі тәсілдерін көрсетіңіз.
4. Мәліметтер топтамасымен (коллекциясымен) қандай негізгі операциялар орындауға болады?
5. Мәліметтерді сақтау кезінде қандай принциптерді ұстанған жөн?
6. STL кітапханасындағы тізбек дегеніміз не?
7. STL кітапханасындағы итератор дегеніміз не? Итераторлар қандай операцияларды сүйемелдейді?
8. Итераторды келесі элементке қалай орнатуға болады?
9. Итераторды алдыңғы элементке қалай орнатуға болады?
10. Егер итераторды тізбектің соңынан кейін тұрған ұяшыққа орнатуға талпынсаңыз не болады?
11. Итератордың қандай түрлері алдыңғы элементке қарай орын ауыстыра алады?
12. Мәліметтерді алгоритмдерден бөліп пайдалану неліктен пайдалы?
13. STL дегеніміз не?
14. Байланысқан тізім деген не? Оның вектордан айырмашылығы неде?
15. Түйін (байланысқан тізімде) деген не?
16. `insert()` функциясы не істейді? `erase()` функциясы не істейді?
17. Тізбектің бос екендігін қалай анықтауға болады?
18. list класы үшін итераторда қандай операциялар қарастырылған?
19. STL кітапханасын пайдалана отырып, контейнер арқылы жылжуды, яғни орын ауыстыруды қалай қамтамасыз етуге болады?
20. Қандай жағдайларда `vector` емес, `string` класын пайдаланған жөн?
21. Қандай жағдайларда `vector` емес, `list` класын пайдаланған жөн?
22. Контейнер деген не?
23. Контейнерде `begin()` және `end()` функциялары не істеуі керек?
24. STL кітапханасында қандай контейнерлер қарастырылған?

25. Итераторлардың санаттарын атап өтіңіз. STL кітапханасында итераторлардың қандай түрлері жүзеге асырылған?
26. Еркін бағытты итераторда қосбағытты итераторларда сүйемелденбейтін қандай операциялар қарастырылған?

ТЕРМИНДЕР

`begin( )`	`value_type`	қосбайланысты тізім
`end( )`	array контейнері	итератор
`erase( )`	алгоритм	итерация
`insert( )`	байланысқан тізім	тізбек
`size_type`	бірбайланысты тізім	үздіксіз жады
`STL`	бос тізбек	элемент
`typedef`	контейнер	

ЖАТТЫҒУЛАР

1. Егер сіз "**Мынаны жасап көріңіз**" қиындыларынан тапсырмаларды әлі орындамасаңыз, онда оны қазір жасаңыз.
2. 20.1.2 бөліміндегі Джек пен Джилдың мысалын программалап көріңіз. Тесттен өткізу үшін бірнешге шағын файлдарды қолданыңыз.
3. Палиндромға байланысты мысалды (20.6 бөлімін қ.) талдап көріңіз; әртүрлі тәсілдерді пайдаланып, 2 пункттегі тапсырманы тағы да орындаңыз.
4. STL кітапханасының тәсілдерін пайдаланып, 20.3.1 бөліміндегі Джек пен Джиллдың мысалындағы қатені тауып жөндеңіз.
5. vector класы үшін енгізу және шығару (`>>` және `<<`) операторларын анықтаңыз.
6. 20.6.2 бөліміндегі ақпаратты пайдаланып, `Document` класы үшін "табу және ауыстыру" операциясын жазыңыз.
7. Реттелмеген `vector<string>` класында лексикографикалық түрде соңғы жолды анықтаңыз.
8. `Document` класының объектісіндегі символдар санын есептейтін функция жазыңыз.
9. `Document` класының объектісіндегі сөздер санын есептейтін функция жазыңыз. Екі нұсқаны ескеріңіз: біріншісінде, сөздер – босорындармен бөлінген символдар тізбегі, екіншісінде, сөздер – әліпбиден тұратын үзіліссіз символдар тізбегі. Мысалы, бірінші анықтама бойынша `alpha.numeric` және `as12b` – бұл сөздер, ал екіншісі бойынша – олардың әрқайсысы екі сөз сияқты қарастырылады.

10. Қолданушы ажыратқыш-символдар жиынтығын өзі беріп, сөздердің санын есептейтін программа жазыңыз.
11. `vector<double>` класының объектісін құрып, оған `list<int>` типтес тізімнің элементтерін параметр ретінде беріп (сілтеме арқылы) көшіріңіз. Көшірменің толық әрі дұрыс екенін тексеріңіз. Сонан соң элементтерді өсулері бойынша реттеп, экранға шығарыңыз.
12. 20.4.1 және 20.4.2 бөлімдеріндегі `list` класының анықтамасын аяқтап, `high()` функциясының жұмысын көрсетіңіз. Тізімнің соңынан кейін тұрған түйінді көрсететін `Link` класының объектісі үшін жады аймағын бөліңіз.
13. Шын мәнісінде, бізге `list` класындағы соңғы элементтен кейін тұрған `Link` класының нақты объектісі қажет емес. Алдыңғы жаттығудағы өз шешіміңізді былай етіп толықтырыңыз: `Link (list<Elem>::end())` класының жоқ объектісіне нұсқауыш ретінде 0 мәні пайдаланылатын болсын; басқаша айтқанда, бос тізімнің өлшемі (көлемі) жеке нұсқауыштың өлшеміне тең болуы мүмкін.
14. `std::list` класының стиліне қарап, бірбайланысты `slist` тізімін анықтаңыз. `list` класында алдыңғы элементке нұсқауыш жоқ болғандықтан, одан `slist` класының қандай операцияларын алып тастауға болады?
15. Нұсқауыштар векторына ұқсас `pvector` класын анықтаңыз, мұны `pvector` класында объектіге нұсқауыштары бар екендігін және әрбір объект оның деструкторы арқылы жойылып отыратындығын есепке алмай отырып жүзеге асыру керек.
16. `pvector` класына ұқсас `ovector` класын анықтаңыз, мұны `[]` және `*` операциялары нұсқауыштарды емес, өзінің сәйкес элементі сілтеме жасайтын объектіге сілтеме қайтаратынын есепке алмай отырып жүзеге асыру керек.
17. `pvector` класы сияқты объектіге нұсқауыштарды есте сақтайтын, бірақ оның қандай объектілері векторға жататынын пайдаланушының өзіне шешуге мүмкіндік беретін механизмді қарастыратын `ownership_vector` класын анықтаңыз (яғни, қандай объектілері деструктормен жойылады). Көмекші мәлімет: егер 13-тарауды есіңізге түсірсеңіз, бұл өте қарапайым жаттығу болып табылады.
18. `Vector` класы үшін мүмкін болатын диапазонның шегінен шығып кетуді тексеретін итераторды анықтаңыз (еркін бағытты итератор).
19. `list` класы үшін мүмкін болатын диапазонның шегінен шығып кетуді тексеретін итераторды анықтаңыз (екібағытты итератор).
20. `vector` және `list` кластарымен жұмыс істеуге кететін уақытты салыстыруға арналған тәжірибе (эксперимент) жасаңыз. Программа жұмысының ұзақтығын өлшеу тәсілі 26.6.1 бөлімінде жазылған. `[0:N)`

диапазонынан кездейсоқ бүтін сандар шығарып алыңыз (генерациялаңыз). Әрбір шыққан санды `vector<int>` векторына кірістіріп қойыңыз (әрбір сан қойылған сайын ол бір элементке артып отырады). `Vector` класының объектісін реттеліп орналасқан түрде сақтаңыз; басқаша айтқанда, алдыңғы барлық мәндері одан кіші немесе тең болуы тиіс, ал барлық келесі мәндері одан үлкен болатындай етіп қойылуы керек. Осы тәжірибені бүтін сандарды сақтайтын `list<int>` класын пайдаланып тағы орындаңыз. `N`-нің қандай мәндерінде `list` класы `vector` класына қарағанда жылдам жұмыс істей алады? Тәжірибе нәтижелерін түсіндіруге тырысыңыз. Бұл тәжірибені жасауды алғаш рет Джон Бентли (John Bentley) ұсынған.

СОҢҒЫ СӨЗ

Егер бізде мәліметтері бар контейнерлердің `N` түрі болса және оларға қолданылатын операциялар саны `M` болса, онда біз кодтың `N*M` фрагментін оңай жазып шығар едік. Егер мәліметтер әртүрлі `K` типтерде болса, онда бізге кодтың `N*M*K` фрагментін жазуға тура келер еді. STL кітапханасы бұл мәселені элементтің типін параметр ретінде (`K` көбейткішін алып тастап) беруге рұқсат бере отырып және мәліметтерге қол жеткізуді алгоритмдерден бөле отырып шешеді. Кез келген контейнерде және кез келген алгоритмде мәліметтерге қол жеткізу үшін итераторларды пайдалана отырып, біз алгоритмдердің `N+M` санымен шектеле аламыз. Бұл өте үлкен жеңілдік. Мысалы, егер бізде 12 контейнер және 60 алгоритм болса, онда тікелей әрекет ету тәсілі 720 функция құруды талап етер еді, осымен қатар STL кітапханасында қабылданған стратегия итераторлардың тек 60 функциясы мен 12 анықтамасын қолдануды талап етеді; осылайша біз жұмыстың 90%-ын үнемдейміз. Бұған қоса, STL кітапханасында алгоритмдердің анықтамасына қатысты, дұрыс жұмыс істейтін кодты құруды қарапайым етіп, оның басқа кодтармен байланысуын (композициясын) жеңілдететін келісімдер қабылданған, олар да уақытты көп үнемдейді.

Алгоритмдер мен ассоциативті жиымдар

"Теория бойынша практика қарапайым".

–Тригве Рийнскауг (Trygve Reenskaug)

Осы тарауда біз STL кітапханасының негізінде жатқан идеялардың сипаттамасын және оның мүмкіншіліктерін шолуды аяқтаймыз. Мұнда біз өзіміздің назарымызды алгоритмдерге аударамыз. Біздің негізгі мақсатымыз – оқырмандарды, олардың, егер айы болмаса күнін, жұмыстарын үнемдейтін көптеген пайдалы алгоритмдермен таныстыру. Әрбір алгоритмді сипаттау оны пайдалану мысалдарымен және олардың жұмысын қамтамасыз ететін программалау технологиясын көрсетумен сүйемелденеді. Екінші мақсат – оқырмандардың талаптарын стандартты немесе басқа да қолжетімді кітапханалар қанағаттандыра алмайтын жағдайларда, оларға қысқаша әрі тиімді өз алгоритмдерін жазуды үйрету. Оған қоса, біз тағы **`map`**, **`set`** және **`unordered_map`** тәрізді үш контейнерді қарастырамыз.

21.1 Стандартты кітапхана алгоритмдері

Стандартты кітапхана алпыс шақты алгоритмдерден тұрады. Олардың әрқайсысы кейде бір іс-әрекеттерімен пайдалы; біз назарымызды жиі қолданылатын, көпшілік пайдаланатын, сол сияқты белгілі бір есепті шығаруға өте пайдалы болып табылатын алгоритмдерге аударамыз:

Таңдамалы стандартты алгоритмдер	
`r=find(b,e,v)`	`r` итераторы `v` элементінің `[b:e)` тізбегінің алғашқы кіруіне сілтеме жасайды.
`r=find_if(b,e,p)`	Егер `p(x)` предикаты `true` мәніне тең болса, `r` итераторы x элементінің `[b:e)` тізбегіне алғашқы кіруіне сілтеме жасайды.
`x=count(b,e,v)`	`x` – бұл `v` элементінің `[b:e)` диапазонына кіру саны.
`x=count_if(b,e,p)`	`x` – бұл `p(x)` предикаты `true` мәніне тең болатын кездегі [b:e) тізбегіндегі элементтер саны.
`sort(b, e)`	`<` операторы арқылы `[b:e)` тізбегін реттейді.
`sort(b, e, p)`	`[b:e)` тізбегін `p` предикаты арқылы реттейді.
`copy(b, e, b2)`	`[b:e)` тізбегін `[b2:b2+(e-b))` тізбегіне көшіреді; `b2` итераторынан кейінгі ұяшықтар саны көшіруге жеткілікті болуы керек.

Таңдамалы стандартты алгоритмдер (*жалғасы*)	
`unique_copy(b,e,b2)`	`[b:e)` тізбегін `[b2:b2+(e-b))` тізбегіне көшіреді; қатар тұрған дубликаттар (сыбайлас ұяшықтардағы бірдей мәліметтер) ескерілмейді.
`merge(b,e,b2,e2,r)`	Екі реттелген `[b2:e2)` және `[b:e)` тізбектерін `[r:r+(e-b)+(e2-b2))` тізбегіне біріктіреді.
`r=equal_range(b,e,v)`	`r` – бұл `v` мәні кіретін реттелген `[b:e)` диапазонындағы ішкі тізбек, негізінде, бұл `v` элементін бинарлық іздеу болып табылады.
`equal(b,e,b2)`	`[b:e)` және `[b2:b2+(e-b))` тізбектерінің барлық элементтерінің тең екендігін тексереді.
`x=accumulate(b,e,i)`	`x` – бұл `i` саны мен `[b:e)` тізбегі элементтерінің қосындысы.
`x=accumulate(b,e,i, op)`	Басқа `accumulate` алгоритмдеріне ұқсас, бірақ қосындысы op операциясы арқылы есептеледі.
`x=inner_product(b,e,b2,i)`	`x – [b:e)` және `[b2:b2+(e-b))`
`x=inner_product(b,e, b2,i,op,op2)`	Басқа `inner_product` алгоритмдеріне ұқсас, бірақ `+` және `*` операцияларының орнына `op` және `op2` операцияларын пайдаланады

Келісім бойынша теңдікті тексеру `==` операторы арқылы, ал реттеу – `<` (кіші) операторының негізінде орындалады. Стандартты кітапхананың алгоритмдері `<algorithm>` тақырыптық файлында анықталған. Оқырмандар толық ақпаратты Б.5 қосымшасынан және де 20.7 бөлімінде көрсетілген мәлімет көздерінен таба алады. Бұл алгоритмдер бір немесе екі тізбекпен жұмыс істейді.

Кіру тізбегі итераторлар жұбымен, нәтижелік тізбек – оның бірінші элементіне орнатылған итератормен анықталады. Алгоритм көбінесе, бір немесе бірнеше операциялармен параметрленеді, оларды объект-функциялар арқылы немесе функцияның өзімен анықтауға болады. Алгоритмдер әдетте, кіру тізбегінің соңына орнатылған итераторды қайтара отырып, ақаулар жайлы хабар береді. Мысалы, `find(b,e,v)` алгоритмі, егер ол `v` мәнін таппаса, `e` элементін қайтарады.

21.2 `find()` қарапайым алгоритмі

Пайдалы алгоритмдердің арасындағы ең қарапайым алгоритм find() алгоритмі болуы мүмкін. Ол тізбектің мәні берілген элементін табады:

```
template<class In, class T>
In find(In first. In last, const T& val)
//[first,last) тізбегіндегі val-ға тең бірінші элементті табады
{
      while (first!=last && *first != val) ++first;
      return first;
}
```

find() алгоритмінің анықтамасына қарайық. Әрине, оның қалай жүзеге асырылғанын білмей-ақ, **find()** алгоритмін қолдануға болады, – іс жүзінде біз оны тіпті қолданып та қойдық (мысалы, 20.6.2 бөлімінде).

Дегенмен, **find()** алгоритмінің анықтамасы көптеген пайдалы жобалық идеяларды бейнелей алады, сондықтан ол оқып үйренуге тұрарлық нәрсе болып саналады.

Алдымен, **find()** алгоритмі итераторлар жұбымен анықталған тізбекке қолданылады. Біз val мәнін жартылай ашық **[first:last)** тізбегінен іздейміз. **find()** функциясынан қайтарылатын нәтиже итератор болып табылады. Ол мәні **val**-ға тең тізбектің бірінші элементіне немесе **last** элементіне нұсқау жасайды. Тізбектің соңғы элементінен кейін тұрған элементке итераторды қайтару – ең көп тараған тәсіл, осы арқылы STL кітапханасының алгоритмдері элементтің табылмағанын хабарлайды. Сонымен, біз **find()** алгоритмін келесі түрде пайдалана аламыз:

```
void f(vector<int>& vf int x)
{
      vector<int>::iterator p = find(v.begin(),v.end(),x);
      if (p!.v.end()) {
            // біз v-дағы x-ті таптық
      }
      else {
            // v-да x-ке тең элемент жоқ
      }
      //…
}
```

Осы мысалда, көп жағдайлардағы сияқты, тізбек құрамында контейнердің барлық элементтері болады (вектордың осы жағдайында). Біз іздеп отырған элементтің табылғанын білу үшін тізбектің соңы мен қайтарылған итераторды салыстырамыз.

Енді біз **find()** алгоритмін, сонымен қатар осы келісімдерге негізделген, ұқсас алгоритмдер тобы қалай қолданылатынын білеміз. Дегенмен, басқа алгоритмдерге көшу алдында, **find()** алгоритмінің анықтамасына тереңірек назар аударайық:

```
template<class In, class T>
In find(In first, In last, const T& val)
      // [first,last) тізбегіндегі
      // val-ға тең бірінші элементті табады
{
      while (first!=last && *first!=val) ++first;
      return first;
}
```

Осы цикл толығынан қарапайым деп ойлайсыз ба? Біз олай ойламаймыз. Шын мәнінде, бұл іргелі алгоритмді ең шағын, тиімді және тікелей түрде көрсету болып табылады. Дегенмен, біз бірнеше мысал қарастырмайынша, бұған қосыла салуға болмайды. Алгоритмнің бірнеше нұсқасын салыстырайық:

```
template<class In, class T>
In find(In first, In last, const T& val)
      // [first,last) тізбегіндегі
      // val-ға тең бірінші элементті табады
{
for (In p = first; p!=last; ++p)
      if (*p == val) return p;
return last;
}
```

Осы екі анықтама логикалық түрде эквивалентті және жақсы компилятор екеуі үшін бірдей код генерациялайды. Бірақ, іс жүзінде, артық айнымалыны (`p`) алып тастауға және барлық тексерулер бір жерде ғана орындалатындай етіп кодты қайта құруға көптеген компиляторлардың шамалары жете бермейді. Бұл не үшін қажет? Жекелеп алғанда, `find()` алгоритмінің бірінші (ұсынылған) нұсқасының стилі кең таралып кеткен және біз оны бөтен программаларды оқу үшін түсінуіміз керек, оған қоса көлемді мәліметтермен жұмыс жасайтын шағын функциялар үшін үлкен мәндер тиімді болып келеді.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Осы екі анықтаманың логикалық түрде эквивалентті болатынына сіз сенімдісіз бе? Неліктен? Олардың эквивалентті болатынын көрсететін аргументтер келтіріңіз. Сонан соң екі алгоритмді де бір мәліметтер тобы үшін қолданып көріңіз. Компьютерлік ғылымдардың атақты маманы Дон Кнут ((Don Knuth) бірде былай деген: "Мен тек алгоритмнің дұрыс екенін дәлелдедім, бірақ мен оны тексерген жоқпын". Тіпті математикалық дәлелдемелердің өздерінде де қателер кетіп жатады. Өзіңіздің дұрыс екендігіңізге көз жеткізгіңіз келсе, онда дәлелдемелерге қоса тесттен өткізу нәтижелерінің де болғаны жөн.

21.2.1 Жалпылама алгоритмдерді пайдалану мысалдары

find() жалпылама алгоритм, яғни ол бірсыпыра кодтар үшін ортақ пайдаланылатын алгоритм, мысалы, оны мәліметтердің әртүрлі типтеріне қолдануға болады. Негізінде оның жалпылама табиғаты екі түрлі сипаттан тұрады:

- STL кітапханасы стилінде **find()** алгоритмін кез келген тізбекке қолдануға болады.
- **find()** алгоритмін элементтердің кез келген типіне қолдануға болады.

Бірнеше мысалдар қарастырамыз (егер олар сіздерге күрделі болып көрінсе, онда 20.4 бөліміндегі диаграммаларға қараңыз):

```
void f(vector<int>& v, int x)
//бүтінсанды векторлармен жұмыс жасайды
{
     vector<int>::iterator p=find(v.begin(),v.end(),x);
     if (p!=v.end()) { /* біз x-ті таптық */ }
     //…
}
```

Мұндағы **find()** алгоритмінде қолданылған итераторлармен орындалатын операциялар **vector<int>::iterator** типіндегі итераторлармен орындалатын операциялар болып табылады; яғни **++** операторы (**++first** өрнегіндегі) нұсқауышты келесі жады ұяшығына жылжытады (вектордың келесі элементі сақталатын жерде), ал ***** операциясы (***first** өрнегіндегі) осы нұсқауышты атаусыз етеді. Итераторларды салыстыру (**first!=last** өрнегіндегі) нұсқауыштарды салыстыруға, ал мәндерді салыстыру (***first != val** өрнегіндегі) – қарапайым бүтін сандарды салыстыруға алып келеді.

Алгоритмді **list** класының объектісіне қолданып көрейік:

```
void f(list<string>&v,string x)
// сөз тіркестері тізімімен жұмыс істейді
{
     list<string>::iterator p = find (v.begin(),v.end(),x);
     if (p!=v.end()) { /* біз x-ті таптық */ }
     //...
}
```

Мұндағы **find()** алгоритміндегі итераторларға қолданылған операциялар **list<string>::iterator** класының итераторларымен орындалатын операциялар болып табылады. Бұл операторлардың өздеріне сәйкес мағыналары бар, сондықтан олардың жұмыс істеу логикасы алдыңғы мысалдағы операторлардың

(`vector<int>` класы үшін) жұмыс істеу логикасымен бірдей болып келеді. Сонымен қатар олар мүлде басқаша түрде іске асырылған; басқаша айтқанда, `++` операторы (`++first` өрнегінде) тізімнің келесі түйініне орнатылған нұсқауыштан кейін тұрады, ал `*` операторы (`*first` өрнегінде) `Link` түйініндегі мәнді табады. Итераторларды салыстыру (`first!=last` өрнегінде) `Link*` класының нұсқауыштарын салыстыруға, ал мәндерді салыстыру `(*first!=val` өрнегінде) `string` класындағы `!=` операторы арқылы сөз тіркестерін салыстыруды білдіреді.

Сонымен, `find()` алгоритмі өте икемді болып шықты: егер біз итераторлармен жұмыс жасайтын қарапайым ережелерді сақтайтын болсақ, онда `find()` алгоритмін кез келген контейнердің кез келген тізбегіндегі элементтерді іздеу үшін қолдана аламыз. Мысалы, `find()` алгоритмінің көмегімен 20.6 бөлімінде анықталған `Document` класының объектісіндегі символдарды іздей аламыз.

```
void f (Document& v, char x)
// Document класының объектілерімен жұмыс істейді
{
     Text_iterator p = find(v.begin(),v.end(),x);
     if (p!=v.end()) { /* біз x-ті таптық */ }
     //...
}
```

Бұл икемділік STL кітапханасы алгоритмдерінің айырықша белгісі болып табылады және ол алгоритмдерді адамдардардың көбісі ойлағаннан гөрі пайдалырақ етіп көрсетеді.

21.3 `find_if()` әмбебап іздеу алгоритмі

Біздің кез келген бір нақты мәнді іздеуіміз сирек кездеседі. Көбінесе біз белгілі бір критерийлерді қанағаттандыратын мәндер жиі қызықтырады. Егер біз өзіміздің жеке іздеу критерийлерімізді өзіміз құрастырсақ, онда `find` операциясын анағұрлым пайдалы түрде орындауымызға болар еді. Мысалы, біз 42-ден артық сандарды іздеп тауып алар едік. Және де регистрді есепке алмай-ақ (жоғарғы немесе төменгі), сөз тіркестерін салыстыруымызға болар еді. Оған қоса, алғашқы тұрған тақ санды да таба алар едік. Мүмкін, біздің `"17 Cherry Tree Lane"` деген адресі бар жазбаны тапқымыз келген шығар.

Қолданушының өзі берген критериіне сәйкес стандартты іздеу алгоритмі `find_if()` деп аталады:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
```

```
{
    while (first!=last && !pred(*first)) ++first;
    return first;
}
```

Әрине (егер бастапқы кодтарды салыстырсақ), ол **find()** алгоритміне ұқсас болып келеді, бірақ онда ***first!=val** шарты емес, **!pred(*first)** шарты қолданылады; басқаша айтқанда, алгоритм іздеуді берілген мәні бар элемент табылған кезде емес, тек **pred()** предикаты ақиқат болған жағдайда ғана тоқтатады.

Предикат (**predicate**) – бұл **true** немесе **false** мәнін қайтаратын функция. Әрине, **find_if()** алгоритмі **pred(*first)** өрнегі дұрыс болуы үшін, бір аргумент қабылдай алатын предикатты талап етеді. Біз еш қиындықсыз мәннің берілген бір қасиетін тексеретін предикат жаза аламыз, мысалы, "сөз тіркесінде **x** әрпі кездесе ме", "санның мәні 42-ден артық па" немесе "сан тақ болып табыла ма?" Мысалы, біз бүтін сандардан тұратын вектордағы алғашқы тақ санды таба аламыз.

```
bool odd(int x) {return x%2;}     // % - модуль бойынша бөлу

void f (vector<int>& v)
{
    vector<int>::iterator p = find_if(v.begin(),v.end(),odd);
    if (p!=v.end()) { /* біз тақ санды таптық */ }
    //…
}
```

Осы шақыру кезінде **find_if()** алгоритмі **odd()** функциясын, алғашқы тақ санды тапқанша, әрбір элементке қолданады. Осы сияқты, біз тізімдегі мәні 42-ден асатын алғашқы элементті де таба аламыз.

```
bool larger_than_42(double x) { return x>42; }

void f(list<double>&v)
{
  list<double>::iterator p=find_if(v.begin(),v.end(),larger_than_42);
  if (p!=v.end()) { /* біз 42-ден асатын мәнді таптық */ }
  //...
}
```

Дегенмен, соңғы мысал толығымен қанағаттандырмайды. Егер біз осыдан кейін 41-ден асатын элементті тапқымыз келсе, қайтеміз? Онда бізге жаңа функция жазу керек шығар. 19-дан үлкен элементті тапқыңыз келе ме? Тағы бір функция жазыңыз. Бұдан гөрі ыңғайлы тәсіл болуы тиіс!

Егер біз элементті **v**-нің кез келген мәнімен салыстырғымыз келсе, онда бұл мәнді **find_if()** алгоритмі предикатының мәні тікелей көрсетілмеген аргументі етіп жасауымыз керек. Біз оны жасап көрейік (идентификатордың қолайлы аты ретінде **v_val** тіркесін алдық).

```
double v_val;
// larger_than_v() предикатының өз аргументімен
// салыстырылатын мән
bool larger_than_v(int x) { return x>v_val; }

void f(list<double>&v, int x)
{
   v_val = 31;
   // келесі larger_than_v предикатын шақыру үшін
   // v_val айнымалысын 31-ге тең етіп аламыз
   list<double>::iterator p = find_if(v.begin(), v.end(), larger_than_v);
   if (p!=v.end()) { /* біз 31-ден асатын мәнді таптық */ }

   v_val = x;
   // келесі larger_than_v предикатын шақыру үшін
   // v_val айнымалысын x-ке тең етіп аламыз
   list<double>::iterator q=find_if (v.begin(),v.end(),larger_than_v);
   if (q!=v.end()) {/* біз x-тен асатын мәнді таптық */}

    //…
}
```

Неткен сұмдық! Осындай программаны жазған адамдар өз істегеніне қарай сыбағасын алар деген сенімдеміз, бірақ біз осы кодты қолданушыларға және кез келген адамға алдын ала жанымыз ашиды. Қайталап айтайық: бұдан гөрі қолайлы тәсіл болуы тиіс!

МЫНАНЫ ЖАСАП КӨРІҢІЗ

v айнымалысын осылай пайдалану неліктен біздің жүрегімізді айнытады? Аз дегенде, түсініксіз қателіктерге алып баратын үш тәсілді атаңыз. Осындай программаларға ерекше жол бермейтін үш қосымшаны атаңыз.

21.4 Объект-функциялар

Сонымен, біз предикатты **find_if()** алгоритміне бергіміз келеді және осы предикат элементтерді оның аргументі ретінде берілген мәндермен салыстырып отырса дейміз. Жекелеп айтар болсақ, біз шамамен мынадай код жазғымыз келеді:

```
void f(list<double>&v, int x)
{
    list<double>::iterator p =
    find_if(v.begin(),v.end(),Larger_than(31));
    if (p!=v.end()) { /* біз 31-ден асатын санды таптық */ }

    list<double>::iterator
                q = find_if(v.begin(),v.end(),Larger_than(x));
    if (q!=v.end()) { /* біз x-тен асатын санды таптық */ }
    //. . .
}
```

Әрине, **Larger_than** функциясы екі шартты қанағаттандыруы керек:

- оны предикат ретінде, мысалы **pred(*first)** түрінде шақыруға болады.
- ол шақыру кезінде берілетін мәнді, мысалы **31** немесе **x**, есте сақтауы мүмкін.

Осы шарттарды орындау үшін, бізге өзін функция сияқты ұстайтын объект, яғни объект-функция қажет. Бізге объект қажет, өйткені осы объектілер ғана мәліметтерді есте сақтай алады, мысалы, салыстыру үшін қажетті мәндер сақталуы тиіс. Мысал қарастырайық:

```
class Larger_than {
    int v;
public:
    Larger_than(int vv) : v(vv) { }   // аргументті сақтайды
    bool operator()(int x) const{ return x>v;} //салыстыру
};
```

Осы анықтама біз предикаттан нені талап етсек, соны көрсететінін атап өткен жөн. Енді оның қалай жұмыс істейтінін түсіну қалды. **Larger_than(31)** өрнегін жазып, біз **v** мүшесінде 31 санын сақтайтын **Larger_than** класының объектісін құрамыз. Мысал қарастырайық.

```
find_if(v.begin(),v.end(),Larger_than(31))
```

Мұнда біз **Larger_than(31)** объектісін **find_if()** алгоритміне оның параметрі ретінде **pred** атымен береміз. Әрбір **v** элементі үшін **find_if()** алгоритмі **pred(*first)** шақырылуын жүзеге асырады.

Бұл функцияны шақыру операторын екпінді етеді, яғни ***first** аргументі бар объект-функция үшін **operator()** функция-мүшесін іске қосады. Осының нәтижесінде элементтің мәндерін салыстыру жүзеге асады, яғни ***first** 31 санымен салыстырылатын болады.

Біз бұдан функцияны шақыруды, кез келген басқа операторға ұқсас, **()** операторы жұмысының нәтижесі ретінде қарастыруға болатынын көреміз. **()** операторын *функцияны шақыру операторы* (function call operator) немесе *қолданбалы оператор* (application operator) деп те атайды. Сонымен, **v[i]** өрнегіндегі **[]** операторының **vector::operator[]** операторына эквивалентті болғаны сияқты, **pred(*first)** өрнегіндегі **()** операторы **Larger_than::operator()** операторына эквивалентті (парапар) болып табылады.

21.4.1. Функция-объектілерге абстрактілік көзқарас

Осылайша, бізде функцияға өзіне қажетті мәліметтерді сақтауға мүмкіндік беретін механизм бар. Әрине, функция-объектілер әмбебап, қуатты және ыңғайлы механизмді құратыны анық. Функция-объект ұғымын толығырақ қарастырайық:

```
class F {    // объект-функцияның абстрактілі мысалы
    S s;           // жағдайы
public:
    F(const S& ss) :s(ss) { /* бастапқы мәнді орнатады */ }
    T operator() (const S& ss) const
    {
      // ss аргументімен бір нәрсе жасайды
      // T типіндегі мән қайтарады (көбінесе T – бұл void,
      // bool немесе S)
    }

    const S& state() const { return s; }
    // жағдайды көрсетеді
    void reset(const S& ss) { s = ss; }
    // жағдайды қалпына келтіреді
};
```

F класының объектісі мәліметтерді өзінің **s** мүшесінде сақтайды. Қажетінше объект-функцияның көптеген мәлімет-мүшелері болуы мүмкін. Кейде "бірдеңе мәліметтерді сақтайды" сөзінің орнына "бірдеңе жағдайында тұр" деп айтылады. Біз **F** класының объектісін құрған кезде, біз осы жағдайды инициалдауымызға болады. Қажет болса біз бұл жағдайды оқи да аламыз. **F** класында жағдайды оқу үшін **state()** операциясы, ал жағдайды жазу үшін – **reset()** операциясы

қарастырылған. Дегенмен, объект-функцияны құру кезінде, біз оның жағдайына қол жеткізу тәсілін таңдауға еріктіміз.

Әрине, біз қарапайым белгілеу жүйесін пайдалана отырып, объект-функцияны тікелей немесе жанамалы түрде шақыра аламыз. Оны шақыру кезінде `F` объект-функциясы бір аргумент алады, бірақ біз қанша параметр қажет болса, сонша параметрлерді ала алатын объект-функцияны да анықтай аламыз.

Объект-функцияларды пайдалану STL кітапханасындағы параметрлеудің негізгі тәсілі болып табылады. Біз объект-функцияларды іздеу алгоритмінде нақты нені іздеп отырғанымызды көрсету үшін пайдаланамыз (21.3 бөлімін қ.). Ол сұрыптау критерийлерін анықтау үшін (21.4.2 бөлімі), сандық алгоритмдердегі арифметикалық операцияларды көрсету үшін (21.5 бөлімі), біз қандай объектілерді тең деп санайтынымызды көрсету үшін (21.8 бөлімі) және де басқа көптеген мақсаттар үшін де пайдаланылуы мүмкін. Объект-функцияларды пайдалану – алгоритмдердің икемділігі мен әмбебаптығының негізгі қайнар көзі болып табылады.

Көбінесе объект-функцияларды қолдану өте тиімді болады. Жекелеп алатын болсақ, шаблондық функцияның аргументі ретінде шағын функцияны мәні бойынша беру өте тиімді жұмыс өнімділігін қамтамасыз етеді. Мұның себебі қарапайым, бірақ ол аргумент ретінде функцияны беру механизмін жақсы білетін адамдар үшін таңқаларлық: әдетте функцияны объект ретінде беру, функцияның өзін беру кезіндегіге қарағанда, қысқашалау және тез жұмыс істейтін код жазуға алып келеді. Бұл тұжырым келесі жағдайлар орындалғанда, ақиқат болып табылады, олар: егер объект шағын ғана болса (мысалы, егер ол бір-екі сөзден ғана тұратын мәліметтерден құралса немесе ешқандай да мәліметтер сақтамаса) немесе сілтеме арқылы берілетін болса, сонымен қатар функцияны шақыру операторы шағын болып (мысалы, `<` операторының көмегімен қарапайым салыстыру), ол құрамдас функция (`inline function` – подставляемая функция) сияқты (мысалы, егер оның анықтамасы класс ішінде болса) анықталған жағдайларда ақиқат болып табылады.

Осы тараудағы мысалдардың көбі, және жалпы кітаптағы мысалдар – осы ережеге сәйкес түрде құрастырылған. Шағын және қарапайым объект-функциялар жұмыс өнімділігінің жоғары болуының негізгі себебі – олар компиляторға типтер туралы тиімді код жасауға (генерациялауға) жеткілікті ақпарат көлемін ұсынады. Тіпті онша күрделі емес оптимизаторлары бар ескірген компиляторлардың өздері функцияны шақырудың орнына, `Larger_than` класындағы салыстыру операциясы үшін қарапайым ғана "үлкен" деген (>) машиналық нұсқауды жылдам жасай алады. Функцияны шақыру, қарапайым салыстыру операциясына қарағанда, әдетте, 10-50 есе ұзақ орындалады. Бұған қоса, функцияны шақыру коды қарапайым салыстыру кодына қарағанда көлемді болып табылады.

21.4.2 Класс мүшелеріндегі предикаттар

Біз бұрын көргеніміздей, стандартты алгоритмдер **int** және **double** сияқты тізбек элементтерінің базалық типтерімен жақсы жұмыс жасайды. Кейбір пәндік аймақтарда қолданушы класы объектілерінің контейнерлері кең қолданылады. Көптеген аймақтарда басты рөл атқаратын мысалды – жазбаны бірнеше критерийлері бойынша сұрыптау әрекетін қарастырайық.

```
struct Record {
    string name;    // стандартты сөз тіркесі
    char addr[24];  // мәліметтер базасымен келісім үшін
                    // ескі стиль
    //…
};

vector<Record> vr;
```

Біз **vr** векторын кейде аты бойынша, ал кейде – адресі бойынша сұрыптағымыз келеді. Егер біз сәнділік пен тиімділікке қатар ұмтылмасақ, онда біздің тәсілдеріміз практикалық қажеттілікпен шектеледі. Біз келесі кодты жаза аламыз:

```
//...
sort(vr.begin(),vr.end(),Cmp_by_name());
//аты бойынша сұрыптау
//...
sort(vr.begin(),vr.end(),Cmp_by_addr());
//адресі бойынша сұрыптау
//...
```

Cmp_by_name – бұл **Record** класының екі объектісін **name** мүшелері бойынша салыстыратын объект-функция. Қолданушыға салыстыру критериін беруге мүмкіндік беру үшін, **sort** стандартты алгоритмінде сұрыптау критериін көрсететін міндетті емес үшінші аргумент қарастырылған. **Cmp_by_name** функциясы sort алгоритмі үшін **Cmp_by_name** объектісін құрып, оны **Record** класының объектілерін салыстыру үшін пайдаланады. Бұл жөнінде бізге мазасыздандудың қажеті жоқ болғандықтан, бұл өте жақсы болып көрінеді. Біздің ендігі істейтініміз – **Cmp_by_name** және **Cmp_by_addr** кластарын анықтап алуымыз керек.

```
// Record класының объектілерін әртүрлі салыстыру жолдары:

struct Cmp_by_name {
   bool operator()(const Record& a,const Record& b) const
     { return a.name < b.name;}
};
```

```
struct Cmp_by_addr {
   bool operator()(const Record& a, const Record& b) const
     {return strncmp(a.addr, b.addr, 24)<0; }     // !!!
};
```

`Cmp_by_name` класының жұмыс айқын. `operator()` функциясын шақырудың `()` операторы `name` сөз тіркесін, `string` стандартты класының `<` операторын пайдалану арқылы салыстырады. Дегенмен, `Cmp_by_addr` класындағы салыстыру келеңсіз болып көрінеді. Бұл біздің адресті 24 символдан тұратын жиым ретінде (және нөлмен аяқталмайтын) сәтсіз бейнелеуімізбен түсіндіріледі. Бұл таңдауды біз бір жағынан, объект-функцияны қателерге онша қарсы тұра алмайтын әлсіз кодтарды жасыру үшін пайдалану үшін жасадық, екінші жағынан STL кітапханасының тәжірибелік тұрғыдан маңызды, бірақ келеңсіз есептерді де шығара алатынын көрсету үшін жасадық. Салыстыру функциясы ұзындығы бекітілген символдар жиымдарын салыстыратын стандартты `strncmp()` функциясын қолдана отырып, екінші тіркес лексикографиялық түрде біріншісінен артық болғанда, теріс сан қайтарады. Егер сізге осындай ескірген салыстыруды жүзеге асыру керек болса, онда осы функцияны есіңізге алыңыз (мысалы, Б. 10.3 бөлімін қ.).

21.5 Сандық алгоритмдер

STL кітапханасының көптеген стандартты алгоритмдері мәліметтерді өңдеумен байланысты болып келеді: олар ақпараттарды көшіреді, сұрыптайды, олардың арасындағы мәндерді іздеуді орындайды және т.с.с. Соған қоса олардың кейбіреулері есептеулерді орындауға арналған. Олар нақты бір есептерді шығару үшін де және STL кітапханасындағы сандық алгоритмдерді жүзеге асырудың жалпы принциптерін көрсету үшін және пайдалы болуы мүмкін. Осындай алгоритмдердің төрт түрі бар:

Сандық алгоритмдер	
`x=accumulate(b,e,i)`	Тізбектерді қосындылайды; мысалы, {a,b,c,d} тізбегінің нәтижесі a+b+c+d-ға тең болады. `x` нәтижесінің типі `i`-дің бастапқы мәнінің типімен бірдей болып келеді.
`x=inner_product(b,e,b2,i)`	Екі тізбектің сәйкесінше алынған мәндерінің жұбын көбейтеді де, нәтижелерін қосындылайды; мысалы, {a,b,c,d} және {e,f, g,h} тізбектері үшін есептеу нәтижесі a*e+b*f+c*g+d*h болады. `x` нәтижесінің типі `i`-дің бастапқы мәнінің типімен бірдей болады.
`r=partial_sum(b,e,r)`	Берілген тізбектің алғашқы n элементінің қосындысынан тұратын тізбек құрады; мысалы, {a,b,c,d} тізбегі үшін оның нәтижесі {a, a+b, a+b+c, a+b+c+d) болады.
`r=adjacent_difference(b,e,b2,r)`	Берілген тізбек элементтерінің арасындағы айырмалардан тұратын тізбек құрады; мысалы, {a,b,c,d} тізбегі үшін оның нәтижесі {a,b-a,c-b,d-c} болады.

Бұл алгоритмдер **<numeric>** тақырыптық файлында анықталған. Біз олардың алғашқы екеуін сипаттаймыз, ал қалғанын оқырмандар қажет болған кезде өз бетімен оқып үйрене алады.

21.5.1 accumulate( ) алгоритмі

Қарапайым және анағұрлым пайдалы сандық алгоритмдердің бірі accumulate() алгоритмі болып табылады. Қарапайым нұсқада ол тізбекке жататын мәндерді қосындылайды.

```
template<class In,class T> T accumulate(In first,In last,T init)
{
     while (first!=last) {
        init = init + *first;
        ++first;
     }
     return init;
}
```

Бастапқы **init** мәнін алған соң, ол оған **[first:last)** тізбегіндегі әрбір мәнді қосып отырады да, қосындыны қайтарады. Қосынды жинақталатын **init** айнымалысын көбінесе *аккумулятор* (accumulator) деп атайды. Мысал қарастырайық:

```
int a[] = { 1, 2, 3, 4, 5 };
cout << accumulate(a, a+sizeof(a)/sizeof(int), 0);
```

Бұл кодтың фрагменті экранға 15 санын шығарады, яғни 0+1+2+3+4+5 (0 бастапқы мән болып табылады). Әрине, **accumulate()** алгоритмін тізбектердің барлық түрлеріне қолдануға болады:

```
void f(vector<double>& vd,int* p,int n)
{
    double sum = accumulate(vd.begin(), vd.endO, 0.0);
    int sum2 = accumulate(p,p+n,0);
}
```

Нәтиженің (қосынды) типі **accumulate()** алгоритмі аккумулятор ретінде қолданатын айнымалының типімен бірдей болады. Бұл мұндағы маңызды рөл атқаратын икемділіктің жоғары дәрежесін қамтамасыз етеді. Мысал қарастырайық:

```
void f(int* p, int n)
{
    int s1 a accumulate(p,p+n,0);
    // бүтін сандарды int типіне қосындылаймыз
    long s1=accumulate(p,p+n,long(0));
    // бүтін сандарды long типіне қосындылау
    double s2=accumulate(p,p+n,0.0);
    // бүтін сандарды double типіне  қосындылау
}
```

Кейбір компьютерлерде **int** типіндегі айнымалыға қарағанда, **long** типіндегі айнымалы анағұрлым үлкен цифрлар санынан тұрады. **double** типіндегі айнымалы **int** типіндегі айнымалыға қарағанда үлкен (кішісіні де) сандарды көрсете алады, бірақ олардың дәлдігі төмен болуы мүмкін. 24-тарауда біз диапазондар және есептеулер дәлдігі туралы сұраққа тағы да қайтып ораламыз.

init айнымалысын аккумулятор ретінде пайдалану аккумулятордың типін беруге мүмкіндік беретін кең таралған идиома болып саналады:

```
void f(vector<double>& vd, int* p, int n)
{
    double s1 = 0;
    s1 = accumulate(vd.begin(),vd.end(),s1);
    int s2 = accumulate (vd.begin(),vd.end(),s2); // ой
    float s3 = 0;
    accumulate(vd.begin(),vd.end(),s3);           // ой
}
```

Аккумуляторды инициалдап, **accumulate()** алгоритмі жұмысының нәтижесін қандай да бір айнымалыға меншіктеуді ұмытпаңыз. Осы мысалда инициализатор ретінде, алгоритмді шақырғанға дейін бастапқы мәнін алмаған, **s2** айнымалысы қолданылды; осындай шақырудың нәтижесін алдын ала болжау қиын болады. Біз **s3** айнымалысын **accumulate()** алгоритміне бердік (мәні бойынша, 8.5.3 бөлімін қ.), бірақ нәтижені ештеңеге меншіктемедік; осындай компиляция уақыттың босқа кеткенін көрсетеді.

21.5.2 accumulate() алгоритмін жалпылау

Сонымен, негізгі **accumulate()** алгоритмі үш аргумент арқылы қосындыны анықтайды. Дегенмен басқа да көптеген пайдалы амалдар, мысалы, тізбектерге қолдануға болатын көбейту және азайту операциялары бар. Сондықтан STL кітапханасында **accumulate()** алгоритмінің орындалатын операцияны да көрсетуге мүмкіндік беретін төрт аргумент арқылы жұмыс істейтін нұсқасы қарастырылған:

```
template<class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp op)
{
     while (first!=last) {
        init = op(init, *first);
        ++first;
     }
     return init;
}
```

Мұнда екі аргумент қабылдап алатын кез келген бинарлық операцияны пайдалануға болады, аргументтер типі аккумулятордың типімен сәйкес келеді. Мысал қарастырайық:

```
array<double,4> a = {1.1,2.2,3.3,4.4}; // 20.9 бөлімін қараңыз
cout<<accumulate(a.begin(),a.end(),1.0,multiplies<double>());
```

Осы кодтың фрагменті баспаға 35.1384 санын шығарады, яғни ол 1.0*1.1*2.2*3.3*4.4 (бастапқы мәні – 1.0). Аргумент ретінде берілетін **multip-lies<double>()** бинарлық операторы көбейтуді орындайтын стандартты объект-функция болып табылады; **multiplies<double>** объект-функциясы **double** типіндегі сандарды көбейтеді, **multiplies<int>** объект-функциясы **int** типіндегі сандарды көбейтеді және т.с.с.

Басқа да бинарлық объект-функциялар бар: **plus** (қосу), **minus** (азайту), **divides** және **modulus** (бөлудің қалдығын есептеу). Олардың барлығы **<functional>** (Б.6.2 бөлімі) тақырыптық файлында анықталған.

Жылжымалы нүктелі сандарды көбейту үшін оның бастапқы мәні **1.0** саны болатынына назар аударыңыз.

sort() алгоритміндегі (21.4.2 бөлімін қараңыз) мысал сияқты, бізді көбінесе құрамдас типтердің қалыпты мәліметтері емес, класс объектілерінде сақталатын мәліметтер қызықтырады. Мысалы, біз тауар бірліктерінің құнын және жалпы санын біле отырып, оның жалпы құнын есептей алар едік:

```
struct Record {
     double unit_price;
     int units; // сатылған бірліктер саны
     // . . .
};
```

Біз **accumulate** алгоритмінің анықтамасындағы кез келген бір операторға, **Record** класының сәйкес элементінен **units** мәліметтерін алып, оны аккумулятор мәніне көбейту тапсырмасын бере аламыз:

```
double price(double v, const Record& r)
{
     return v + r.unit_price * r.units;
     // бағасын есептеп нәтижесін  жинақтайды
}

void f(const vector<Record>& vr)
{
     double total = accumulate(vr.begin(),vr.end(),0.0,price);
     // . . .
}
```

Біз ерініп, бағаны есептеу үшін объект-функцияны емес, функцияны пайдаланып, осылай да жасауға болатынын көрсеттік. Бірақ сонда да біз келесі жағдайлар үшін:

- егер шақырулар арасында мәліметтерді сақтау қажет болғанда,
- егер оларды құрамдас (inlining) деп хабарлауға болатындай соншалықты қысқа болса (тым болмағанда, кейбір өте қарапайым операциялар үшін), онда осыларға объект-функцияны қолдануды ұсынамыз.

Біз мұнда жоғарыдағы тізімнің екінші пунктін басшылыққа алып, объект-функцияны пайдалана алар едік.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Жоғарыда көрсетілген функцияларды қолдана отырып, `vector<Record>` класын анықтаңыз да, оны өзіңіз таңдап алған төрт жазбамен инициалдап, жалпы құнды есептеңіз.

21.5.3 `inner_product` алгоритмі

Екі вектор алып, оның сәйкес элементтерін қос-қостан көбейтіп, шыққан көбейтінділерді қосыңыз. Осы есептеулердің нәтижесі екі вектордың *скалярлық көбейтіндісі* (inner product) деп аталады және ол көптеген аймақтарда кеңінен қолданылатын операциялар болып табылады (мысалы, физикада және сызықтық алгебрада; 24.6 бөлім).

Егер сіз сөзден гөрі программаны артық көрсеңіз, онда STL кітапханасындағы осы алгоритмнің мына нұсқасын оқып көріңіз.

```
template<class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init)
```

```
// ескерту: екі вектордың скалярлық көбейтіндісін есептейді
{
    while(first!=last) {
        init = init+(*first)*(*first2);
        //элементтерді көбейтеміз
        ++first;
        ++first2;
    }
    return init;
}
```

Алгоритмнің бұл нұсқасы кез келген элементтер типінен тұратын тізбектің кез келген түрі үшін скалярлық көбейтінді түсінігін ортақ етіп жалпылайды. Мысал ретінде биржалық индексті қарастырайық. Ол компанияларға белгілі бір салмақты меншіктеу арқылы есептеледі. Мысалы, Доу-Джонстің Alcoa индексі кітапты жазу кезінде 2,4808 мәнін құрады. Индекстің ағымдағы мәнін анықтау үшін, әрбір компанияның акция бағасын оның салмағына көбейтіп, алынған нәтижелерді қосамыз. Әрине, мұндай индекс баға мен салмақтың скалярлық көбейтіндісін көрсетеді. Мысал қарастырайық:

```
// Доу-Джонс индексін есептеу
vector<double> dow_price;
// әрбір компанияның акция құны
dow_price.push_back(81.86);
dow_price.push_back(34.69);
dow_price.pueh_back(54.45);
//. . .

list<double> dow_weight;
// индекстегі әрбір компанияның салмағы
dow_weight.push_back(5.8549);
dow_weight.push_back(2.4808);
dow_weight.push_back(3.8940);
//. . .

double dji_index = inner_product(
//(weight,value) жұптарын көбейтеміз және қосамыз
        dow_price.begin(),
        dow_price.end(),
        dow_weight.begin(), 0.0);

cout<< "DJI мәні " << dji_index << '\n';
```

`inner_product()` алгоритмінің екі тізбекті алатынына назар аударыңыз. Сонымен қатар ол тек үш аргументті алады: екінші тізбектің тек басы ғана беріледі. Екінші тізбек біріншісінен аз болмайтын элементтер санынан тұрады деп есептеледі. Қарсы жағдайда, біз программаның орындалуы кезінде қате туралы хабарлама аламыз. `inner_product()` алгоритмінде екінші тізбек біріншісінен гөрі саны көп элементтерден тұруы мүмкін; мұндайда артық элементтері пайдаланылмайды.

Екі тізбектің де бірдей типте болуы немесе типтері бірдей элементтерден құралуы міндетті емес. Осы тұжырымды бейнелеп көрсету үшін, біз бағаларын `vector` класының объектісіне, ал салмақтарын – `list` класының объектісіне жаздық.

21.5.4 `inner_product()` алгоритмін жалпылау

`inner_product()` алгоритмін `accumulate()` алгоритмі сияқты жалпылауға болады. Бірақ алдыңғы жалпылаудан айырмашылығы `inner_product()` алгоритміне тағы екі аргумент қажет: бірінші – аккумуляторды `accumulate()` алгоритміндегі сияқты жаңа мәнімен байланыстыру үшін, ал екіншісі – мәндер жұбымен байланыстыру үшін керек:

```
Template<class In,class In2,class T,class BinOp,
    class BinOp2 >
T inner_product(In first,In last,In2 first2,
    T init,BinOp op, Bin0p2 op2)
{
     while(first!=last) {
        init = op(init,op2(*first,*first2));
        ++first;
        ++first2;
     }
return init;
}
```

21.6.3 бөлімінде Доу-Джонс индексі мысалына қайта ораламыз да, `inner_product()` алгоритмінің жалпыланған нұсқасын есепті шешудің сәнді бөлігі ретінде қолданамыз.

21.6 Ассоциативті контейнерлер

`vector` класынан кейін қолдану жиілігі жағынан екінші орында тұрған `map` стандартты контейнері болып табылады. Ол жұпталған (кілт, мән) түрдегі

реттелген тізбек күйінде болады да, керекті мәнді кілт бойынша табуға мүмкіндік береді; мысалы, `my_phone_book["Nicholas"]` элементі Николастың телефон нөмірі болуы мүмкін. Кең таралуы жағынан map класының жалғыз ғана лайықты бәсекелесі болып `unordered_map` класы есептеледі (21.6.4 бөлімін қ.), ол сөз тіркестерін көрсететін кілттер үшін оңтайландырылып жасалған. `map` және `unordered_map` контейнерлеріне ұқсас мәліметтер құрылымдары әртүрлі аттармен белгілі болып жүр, мысалы, *ассоциативті жиымдар* (associative arrays), *хеш-кестелер* (hash tables) және *қызыл-қара бұтақтар* (red-black trees). Кең таралған әрі пайдалы түсініктердің әрқашанда көптеген атаулары болады. Біз олардың барлығын да *ассоциативті контейнерлер* (associative containers) деп атаймыз.

Стандартты кітапханада сегіз ассоциативті контейнерлер қарастырылған:

Ассоциативті контейнер	
`map`	Жұптардың (кілт, мән) реттелген контейнері
`set`	Кілттердің реттелген контейнері
`unordered_map`	Жұптардың (кілт, мән) реттелмеген контейнері
`unordered_set`	Кілттердің реттелмеген контейнері
`multimap`	Кілті бірнеше рет кездесетін `map` контейнері
`multiset`	Кілті бірнеше рет кездесетін `set` контейнері
`unordered_multimap`	Кілті бірнеше рет кездесетін `unordered_map` контейнері
`unordered_multiset`	Кілті бірнеше рет кездесетін `unordered_ set` контейнері

Бұл контейнерлер `<map>`, `<set>`, `<unordered_map>` және `<unordered_set>` тақырыптық файлдарында анықталған.

21.6.1. Ассоциативті жиымдар

Қарапайым есеп қарастырамыз: сөздердің мәтінге ену нөмірлері тізімін құрайық. Бұл үшін сөздер тізімін мәтінде кіру санымен бірге жазған жөн. Жаңа сөзді оқи отырып, біз оның бұрын кездескен/кездеспегенін тексеріп отырамыз; егер кездеспесе, сөзді тізімге енгізіп қойып, оны 1 санымен байланыстырамыз. Бұл үшін `list` немесе `vector` типіндегі объектіні қолдануға болар еді, бірақ біз онда әрбір оқылған сөзді іздеп отыруымыз керек. Мұндай шешім өте баяу болар еді. `map` класы өзінің кілттерін, егер олар кездесетін болса, жеңіл көріп алу үшін сақтайды. Мұндайда іздеу қарапайым есеп түрінде болады.

```
int main ()
{
    map<string,int> words;
    // жұптарды (сөз, жиілік) сақтайды
```

```
    string s;
    while(cin>>s) ++words[s];
    //words контейнері сөз тіркестерімен индекстеледі

    typedef map<string,int>::const_iterator Iter;
    for (Iter p = words.begin(); p!=words.end(); ++p)
      cout<<p->first<< ":" << p->second << '\n';
}
```

Осы программаның ең қызық бөлігі болып `++words[s]` өрнегі табылады. `main()` функциясының бірінші жолында көрініп тұрғандай, `words` айнымалысы – жұптардан `(string,int)` тұратын `map` класының объектісі; яғни `words` контейнері `string` тіркестерін `int` бүтін сан түрінде көрсетеді. Басқаша айтқанда, `words` контейнерінде `string` класының объектісі болса, ол `int` типіндегі сәйкес санға қол жеткізу мүмкіндігін береді. Сонымен, біз `string` класының объектісімен `words` контейнерін индекстеген кезде (енгізу ағымынан оқылатын сөздерден тұратын), `words[s]` элементі `s` тіркесіне сәйкес `int` типіндегі санға сілтеме болып табылады. Нақты мысал қарастырайық:

```
words["sultan"]
```

Егер `"sultan"` сөз тіркесі әлі кездеспесе, онда ол int типі үшін, келісім бойынша берілген мәнімен, `words` контейнеріне қойылады, яғни 0 енгізіледі. Енді `words` контейнерінде `("sultan",0)` элементі бар. Сол себепті, егер `"sultan"` тіркесі бұдан бұрын енгізілмеген болса, онда `++words["sultan"]` өрнегі `"sultan"` сөз тіркесімен `1` санын байланыстырады. Анығырақ айтқанда, `map` класының объектісі `"sultan"` тіркесінің онда жоқ екенін анықтап, оған `("sultan",0)` жұбын енгізіп қояды, сонан соң `++` операторы оның мәнін бірге арттырады, соның нәтижесінде ол `1`-ге тең болады.

Программаны тағы бір рет талдайық: `++words[s]` өрнегі енгізу ағымынан бір сөзді алып, оның мәнін бірге арттырады. Бірінші енгізу кезінде әрбір сөз 1 мәнін алады. Енді циклдың мағынасы түсінікті болды:

```
while (cin>>s) ++words[s];
```

Ол әрбір сөзді (босорындармен бөлінген) енгізу ағымынан оқып алады да, оның контейнерге кіру санын есептейді. Енді бізге нәтижені шығару жеткілікті. map контейнері бойынша орын ауыстырып ығысу `STL` кітапханасының кез келген басқа контейнерлеріндегідей түрде орындалады. Контейнердің `map<string, int>` элементтері `pair<string, int>` типінде болады. `pair` класы объектісінің бірінші мүшесі `first` болып, екіншісі – `second` болып аталады. Мәлімет шығару циклы төмендегідей түрде болады:

```
typedef map<string,int>::const_iterator Iter;
for(Iter p=words.begin(); p!=words.end(); ++p)
      cout << p->first << ": " << p->second <<'\n';
```

typedef операторы (20.5 және A.16 бөлімдерін қ.) жұмыстың қолайлылығын қамтамасыз ету мен программаның ыңғайлы оқылуы үшін тағайындалған.

Біз программаға мәтін ретінде *The C++ Programming Language* кітабының бірінші басылымындағы кіріспе сөзді енгіздік:

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C *programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types.*

Программа жұмысының нәтижесі төменде көрсетілген:

```
C: 1
C++; 3
C, : 1
Except: 1
In: 1
a: 2
addition: 1
and: 1
by: 1
defining: 1
designed: 1
details,: 1
efficient: 1
enjoyable: 1
facilities: 2
flexible: 1
for: 3
general: 1
is: 2
language: 1
language.: 1
make: 1
minor: 1
more: 1
new: 1
of: 1
programmer.: 1
```

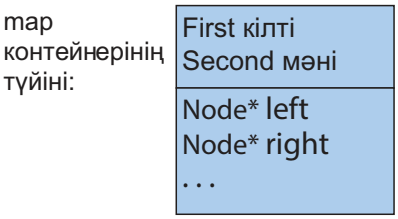
```
programming: 3
provided: 1
provides: 1
purpose: 1
serious: 1
superset: 1
the: 3
to: 2
types.: 1
```

Егер сіз жоғарғы немесе төменгі регистрлер әріптерінің арасына айырмашылық жасағыңыз келмесе немесе пунктуациялық белгілерді есепке алғыңыз келсе, онда осы есепті де шығаруға болады: 13-жаттығуды қараңыз.

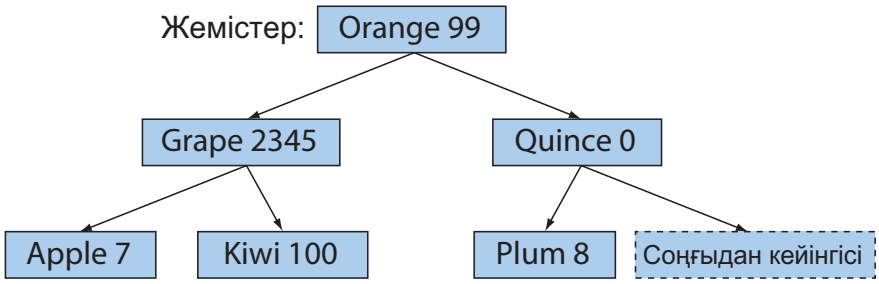
21.6.2. Ассоциативті жиымдарға шолу

Сонымен `map` контейнері дегеніміз не? Ассоциативті жиымдарды жүзеге асырудың тәсілдері көп, бірақ олар `STL` кітапханасында теңгерілген бұтақтар негізінде жүзеге асырылды. Анығырақ айтсақ, олар қызыл-қара бұтақтар түрінде көрсетілген. Біз оның нақтылықтарына назар аудармаймыз, бірақ осы техникалық терминдер сізге таныс болғандықтан, олар туралы түсініктемелерді әдебиеттен немесе веб-парақтардан таба аласыз.

Бұтақ түйіндерден тұрады (тізімнің түйіндерден тұратыны сияқты; 20.4 бөлімін қ.). `Node` класының объектісінде кілт, соған сәйкес сан және келесі екі түйінге нұсқауыштар сақталады:



Егер біз компьютер жадындағы `map<Fruit,int>` класының объектісіне төмендегі жұптарды кірістіріп қоятын болсақ (Kiwi, 100), (Quince,0), (Plum,8), (Apple,7), (Grape,2345) және (Orange,99), ол мынадай болып көрінер еді:



Кілт **Node** класының **first** атты мүшесінде сақталатындықтан, іздеудің бинарлық бұтағын ұйымдастырудың негізгі ережесі мынадай болады:

```
left->first<first && first<right->first
```

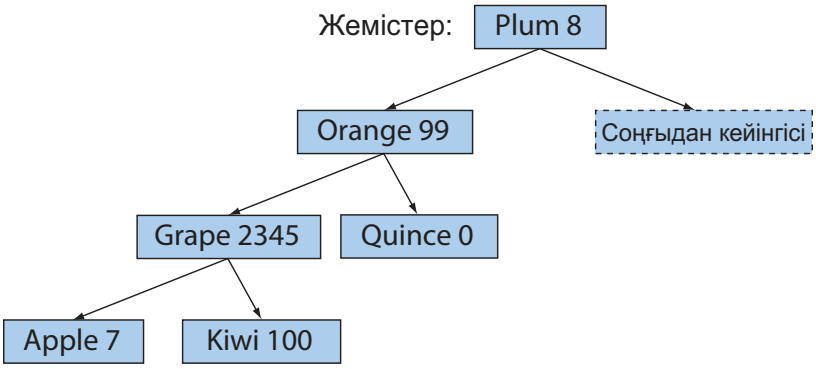
Басқаша айтқанда, әрбір түйін үшін екі шарт орындалады:

- сол жақ ішкі түйіннің кілті түйіннің кілтінен кіші.
- түйіннің кілті оң жақ ішкі түйіннен кіші.

Бұл шарттар бұтақтың әрбір түйіні үшін орындалатынына көз жеткізуіңізге болады. Бұл бізге іздеуді ағаштың түбінен бастап бұтақтан төмен қарай орындауға мүмкіндік береді. Компьютерлік ғылым туралы әдебиеттерде бұтақтардың төменге қарай өсетіні қызық нәрсе. Түпкі түйін мәліметтер жұбынан (Orange, 99) тұратын түйін болып табылады. Біз қолайлы орын тапқанша, бұтақтармен төмен қарай жылжимыз. Егер (жоғарыда көрсетілген мысалдағы сияқты) ағаштың әрбір ішкі бұтағындағы түйіндер саны ағаш түбінен сол бұтақпен бірдей қашықтықта орналасқан басқа ішкі бұтақтардың түйіндері санымен шамамен бірдей болатын болса, мұндай бұтақ *теңгерілген* (balanced) деп аталады. Теңгерілген бұтақта берілген түйінге жеткенге дейінгі біз жүріп өтетін түйіндердің орташа саны ең кіші (минималды) деңгейде болады.

Түйінде теңгерілімді сүйемелдеу үшін контейнер пайдаланатын қосымша мәліметтер сақталуы мүмкін. Егер әрбір түйіннің сол жағындағы және оң жағындағы мұрагерлер саны шамамен бірдей болса, онда бұтақ теңгерілген болып саналады. Егер N түйіннен тұратын бұтақ теңгерілген болса, онда түйінді тауып алу үшін log_2N мәнінен аспайтын түйіндерден өту қажет. Бұл, егер кілттер тізімде сақталып, іздеу басынан бастап орындалатын болса, біз өтуге тиіс орташа $N/2$ түйіннен гөрі анағұрлым жақсы (өйткені сызықтық іздеудің ең төменгі деңгейінің өзінде бізге N түйіннен өтуіміз керек болар еді). (21.6.4 бөлімін қ.).

Мысал үшін теңгерілмеген бұтақтың қалай бейнеленетінін көрсетейік:



Бұл бұтақ бұрынғыдай, әрбір түйіннің кілті сол жақ ішкі түйіннің кілтінен үлкен және оң жақ ішкі түйіннің кілтінен кіші деп талап ететін критерийлерді қанағаттандырады:

```
left->first<first && first<right->first
```

Дегенмен бұл бұтақ теңгерілмеген болып табылады, сондықтан бізге теңгерілген бұтақтағы сияқты, Apple және Kiwi түйіндерін табу үшін, екі өтпелі аралықтың орнына үш аралықтан өтуіміз керек болады. Көптеген түйіндерден тұратын бұтақтар үшін бұл айырмашылық елеулі болуы мүмкін, сондықтан **map** контейнерлерін жүзеге асыру үшін теңгерілген бұтақтар қолданылады.

map контейнерін жүзеге асыру үшін қолданылатын бұтақтарды ұйымдастыру қағидаларын талқылау міндетті емес. Кәсіби мамандар, тым болмағанда, олардың жұмыс қағидаларын біледі деп жорамалдау жеткілікті шығар. Бізге бар керегі – бұл стандартты кітапханадағы **map** класының интерфейсі. Төменде оның біршама қысқартылған нұсқасы келтірілген:

```
template<class Key,class Value,
                        class Cmp = less<Key> > class map
{
   // . . .
   typedef pair<Key,Value> value_type;
   // map контейнері (Key,Value) жұптарын сақтайды

   typedef sometype1 iterator; // бұтақтың түйініне нұсқауыш
   typedef sometype2 const_iterator;

   iterator begin();  // бірінші элементке нұсқайды
   iterator end();//соңғы элементтен кейінгі элементке нұсқайды

   Value& operator[](const Key& k);
                  // k айнымалысы бойынша индекстеу
   iterator find(const Key& k);    // k кілті бойынша іздеу
```

```
   void erase(iterator p);
   // p итераторы нұсқайтын элементті жою
   pair<iterator,bool> insert(const value_type&);
   // (key,value) жұбын енгізу
   // . . .
};
```

Контейнердің осы нұсқасы **<map>** тақырыптық файлында анықталған. Итераторды **Node*** нұсқауышы ретінде көрсетуге болады, бірақ итераторды жүзеге асыру кезінде оны нақты бір типте деп есептеуге болмайды.

vector және **list** кластарының (20.5 және B.4 бөлімдерін қ.) интерфейсі ұқсас екендігі анық көрініп тұр. Негізгі айырмашылығы, контейнермен орын ауыстырып жылжу кезінде енді оның элементтері болып **pair<Key,value>** типіндегі жұптар саналады. Бұл тип STL кітапханасында өте пайдалы болып табылады:

```
   template<class T1, class T2> struct pair {
      typedef T1 first_type;
      typedef T2 second_type;
      T1 first;
      T2 second;

      pair() :first(T1()), second(T2()) { }
      pair(const T1& x, const T2& y):first(x), second(y) { }
      template<class U, class V>
      pair(const pair<U,V>& p):first(p.first), second (p. second) { }
   };
   template<class T1, class T2>
   pair<T1,T2> make_pair(T1 x, T2 y)
   {
      return pair<T1,T2>(x,y);
   }
```

Біз стандарттан **pair** класының толық анықтамасы мен оның пайдалы **make_pair()** көмекші функциясын көшіріп алдық.

map контейнері бойынша орын ауыстырып жылжу кезінде, оның элементтері кілтпен анықталған тәртіппен теріліп алынады. Мысалы, егер біз мысалда сипатталған контейнер бойынша жылжитын болсақ, онда келесі реттілікті аламыз:

```
(Apple,7) (Grape,2345) (Kiwi,100) (Orange,99) (Plum,8) (Quince,0)
```

Түйіндерді енгізіп қою реттілігі маңызды емес.

insert() операциясы түсініксіз мән қайтарады, әдетте оны біз қарапайым программаларда көбінесе есепке алмаймыз. Егер берілген жұп (кілт, мән)

`insert()` функциясын шақыру арқылы қойылған болса, бұл (кілт, мән) жұбына орнатылған итератордан және `bool` типіндегі айнымалыдан тұратын, true мәнін қабылдайтын жұп болып саналады. Егер кілт контейнерде болған болса, онда кірістіріп қою есепке алынбайды да, `bool` типті мән `false` мәнін қабылдайды.

Біз ассоциативті жиымды қарастыру реттілігін үшінші аргументтің (`map` класында хабарланған `Cmp` предикаты) көмегімен анықтай аламыз. Мысал қарастырайық:

```
map<string, double, No_case> m;
```

`No_case` предикаты символдарды салыстыруды регистрді есепке алмай отырып анықтайды (21.8 бөлімін қ.). Келісім бойынша айналу реттілігі `less<Key>` предикатымен, яғни "кем" деген қатынаспен анықталады.

21.6.3 Ассоциативті жиымның тағы бір мысалы

`map` контейнерінің пайдалылығын бағалау үшін, 21.5.3 бөліміндегі Доу-Джонс индексі мысалына қайтып оралайық. Егер ондағы барлық салмақтар `vector` класының объектілеріндегі өз аттарына сәйкес позицияларда жазылса, сипатталған код дұрыс жұмыс жасайды. Бұл талап тікелей сипатта емес, сондықтан ол түсініксіздеу қателер көзі болып кетуі мүмкін. Бұл мәселені шешудің көптеген тәсілдері бар, бірақ солардың ішіндегі тартымдысы – барлық салмақтарды өздерінің тикерімен бірге, мысалы ("AA",2.4808) түрінде, сақтау болып табылады. Тикер – бұл компания атының қысқартылған атауы. Осы сияқты етіп, компанияның тикері мен оның акция бағасын да бірге сақтауға болады, мысалы ("AA",34.69) түрінде. Қорытындылай келе, АҚШ-тың нарық қорымен сирек кездесетін адамдар үшін, біз тикерді компания атауымен бірге жазып сақтай аламыз, мысалы ("AA","Alcoa Inc."); басқаша айтқанда, осыларға сәйкес мәндерді үш ассоциативті жиымындарда сақтай аламыз.

Алдымен (символ, баға) жұбынан тұратын ассоциативті контейнер құрайық:

```
map<string,double> dow_price;
// Доу-Джонс индексі (символ, баға);
// ағымдағы котировкаларды www.djindexes.com веб-сайтынан қараңыз
dow_price["MMM"] = 81.86;
dow_price ["AA"] = 34.69;
dow_price ["MO"] = 54.45;
//...
```

(символ, салмақ) жұбынан тұратын ассоциативті жиым былай хабарланады:

```
map<string,double> dow_weight;
// Доу-Джонс индексі (символ,салмақ)
```

```
dow_weight.insert(makepair("MMM", 5.8549));
dow_weight.insert(make_pair("AA",2.4808));
dow_weight.insert(make_j>air("MO",3.8940));
//...
```

Біз негізінде, **map** контейнерінің элементтері болып, **pair** класының объектілері табылатынын көрсету үшін **insert()** және **make_pair()** функцияларын пайдаландық. Бұл мысал мәндерді белгілеуді де көрсете алады; біз индекстеу түсініктірек және – маңыздылығы аз болса да – ол оңай жазуға болады деп санаймыз.

(Символ, аты) жұбынан тұратын ассоциативті контейнер:

```
map<string, string> dow_name;  //Доу-Джонс (символ, аты)
dowjiameCMMM"] = "3M Co.";
dow_name["AA"] = "Alcoa Inc.";
dow_name["MO"] = "Altria Group Inc.";
//...
```

Бұл ассоциативті контейнерлер арқылы кез келген ақпаратты оңай алып алуға болады. Мысал қарастырайық:

```
double alcoa_price = dow_price ["AAA"];
// ассоциативті жиымнан мәндерді оқимыз
double boeing_price = dow_price ["BA"];

if (dow_price.find("INTC")!=dow_jprice.end())
// ассоциативті жиымның элементін табамыз
    cout << "Intel is in the Dow\n";
```

Ассоциативті жиым бойынша орын ауыстырып жылжу оңай. Біз кілт – **first**, ал мәні – **second** деп аталатынын есімізде сақтауымыз қажет.

```
typedef map<string,double>::const_iterator Dow_iterator;

// Доу-Джонс индексіне кіретін әрбір компанияның
// акциясының бағасын жазады:
for (Dow_iterator p = dow_price.begin(); p!=dow_price.end(); ++p)
{
    const string& symbol = p->first;    // тикер
        cout << symbol << '\t'
        <<p->second << '\t'
        << dow_name[symbol] << '\n';
}
```

Біз кейбір есептеулерді тікелей ассоциативті контейнерлерді пайдалана отырып орындай аламыз. Мысалы, 21.5.3 бөліміндегі сияқты индексті есептей аламыз. Біз акция бағасы мен салмақты соларға сәйкес ассоциативті жиымнан алып, оларды көбейтуіміз қажет. Осы есептеулерді кез келген екі ассоциативті жиымдармен `map<string, double>` орындайтын функцияны оңай жазуға болады:

```
double weighted_value(
     const pair<string,double>& a,
     const pair<string,double>& b
           )    // мәндерді алады да, көбейтеді
{
     return a.second * b.second;
}
```

Енді жай ғана осы функцияны `inner_product()` алгоритмінің жалпыланған нұсқасына қойып, индекстің мәнін аламыз:

```
double dji_index =
    inner_product(dow_price.begin(),dow_price.end(),
    //барлық компаниялар
      dow_weight.begin(),   // олардың салмағы
      0.0,                  // бастапқы мәні
      plus<double>(),       // қосынды (қарапайым)
      weighted_value);      // мәнін және салмағын алады да,
                            // сонан соң оларды көбейтеді
```

Неліктен осындай мәліметтерді векторларда емес, ассоциативті жиымдарда сақтау қолайлы болады? Біз `map` класын әртүрлі мәндердің арасындағы байланыс айқын болу үшін қолдандық. Бұл себептердің бірі. Оған қоса, `map` контейнері элементтерді олардың кілті арқылы анықталған реттілікпен сақтайды. Мысалы, `dow` контейнерін қарастыру кезінде біз символдарды алфавиттік тәртіппен шығардық; егер де біз `vector` класын пайдаланса, онда оны сұрыптауға мәжбүр болатын едік. Көбінесе `map` класын мәндерді кілттері бойынша іздеу үшін қолданады. Реттелген құрылымдағы `map` контейнеріне қарағанда, `find()` алгоритмінің көмегімен элементтерді іздеу ірі тізбектер үшін анағұрлым баяу орындалады.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Осы мысалды жұмыс жағдайына келтіріңіз. Сонан соң өзіңіздің қалауыңыз бойынша бірнеше компанияны қосыңыз да, олардың салмақтарын енгізіңіз.

21.6.4 unordered_map() алгоритмі

vector контейнеріндегі элементті табу үшін **find()** алгоритмі, вектордың бірінші элементінен бастап, ізделініп отырған элементке немесе вектордың соңғы элементіне дейін қарап барып аяқтап, барлық элементтерді тексеруі керек. Осы іздеудің орташа күрделілігі вектордың *(N)* ұзындығына пропорционал болады; мұндай жағдайда алгоритмнің күрделілігі *O(N)* деп айтылады.

map контейнеріндегі элементті табу үшін индекстеу операторы ағаштың тамырынан бастап, ізделіп отырған мәнмен немесе бұтақ жапырағымен аяқтап, барлық элементтерді тексеріп шығуы керек. Осы іздеудің орташа күрделілігі бұтақтың тереңдігіне пропорционал. *N* элементтен тұратын теңгерілген бинарлық бұтақтың максималды тереңдігі log_2N-ге тең, ал ондағы іздеу күрделілігінің дәрежесі $O(log_2N)$ шамасында болады, яғни ол log_2N мәніне пропорционал. Бұл *O(N)*-ге қарағанда, анағұрлым жақсы:

N	15	128	1023	16 383
log_2N	4	7	10	14

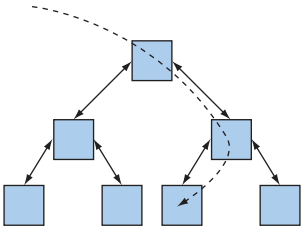
Іздеудің нақты күрделілігі, бізге ізделінетін мәндерді неғұрлым тез табумен және салыстыруды орындайтын операцияларға кететін шығындармен байланысты болады. Әдетте, нұсқауыштар ізімен жүріп өту (**map** контейнері ішінен іздеу кезінде) нұсқауышты инкременттеуден гөрі (**find()** алгоритмі арқылы **vector** контейнері ішінде іздеу жасаған кезде) күрделірек болып табылады.

Кейбір типтер үшін, әсіресе бүтін сандар мен символдық тіркестер үшін, **map** контейнерінің бұтақтары бойынша іздеуге қарағанда, өте жоғарғы деңгейге қол жеткізуге болады. Бүге-шігесіне назар аудармай-ақ, біз кілт бойынша **vector** контейнеріндегі индексті есептеуге болатын идеямызды көрсетейік. Бұл индекс *хеш-функцияның (hash value) мәні* деп, ал осы тәсіл қолданылатын контейнер – *хеш-кесте (hash table)* деп аталады. Хеш-кестедегі ұяшықтар санына қарағанда, мұндағы мүмкін болатын кілттер саны анағұрлым көп. Мысалы, хеш-функция көбінесе мыңдаған элементтерден тұратын миллиардтаған сөз тізбектерін вектор индексіне бейнелеу үшін қолданылады. Мұндай есеп күрделі болуы мүмкін, бірақ оны шығаруға болады. Бұл үлкен **map** контейнерлерін жүзеге асыру кезінде аса пайдалы. Хеш-кестенің негізгі артықшылығы – ондағы іздеудің орташа күрделілігі тұрақты болады да, ол элементтер санына тәуелді болмайды, яғни O(1) дәрежесімен анықталады. Бұл, әрине, үлкен ассоциативті жиымдар үшін, елеулі артықшылық, мысалы, 500 мың веб-адрестерден тұратын жиымдарға қолдану өте қолайлы болады. Оқырмандар хеш-іздеу туралы толығырақ ақпаратты **unordered_map** контейнері туралы құжаттамадан (веб желілерінен алып) немесе мәліметтер құрылымы туралы кез келген оқулықтан (мазмұнындағы *хеш-кесте* және *хештеу* деген тақырыптардан іздеңіз) таба алады.

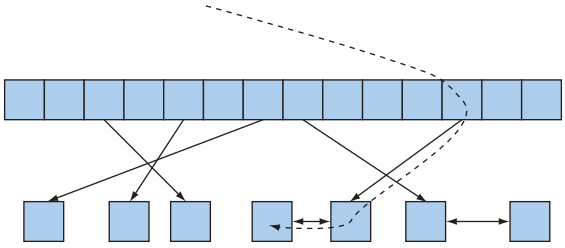
Енді вектордағы (реттелмеген), теңгерілген бинарлық бұтақтағы және хеш-кестедегі іздеу көрсетілімін графикалық түрде қарастырайық:

- Реттелмеген **vector** контейнеріндегі іздеу:

- **map** контейнеріндегі (теңгерілген бинарлық бұтақта) іздеу:



- **unordered_map** контейнеріндегі (хеш-кесте) іздеу:



STL кітапханасының **unordered_map** контейнері хеш-кестенің көмегімен жүзеге асады, **map** контейнері – теңгерілген бинарлық бұтақ негізінде, ал **vector** контейнері – жиым түрінде жүзеге асады. STL кітапханасының пайдалылығы, бір жағынан, мәліметтерді сақтаудың әртүрлі тәсілдерін бір жерге біріктіріп, оларға қол жеткізуге, екінші жағынан, алгоритмдерді де біріктіруге мүмкіндік берді.

Эмпирикалық ереже келесілерді айтады:

- Егер сіздің **vector** контейнерін пайдаланбауға ешқандай негізіңіз болмаса, оны қолданыңыз.

- Егер сізге мәні бойынша іздеуді орындау керек болса, онда **map** контейнерін қолданыңыз (және де егер кілттің типі "кіші" операциясын тиімді орындауға мүмкіндік берсе).

- Егер сізге үлкен ассоциативті жиымда жиі мәлімет іздеу керек болса және сізге оның реттелген түрі қажет болмаса, онда **unordered_map** контейнерін қолданыңыз (және егер сіздің кілтіңіздің типі хеш-функцияны тиімді қолдануға мүмкіндік беретін болса).

Біз **unordered_map** контейнерін толығымен сипаттамаймыз. Оны **map** контейнері сияқты **string** немесе **int** типіндегі кілттермен қолдануға болады, тек элементтерді айнала қарастырғанда, олардың реттелмегенінен басқа жағдайлары ғана қарастырылуы тиіс. Мысалы, біз 21.6.3 бөліміндегі Доу-Джонс индексін есептейтін кодтың фрагментін келесідей түрде көшіріп жазуымызға болады:

```
unordered_map<string, double> dow_price;

typedef unordered_map<string, double>::const_iterator Dow_iterator;

for (Dow_iterator p = dow_price.begin(); p!=dow_price.end() ; ++p)
{
    const string& symbol = p->first; //"тикер" символы
        cout << symbol << '\t'
        << p->second << '\t'
        << dow_name[symbol] << '\n';
}
```

Енді **dow** контейнеріндегі іздеуді тезірек орындауға болады. Дегенмен, бұл тездету байқалмай қалуы да мүмкін, өйткені осы индекске тек отыз компания ғана қосылған. Егер де біз Нью-Йорк биржалық қорында жұмыс істейтін (котировка алған) барлық компанияларының акциялары бағасын ескерсек, онда программа жұмысының өнімділігінің айырмашылығын бірден сезер едік. Әзірге біз тек логикалық айырмашылығын ғана атап өтеміз: әрбір итерациядағы мәліметтер алфавиттік тәртіпте көрсетілмейді.

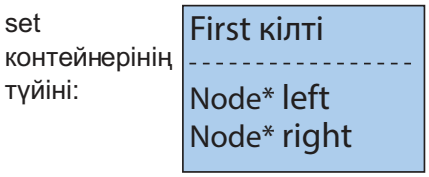
C++ тілінің стандартында реттелмеген ассоциативті жиымдар жаңалық болып табылады, олар әлі оның толық құқылы элементі бола қойған жоқ, өйткені олар стандарттың өз мәтінінде емес, C++ тілін Стандарттау комиссиясының техникалық есеп беруінде (Technical Report) ғана сипатталған. Дегенмен олар кең тараған, олар болмаған жерлерде соған ұқсас аналогтарын кездестіруге болады, мысалы, **hash_map** класы сияқтылар.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

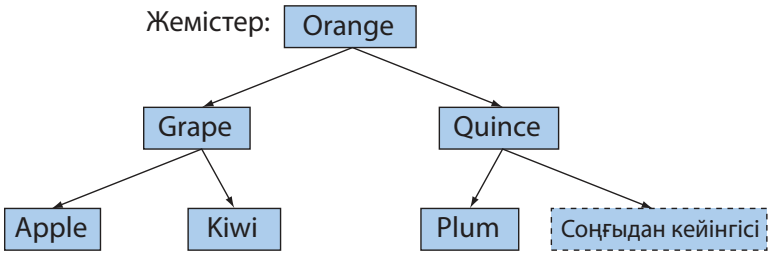
#include<unordered_map> директивасын қолданып шағын программа жазыңыз. Егер ол жұмыс істемесе, онда **unordered_map** класы сіздегі C++ тілінің нұсқасына қосылмаған. Егер сізге, шынымен де, **unordered_map** контейнері қажет болса, онда оның табылған бір қолжетімді нұсқасын веб желісінен алуыңызға болады (мысалы, *www.boost.org* сайтын қ.).

21.6.5 Жиындар

`set` контейнерін мәндері онша маңызды емес немесе мәндері жоқ ассоциативті жиым ретінде көрсетуге болады. `set` контейнерін келесідей түрде бейнелей аламыз:



Мысалы, жемістер көрсетілген `set` контейнерін (21.6.2 бөлімін қ.) келесі түрде бейнелеуге болады:



`set` контейнері несімен пайдалы? Біз қандай да бір мәнді көрдік пе, көрмедік пе, соларды есеп шығару кезінде есімізде сақтайтын көптеген жағдайлар бар екен. Мысалдардың бірі – қолда бар жемістерді атап көрсету (бағасына тәуелді емес); екінші мысал – сөздік құрастыру. Осы контейнерді қолданудың басқа бір тәсілі – элементтері объект болып келетін, көптеген ақпараттан тұратын, ондағы кілттің рөлін олардың бір мүшесі атқаратын "жазбалар" жиыны. Мысал қарастырайық:

```
struct Fruit {
    string name;
    int count;
    double unit_price;
    Date last_sale_date;
    //...
};

struct Fruit_order {
    bool operator()(const Fruit& a, const Fruit& b) const
    {
        return a.name<b.name;
    }
     };
```

```
set<Fruit, Fruit_order> inventory;
//жемістерді салыстыру үшін Fruit_order(x,y)-ті қолданыңыз
```

Мұнда біз тағы да көретініміз, объект-функция STL кітапханасының компоненттері арқылы шығаруға қолайлы есептер ауқымын анағұрлым кеңейтеді.

`set` контейнерінің мәндері болмағандықтан, ол индекстеу операциясын сүйемелдемейді (`operator [] ()`). Олардың орнына біз `insert()` және `erase()` сияқты "тізімдерге қолданылатын операцияларды" қолдануымыз қажет. Өкінішке орай, `map` және `set` контейнерлері мынадай себеп бойынша `push_back()` функциясын қолдамайды: жаңа элементті кірістіріп қою орнын программалаушы емес, `set` контейнері анықтайды. Оның орнына `insert()` функциясын қолданған жөн. Мысал қарастырайық:

```
inventory.insert(Fruit("quince",5));
inventory.insert(Fruit("apple", 200, 0.37));
```

`set` контейнерінің `map` контейнерінен бір артықшылығы біз итератордан алынған мәнді тікелей қолдана аламыз. `set` контейнерінде `map` контейнеріндегі сияқты жұптар (кілт, мән) жоқ болғандықтан, атаусыз ету операторы элементтің мәнін қайтарады:

```
typedef set<Fruit>::const_iterator SI;
for(SI p=inventory.begin(),p!=inventory.end();++p)
     cout<<*p << '\n';
```

Әрине, егер сіз `<<` операторын `Fruit` класы үшін анықтасаңыз, онда осы фрагмент жұмыс істейді.

21.7. Көшіру

21.2 бөлімінде біз `find()` функциясын "қарапайым пайдалы алгоритм" деп атадық. Әрине, осы көзқарасты аргументтеуге де болады. Көптеген қарапайым алгоритмдер пайдалы, тіпті өте қарапайым (тривиалды) болып келеді. Егер біреу жазып, түзетіп ретке келтіріп қойған код болса, несіне жаңа программа жазуымыз керек? Қарапайымдылығы мен пайдалылығы жағынан `copy()` алгоритмі `find()` алгоритмінен әлдеқайда жоғары тұр. STL кітапханасында `copy()` алгоритмінің үш нұсқасы бар:

Copy operations	
`copy(b,e,b2)`	`[b:e)` тізбегін `[b2:b2+(e- b))` тізбегіне көшіреді
`unique_copy(b,e,b2)`	`[b:e)` тізбегін `[b2:b2+(e- b))` тізбегіне сыбайлас көшірмелерін алып тастай отырып көшіреді
`copy_if(b,e,b2,p)`	`[b:e)` тізбегінен `[b2:b2+(e- b))` тізбегіне `p` предикатын қанағаттандыратын элементтерін ғана көшіреді

21.7.1 `copy()` алгоритмі

`copy()` алгоритмінің негізгі нұсқасы келесі түрде анықталған:

```
template<class In,class Out>Out copy(In first,In last,Out res)
{
    while (first!=last) {
        *res = *first; // элементті көшіреді
        ++res;
        ++first;
}
    return res;
}
```

`copy()` алгоритмі, итераторлар жұбын алған соң, итератормен оның бірінші элементіне берілген тізбекті басқа тізбекке көшіреді. Мысал қарастырайық:

```
void f(vector<double>& vd, list<int>& li)
// int типіндегі сандар тізімінің элементтерін
// double типіндегі сандар векторына көшіреді
{
  if (vd.size()<li.size())error("мақсатты контейнер тым шағын");
  copy (li.begin(), li.end(), vd.begin());
  //...
}
```

Кіру тізбегінің типі нәтижелік тізбектің типінен басқаша болуы мүмкін, соған назар аударыңыз. Бұл STL кітапханасының алгоритмдерінің әмбебаптығын арттырады: олар тізбектердің барлық түрлерімен, олардың іске асуына ешқандай артық жорамал жасамай-ақ, жұмыс істейді. Біз мұнда нәтижелік тізбекке енгізілетін элементтерді жазуға арналған орын жеткілікті екендігін тексеруді ұмытқан жоқпыз. Осындай тексеру программалаушының міндетіне кіреді.

STL кітапханасының алгоритмдері жоғары әмбебаптылық пен тиімді жұмыс өнімділігіне қол жеткізу үшін программаланды; келісім бойынша олар диапазонды тексермейді және қолданушыларды қорғайтын басқа тесттерді де орындамайды. Осылар қажет болған сайын, қолданушының өзі осындай тексеруді орындап шығуы тиіс.

21.7.2 Ағындар итераторы

Сіздер "шығару ағынына көшіру" немесе "енгізу ағынынан көшіру" сөздерін жиі еститін боласыздар. Бұл – кейбір енгізу-шығару түрлерін ыңғайлы және пайдалы сипаттау тәсілі. Осы операцияны орындау үшін **copy()** алгоритмі қолданылады.

Тізбектердің қасиеттерін естеріңізге салайық:

- Тізбектің басы мен соңы бар.
- Тізбектің келесі элементіне көшу **++** операторының көмегімен орындалады.
- Тізбек элементінің мәнін * операторы арқылы табуға болады.

Енгізу және шығару ағындарын дәл осылай жеңіл сипаттауға болады. Мысал қарастырайық:

```
ostream_iterator<string> oo(cout);
// *oo ағынын cout ағынымен жазу үшін байланыстырамыз
*oo = "Hello, ";                // яғни cout << "Hello, "
++oo;               // "келесі элементті шығаруға дайын"
*oo = "World! \n";              // яғни cout << "World!\n"
```

Стандартты кітапханада шығару ағындарымен жұмыс жасауға арналған **ostream_iterator** типі бар; **ostream_iterator<T>** – бұл **T** типіндегі мәндерді жазу үшін қолдануға болатын итератор.

Стандартты кітапханада тағы **T** типіндегі мәндерді оқуға арналған **istream_iterator<T>** типі бар:

```
istream_iterator<string> ii(cin);
//*ii-ді оқу – бұл сөз тіркесін cin ағынынан оқу

string s1 = *ii;
//яғни cin>>s1 "келесі элементті енгізуге дайын"
++ii;
string s2 = *ii;          // яғни cin>>s2
```

ostream_iterator және **istream_iterator** итераторларын қолданып, **copy()** алгоритмінің көмегімен мәліметтерді енгізуге және шығаруға болады. Мысалы, асығыс жасалған сөздікті келесі түрде қалыптастыруға болады:

```
int main()
{
   string from, to;
   cin >> from >> to;
   //бастапқы және мақсаттық файлдың атын енгіземіз

   ifstream is(from.c_str ()); //енгізу ағынын ашамыз
   ofstream os(to.c_str());    //шығару ағынын ашамыз

   istream_iterator<string> ii(is);
   //ағыннан енгізу итераторын құрамыз
   istream_iterator<string> eos;
   //енгізудің сигналдық белгісі
   ostream_iterator<string> oo(os,"\n");
   //ағынға шығару итераторын құру

   vector<string> b(ii,eos);
   //b - енгізу ағынының мәліметтерімен инициалданатын вектор
   sort(b.begin() ,b.end());       // буферді сұрыптау
   copy(b.begin() ,b.end() ,oo); // шығару үшін көшіру буфері
}
```

eos итераторы – бұл "енгізу соңын" білдіретін сигналдық белгі. **istream** ағыны енгізудің соңына жеткенде (көбінесе **eof** деп аталатын), оның **istream_iterator** итераторы келісім бойынша берілген **istream_iterator** итераторына тең болады және **eos** деп аталады.

Біздің **vector** класының объектісін итераторлар жұбымен инициалдайтынымызға назар аударыңыз. Контейнерді инициалдайтын **(a,b)** итераторлар жұбы **"[a:b)** тізбегін контейнерге оқу" дегенді білдіреді. Әрине, біз бұл үшін **(ii,eos)** итераторлар жұбын – енгізудің басы мен соңын пайдаландық. Бұл бізге тікелей түрде **>>** операторын және **push_back()** функциясын қолданбауға мүмкіндік береді. Біз мұның баламалы нұсқасын қолдануға ұсыныс бермейміз:

```
vector<string> b(max_size);
//енгізу мәліметтерінің көлемін табуға талпынбаңыз
copy(ii,eos,b.begin());
```

Енгізудің максималды өлшемін анықтағысы келетін адамдар, әдетте оны дұрыс бағалай алмайды да, буферді толтырып жіберіп, өзі үшін де, қолданушылар үшін де бірсыпыра мәселелер туындатады. Буферді толтырып жіберу мәліметтердің сақталуына да қауіпті болып табылады.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Программаны жұмыс істейтін қалыпқа келтіріп, оны шағын, айталық бірнеше жүз сөзден тұратын файл арқылы тесттен өткізіңіз. Сонан соң енгізу мәліметтерінің көлемін табуда "ұсынылмайтын" нұсқаны зерттеңіз де, `b` енгізу буфері толып кеткен кезде не болатынын көріп шығыңыз. Мұндағы ең жаман сценарий болып, өзіңіз еш жамандығын байқамай, программаны қолданушыға беретін нұсқаңыз саналады, осыған назар аударыңыз.

Біздің шағын программамызда біз сөздерді оқимыз, содан кейін оларды реттейміз. Әзірге біздің жасап жатқанымыздың барлығы дұрыс шығар, бірақ біз неге сөздерді "қате" ұяшықтарға жазып, сонан соң оларды сұрыптауға мәжбүр боламыз? Одан да басқа тағы бір жаман жағы, біз сөздерді жазып, баспаға шығарғанда, олар енгізу ағынында неше рет кездессе, сонша рет шығарамыз.

Соңғы мәселені, `copy()` алгоритмінің орнына `unique_copy()` алгоритмін қолдана отырып шешуге болады. `unique_copy()` функциясы қайталанатын бірдей мәндерді көшірмейді. Мысалы, қарапайым `copy()` функциясын шақыру кезінде программа мына жолды енгізеді де,

```
the man bit the dog
```

экранға мына сөздерді шығарады:

```
bit
dog
man
the
the
```

Егер де `unique_copy()` алгоритмін қолдансақ, онда программа келесі сөздерді шығарады:

```
bit
dog
man
the
```

Жаңа жолға өту қайдан шықты? Мәліметтерді ажыратқыш белгілермен шығару сонша кең таралған, `ostream_iterator` класының конструкторы сізге (қажет болғанда) әрбір мәннен кейін шығатын тіркестерді көрсетуге мүмкіндік береді:

```
ostream_iterator<string> oo(os,"\n");
// шығару ағыны үшін итератор құрады
```

Әрине, жаңа жолға көшу – бұл адамдарға нәтижесін талдап көруге мүмкіндік беретін, мәлімет шығару үшін кең таралған таңдау, бірақ, мүмкін, сіз босорындарды қолданғыңыз келетін шығар? Біз келесі кодты жаза алар едік:

```
ostream_iterator<string> oo(os," ");
//шығару ағыны үшін итератор құрады
```

Осы жағдай үшін шығару нәтижесі мынадай болар еді:

```
bit dog man the
```

21.7.3 Тәртіпті сақтау үшін set класын пайдалану

Осылай мәлімет шығаруды орындай алатын анағұрлым қарапайым тәсілі бар: ол **vector**-ды емес, **set** контейнерін қолдану болып табылады.

```
int main()
{
   string from, to;
   cin>>from>>to;// бастапқы және мақсаттық файлдардың аттары

   ifstream is(from.c_str()); // енгізу ағынын құрамыз
   ofstream os(to.c_str());   // шығару ағынын құрамыз

   istream_iterator<string> ii(is);
   // ағыннан енгізу итераторын құрамыз
   istream_iterator<string> eos;
   // енгізудің сигналдық белгісі
   ostream_iterator<string> oo(os," ");
   // ағынға шығару итераторын құрамыз

   set<string> b(ii,eos);
   // b - енгізу ағынының мәліметтерімен инициалданатын вектор
   copy(b.begin(),b.end(),oo);
   // шығару ағынына буфер көшіреміз
}
```

Біз **set** контейнеріне мәндерді кірістіріп қойған кезімізде бірдей көшірмелер (дубликаты) есепке алынбайды. Оған қоса, **set** контейнерінің элементтері қажетті тәртіппен (ретпен) сақталады. Егер сіздің қолыңызда дұрыс құралдар болса, онда көптеген есептерді еш қиындықсыз шығаруға болады.

21.7.4 copy_if() алгоритмі

copy() алгоритмі көшіруді ешқандай шартсыз орындайды. **unique_copy()** алгоритмі қайталанатын бірдей мәндері бар көршілес элементтердің бірін алып тастайды. Үшінші алгоритм берілген предикаты ақиқат болған элементтерді ғана көшіреді.

```
template<class In,class Out,class Pred>
Out copy_if (In first, In last, Out res, Pred p)
// предикатты қанағаттандыратын элементтерді көшіреді
{
     while (first!=last) {
        if (p(*first)) *res++ = *first;
        ++first;
     }
     return res;
}
```

Біз 21.4 бөліміндегі **Larger_than** объект-функциясын қолдана отырып, тізбектің алтыдан үлкен барлық элементтерін таба аламыз:

```
void f(const vector<int>& v)
     // алтыдан үлкен барлық элементтерді көшіреміз
{
     vector<int> v2(v.size());
     copy_if(v.begin(), v.end(), v2.begin(), Larger_than(6));
     //...
}
```

Менің қателігімнен осы алгоритм 1998 ISO Standard стандартына енбей қалды. Қазіргі кезде бұл қателік жөнделген, бірақ осы уақытқа дейін C++ тілінің **copy_if** алгоритмі енгізілмеген нұсқалары кездеседі. Мұндай жағдайда осы бөлімде берілген анықтаманы пайдаланыңыз.

21.8 Сұрыптау және іздеу

Біз мәліметтерді жиі реттегіміз келеді. Біз осыған тәртіпті сүйемелдейтін **map** немесе **set** сияқты немесе сұрыптауды орындайтын құрылымдарды пайдалана отырып жете аламыз. STL кітапханасындағы сұрыптаудың ең кең таралған және пайдалы операциясы болып, біз бірнеше рет пайдаланған **sort()** функциясы саналады. Келісім бойынша **sort()** функциясы сұрыптау критериі ретінде **<** операторын қолданады, бірақ біз өзіміздің жеке критерийлерімізді де бере аламыз.

```
template<class Ran> void sort(Ran first, Ran last);
template<class Ran,class Cmp>void sort(Ran first,Ran last,Cmp cmp);
```

Қолданушының өзі анықтаған критерилерге негізделген сұрыптаудың мысалы ретінде, регистрді есепке алмай, сөз тіркестерін қалай сұрыптауға болатынын көрсетейік:

```
struct No_case {
//төменгі регистрдегі(x) < төменгі регистрдегі(y)?
    bool operator()(const string& x, const string& y) const
    {
       for(int i = 0; i<x.length(); ++i) {
          if (i == y.length()) return false; // y<x
          char xx = tolower(x[i]);
          char yy = tolower(y[i]);
          if (xx<yy) return true;            // x<y
          if (yy<xx) return false;           // y<x
       }
       if (x.length()==y.length()) return false;
       return true;       // x<y (x жолында символдар аз)
    }
};

void sort_and_print(vector<string>& vc)
{
    sort(vc.begin(),vc.end(),No_case());

    for (vector<string>::const_iterator p = vc.begin();
    p!=vc.end(); ++p)
       cout << *p<<'\n';
}
```

Тізбек сұрыптала салысымен, бізге `find()` функциясының көмегімен контейнердің басынан бастап барлық элементтерін қайта қарастырудың қажеті жоқ; оның орнына элементтердің реттелу тәртібін есепке алатын бинарлық іздеуді қолдануға болады. Негізінде, бинарлық іздеу келесі әрекеттерге келіп тіреледі.

Біз x-тің мәнін іздеп отырмыз деп айталық; ортадағы элементке қараймыз.

- Егер осы элементтің мәні *x*-ке тең болса, онда біз оны таптық!
- Егер осы элементтің мәні *x*-тен кіші болса, онда мәні *x*-ке тең кез келген элемент оң жақта орналасады, сондықтан біз оң жақ жартыны қарастырамыз (бинарлық іздеуді оң жақ бөлікке қолданамыз).

- Егер осы элементтің мәні *x*-тен артық болса, онда мәні *x*-ке тең кез келген элемент сол жақта орналасады, сондықтан біз бөліктің сол жақ жартысын қарастырамыз (бинарлық іздеуді сол жақ бөлікке қолданамыз).
- Егер біз соңғы элементке жетсек (солға немесе оңға қарай жылжып) және x-ке мәнді таппасақ, онда контейнерде мұндай элементтің жоқ болғаны.

Ұзын тізбектер үшін бинарлық іздеу **find()** алгоритміне (сызықты іздеуді көрсететін) қарағанда, өте тез орындалады. Стандартты кітапханада бинарлық іздеу алгоритмдері **search()** және **equal_range()** болып аталады. "Ұзын" деген сөзбен біз нені түсінеміз? Бұл нақты жағдайларға байланысты болады, бірақ **find()** алгоритмінен **search()** алгоритмінің артықшылығын көрсету үшін, әдетте он элемент жеткілікті. Мың элементтен тұратын тізбекте **search()** алгоритмі **find()** алгоритміне қарағанда, шамамен 200 есе тез жұмыс істейді (21.6.4 бөлімін қ.).

binary_search алгоритмінің екі нұсқасы бар:

```
template<class Ran, class T>
bool binary_search(Ran first, Ran last, const T& val);
template<class Ran, class T, class Cmp>
bool binary_search(Ran first, Ran last, const T& val, Cmp cmp);
```

Бұл алгоритмдер олардың кіру тізбектерінің реттеліп тұруын талап етеді. Егер осы шарт орындалмаса, онда шексіз циклдар сияқты қызық заттар болуы мүмкін. **binary_search()** алгоритмі контейнерде берілген мәннің бар немесе жоқ екендігін хабарлайды.

```
void f(vector<string>& vs)   //vs реттелген (сұрыпталған)
{
    if (binary_search(vs.begin(),vs.end(),"starfruit")) {
    // контейнерде "starfruit" тіркесі бар
    }
    //...
}
```

Сонымен, егер бізді берілген мәннің контейнерде бар/жоқ екендігі қызықтырса, онда оны анықтау үшін **binary_search()** алгоритмі – өте жақсы құрал. Егер бізге нақты бір элементті табу керек болса, онда біз **lower_bound(), upper_bound()** немесе **equal_range()** (23.4 және Б.5.4 бөлімдері) функцияларын қолдана аламыз. Көбінесе, контейнердің элементтері жай кілт емес, көлемді ақпараттан тұратын объектілерден құралса, контейнерде кілттері бірдей бірнеше элемент болса немесе бізді қандай элемент іздеу критериін қанағаттандыратыны қызықтырса, осы функцияларды пайдалану қажет болады.

ТАПСЫРМА

Әрбір операцияны орындағаннан кейін вектордың мазмұнын экранға шығарыңыз.

1. `struct Item { string name; int iid; double value; /* */};` құрылымын анықтаңыз; `vector<Item> vi` контейнерін құрып, оны файлдан алынған он сөз тіркесімен толтырыңыз.
2. `vi` контейнерін `name` өрісі бойынша сұрыптаңыз.
3. `vi` контейнерін `iid` өрісі бойынша сұрыптаңыз.
4. `vi` контейнерін `value` өрісі бойынша сұрыптаңыз; оны мәндерін кему ретімен баспаға шығарыңыз (яғни ең үлкен мәні бірінші болып шығуы керек).
5. Контейнерге `Item("horse shoe",99,12.34)` және `Item("Canon S400", 9988, 499.95)` элементтерін кірістіріп қойыңыз.
6. `name` өрісін бере отырып, `vi` контейнерінен `Item`-нің екі элементін жойыңыз.
7. `iid` өрісін бере отырып, `vi` контейнерінен `Item`-нің екі элементін жойыңыз.
8. `vector<Item>` типіндегі емес, `list<Item>` типіндегі контейнермен жаттығуды қайталаңыз.

Енді `map` контейнерімен жұмыс істеңіз.

1. `msi` атты `map<string, int>` контейнерін анықтаңыз.
2. Оған он жұпты (аты, мәні) кірістіріп қойыңыз, мысалы `msi["lecture"]=21`.
3. `cout` ағынына (аты, мәні) жұбын өзіңізге қолайлы түрде шығарыңыз.
4. `msi` контейнерінен (аты, мәні) жұбын жойыңыз.
5. `cin` ағынынан жұптарды оқитын және оларды `msi` контейнеріне орналастыратын функция жазыңыз.
6. Енгізу ағынынан он жұпты оқып, оларды `msi` контейнеріне орналастырыңыз.
7. `msi` контейнерінің элементтерін cout ағынына жазыңыз.
8. Мәндердің (бүтін) қосындысын `msi` контейнерінен шығарыңыз.
9. `mis` атты `map<int, string>` контейнерін анықтаңыз.
10. Мәндерді `msi` контейнерінен `mis` контейнеріне енгізіңіз; басқаша айтқанда, егер `msi` контейнерінде `("lecture",21)` элементі болса, онда mis контейнерінде де `(21,"lecture")` элементі болу керек.
11. `mis` контейнерінің элементтерін `cout` ағынына шығарыңыз.

Vector контейнеріне қатысты бірнеше тапсырмалар.

1. Файлдан жылжымалы нүктелі сандардың (16 мәннен аз болмауы керек) бірнешеуін **vd** атымен **vector<double>** контейнеріне оқыңыз.
2. **vd** контейнерінің элементтерін **cout** ағынына шығарыңыз.
3. Элементтер саны **vd** контейнеріндегідей болатын **vector<int>** типінен тұратын **vi** векторын құрыңыз; элементтерді **vd** контейнерінен **vi** контейнеріне көшіріңіз.
4. **cout** ағынына **(vd[i],vi[i])** жұптарын бір жолға бір-бірден шығарыңыз.
5. Экранға **vd** контейнері элементтерінің қосындысын шығарыңыз.
6. Экранға **vd** және **vi** контейнерлері элементтерінің қосындыларының айырмасын шығарыңыз.
7. Аргументтері ретінде тізбекті (итераторлар жұбы) алатын стандартты **reverse** алгоритмі бар; **vd** элементтерінің орналасу реттілігін кері бағытқа ауыстырып, оларды **cout** ағынына шығарыңыз.
8. **vd** контейнеріндегі элементтердің орташа мәнін есептеп, оны экранға шығарыңыз.
9. **vd2** атты **vector<double>** жаңа контейнерін құрыңыз да, оған **vd** контейнерінің орташа мәнінен кіші элементтерін көшіріп шығыңыз.
10. **vd** контейнерін сұрыптап алып, оның элементтерін экранға шығарыңыз.

БАҚЫЛАУ СҰРАҚТАРЫ

1. **STL** кітапханасындағы пайдалы алгоритмдерден мысалдар келтіріңіз.
2. **find()** алгоритмі не істейді? Кем дегенде бес мысал келтіріңіз.
3. **count_if()** алгоритмі не істейді?
4. **sort(b, e)** алгоритмі іздеу критериі ретінде нені қолданады?
5. STL кітапханасының алгоритмдері енгізу аргументі ретінде контейнерлерді қалай алады?
6. STL кітапханасының алгоритмдері шығару аргументі ретінде контейнерлерді қалай алады?
7. STL кітапханасының алгоритмдері "табылған жоқ" немесе "ақау" деген жағдайларды қалай белгілейді?
8. Функция-объект деген не?
9. Функция-объектінің функциядан қандай айырмашылығы бар?
10. Предикат деген не?
11. **accumulate()** алгоритмі не істейді?
12. **inner_product()** алгоритмі не істейді?

13. Ассоциативті контейнер деген не? Тым болмағанда үш мысал келтіріңіз.
14. `list` класы ассоциативті контейнер болып табыла ма? Неліктен жоқ?
15. Бинарлық бұтақты ұйымдастыру қағидасын (принципін) айтып беріңіз.
16. Теңгерілген бұтақ (шамамен) деген не?
17. `map` контейнерінде элемент қанша орын алады?
18. `vector` контейнерінде элемент қанша орын алады?
19. Егер `map` контейнері (реттелген) болса, онда `unordered_map` контейнері не үшін қажет?
20. `set` контейнерінің `map` контейнерінен қандай айырмашылығы бар?
21. `multi_map` контейнерінің `map` контейнерінен қандай айырмашылығы бар?
22. Егер біз қарапайым циклды жаза алатын болсақ, `copy()` алгоритмі не үшін қажет?
23. Бинарлық іздеу дегеніміз не?

ТЕРМИНДЕР

`accumulate()`	`map`	жалпыланған
`binary_search()`	`set`	қосымша: ()
`copy_if()`	`sort()`	объект-функция
`copy()`	`unique_copy()`	предикат
`equal_range()`	`unordered_ map()`	сұрыптау
`find()`	`upper_bound()`	теңгерілген бұтақ
`find_if()`	ағын итераторы	тізбек
`inner_product()`	алгоритм	хеш-функция
`lower_bound()`	ассоциативті контейнер	іздеу

ЖАТТЫҒУЛАР

1. Тарауды қайтадан оқып шығып, "Мынаны жасап көріңіз" қиындыларынан тапсырмаларды әлі орындамасаңыз, онда оны қазір жасап шығыңыз.
2. STL кітапханасы бойынша құжаттардың сенімді дереккөзін тауып, барлық стандартты алгоритмдерді атап өтіңіз.
3. `count()` алгоритмін өз бетіңізбен жүзеге асырыңыз. Оны тесттен өткізіңіз.
4. `count_if()` алгоритмін өз бетіңізбен жүзеге асырыңыз. Оны тесттен өткізіңіз.
5. Егер біз элементі табылмады дегенді білдіретін `end()` итераторын қайтара алмасақ, онда бізге не істеуге болады? `find()` және `count()` алгоритмдерін

бірінші және соңғы элементтеріне орнатылған итераторларды алатындай етіп, қайтадан жобалаңыз және жүзеге асырыңыз. Нәтижесін стандартты нұсқаларымен салыстырыңыз.

6. 21.6.5 бөліміндегі `Fruit` класының мысалында біз `Fruit` құрылымын `set` контейнеріне көшірдік. Егер де біз осы құрылымдарды көшіргіміз келмесе, не істейміз? Біз оның орнына `set<Fruit*>` контейнерін қолдана алар едік. Дегенмен осы жағдайда біз осы контейнер үшін салыстыру операторын анықтауға мәжбүр болар едік. `set<Fruit*, Fruit_comparison>` контейнерін қолданып осы жаттығуды тағы да бір рет орындаңыз. Осы жүзеге асырулардың арасындағы айырмашылықты талдап көріңіз.
7. `vector<int>` класы (стандартты алгоритмді қолданбай) үшін бинарлық іздеу функциясын жазыңыз. Өз қалауыңыз бойынша кез келген бір интерфейсті таңдап алып, оны тесттен өткізіңіз. Сіз жасаған бинарлық іздеу функциясының дұрыс жұмыс істеп тұрғанына қаншалықты сенімдісіз? `list<string>` контейнері үшін бинарлық іздеу функциясын жазып шығыңыз. Оны тесттен өткізіңіз. Осы екі бинарлық іздеу функциясы қаншалықты ұқсас? Қалай ойлайсыз, егер де сіз STL кітапханасы туралы ештеңе білмесеңіз, олар бір-біріне соншалықты ұқсас болар ма еді?
8. 21.6.1 бөліміндегі сөздердің жиілігін есептейтін мысалға оралыңыз да, оны өзгертіп, сөздерді лексикографикалық тәртіппен емес, жиіліктері бойынша реттеліп шығатындай етіңіз. Мысалы, экранға `C++: 3` жолы емес, `3: C++` жолы шығатындай болсын.
9. `Order` (тапсырыс) класын анықтаңыз, оның мүшелері клиенттің атынан, оның адресінен, туған күнінен және `vector<Purchase>` контейнерінен тұрады. `Purchase` класы тауарды сипаттайтын `name`, `unit_price` және count өрістерінен тұруы қажет. Файлдан мәлімет оқу және `Order` класының объектілерін файлға жазу механизмін анықтаңыз. `Order` класының объектілерін экранға шығару механизмін анықтаңыз. `Order` класының, тым болмағанда, он объектісінен тұратын файл құрыңыз да, оны `vector<Order>` контейнеріне оқып, аты (клиенттің) бойынша сұрыптап, нәтижесін қайтадан файлға жазып шығыңыз. `Order` класының, тым болмағанда, он объектісінен тұратын басқа бір файл құрыңыз, оның, шамамен алғанда, үштен бірі бірінші файлда сақталатын болсын, оларды `list<Order>` контейнеріне оқып, адресі (клиенттің) бойынша сұрыптап, нәтижесін қайтадан файлға жазыңыз. `std::merge()` функциясын қолдана отырып, екі файлды біріктіріп, үшінші файлға жазыңыз.
10. Алдыңғы жаттығудың екі файлындағы тапсырыстардың жалпы санын есептеңіз. `Purchase` класының (әлбетте) жеке объектісінің мәні `unit_price*count`-қа тең.
11. Файлдан тапсырыстарды енгізу үшін қолданушының графикалық интерфейсін құрыңыз.
12. Тапсырыстар файлына сұраныс беретін қолданушының графикалық

интерфейсін құрыңыз; мысалы, "`Joe-дан` келіп түскен барлық тапсырыстарды табу", "`Hardware` файлындағы тапсырыстардың жалпы бағасын анықтау" немесе "`Clothing` файлындағы барлық тапсырыстарды көрсету" сияқты болуы тиіс. Көмекші мәлімет: алдымен қарапайым интерфейс құрып алып, соның негізінде графикалық интерфейс құруға тырысыңыз.

13. Сөздерді іздеуге арналған сұраныстарды өңдейтін программада қолдану үшін мәтіндік файлды "тазалайтын" программа жазыңыз; басқаша айтқанда, пунктуация белгілерін босорындарға ауыстырып, сөздерді төменгі регистрге аударып, *don't* өрнегін *do not* сөзіне алмастырып (және т.б.) және көпше түрдегі зат есімдерді жекеше түрдегі зат есімдерге (мысалы, *ships* сөзі *ship* болады) ауыстырыңыз. Асыра сілтеп жібермеңіз. Мысалы, көпше түрді анықтау қиынырақ болады, сондықтан егер *ship* және *ships* сөздерін кездестірсеңіз, соңындағы *s* әрпін алып тастап, *ship* сөзін қалдырсаңыз болғаны. Осы программаны көлемі 5 000 сөзден кем болмайтын нақты мәтіндік файлға (мысалы, ғылыми мақалаға) қолданыңыз.
14. Келесі сұрақтарға жауап беретін және келесі тапсырмаларды орындайтын программа (алдыңғы жаттығудың нәтижесін қолданып) жазыңыз: "*ship* сөзі файлда неше рет кездеседі?", "Қандай сөз жиі кездеседі?", "Файлдағы ең ұзын қандай сөз?", "Файлдағы ең қысқа қандай сөз?", "*s* әрпінен басталатын барлық сөздерді көрсетіңіз?" және "Төрт әріптен тұратын барлық сөздерді көрсетіңіз".
15. Алдыңғы жаттығуға арналған қолданушының графикалық интерфейсін құрыңыз.

СОҢҒЫ СӨЗ

STL кітапханасы C++ тілінің стандартты ISO кітапханасының контейнерлері мен алгоритмдерінің бір бөлігі болып саналады. Ол қолданушыларды икемді және пайдалы болып табылатын негізгі құралдармен қамтамасыз етеді. Ол бізді көптеген жұмыстарды орындаудан босатады: олардан гөрі дөңгелекті ойлап тапқан қызықты шығар, бірақ ол жемісті еңбек болмайды. Оған қоса, STL кітапханасы жалпылама программалаудың мысалы болып табылады, ол нақты мәселелерді шешу мен есептер шығаруда қуатты және ортақ құралдарды пайдалануға болатынын көрсетеді. Егер сізге, көптеген программалаушыларға да қажет іс болып саналатын, мәліметтерді өңдеу жұмыстарын орындау керек болса, онда STL кітапханасы бірсыпыра идеяларды, тәсілдерді жүзеге асырудағы нағыз керекті көмекші құрал мысалы болып табылады.

IV бөлім

Қосымша тақырыптар

Мұраттар және тарих

Біреу: "Маған жай ғана менің нені қалайтынымды айтсам жеткілікті болатын программалау тілі қажет", десе, оған мұз кәмпит беріңіз.

- Алан Перлис (Alan Perlis)

Осы тарауда өте қысқа және таңдаулы түрде программалау тілдерінің тарихы жазылған және солардың құрметіне құрылған мұраттар (идеалдар) сипатталған. Осы идеалдар және оларды өрнектейтін программалау тілдері кәсіби шеберліктің негізін қалайды. Бұл кітапта C++ тілі қолданылғандықтан, біз назарымызды осы тілге және осының әсерінен туындаған басқа тілдерге де аударды қ. Осы тараудың мақсаты – кітапта ұсынылған идеялардың негізін айтып, даму жолдарын көрсету. Тілдердің әрқайсысын сипаттай отырып, біз оны құрған жан немесе құрушылар туралы айтамыз: тіл – бұл тек абстрактілі туынды емес, ол, сонымен бірге, белгілі бір уақыт аралығында адамдар алдында пайда болған мәселелерге арналған нақты бір шешім.

22.1 Тарих, мұраттар және кәсіби шеберлік

"Тарих – бұл бекершілік", – деп Генри Форд апелляциясыз мәлімдеген (Henry Ford). Бұған қарама қарсы ой ежелгі заманнан бері кеңінен айтылып келеді: "Тарихты білмеген адам, оны қайталап шығады". Мәселе қандай тарихты таңдап, қайсысын біліп, ал қайсысын алып тастауға болатынында: басқа бір белгілі тұжырым "ақпараттың 95%-ы – бұл бекершілік" (өз жағымыздан атап өтеріміз, 95% бұл төмендетіліп берілген баға шығар). Тарих пен қазіргі заманның байланысына деген біздің көзқарасымыз, тарихты түсінбей кәсіби маман болу мүмкін емес.

Өз білімі саласы аймағында тарихты аз білетін адамдар көбінесе тез сенгіш болып келеді, өйткені кез келген заттың тарихы шынайы, бірақ жұмыс істемейтін идеялармен толтырылған. Тарих төркіні, бағасы іс тәжірибемен дәлелденетін идеялардан тұрады.

Біз көптеген программалау тілдері мен операциялық жүйелер, мәліметтер базалары, графикалық жүйелер, желілер, веб, сценарилер және т.с.с. әртүрлі программалық жабдықтамалардың негізінде жатқан өзекті идеялардың шығуын жалықпастан-ақ айтып шығар едік. Бірақ осы пайдалы әрі маңызды қосымшаларды кез келген жерлерден тауып алуға болады. Бізге берілген орын идеалдарды, яғни мұраттарды үстіртін ғана сипаттап, программалау тілдерінің тарихын баяндауға ғана жетеді.

Программалаудың ақырғы мақсаты пайдалы жүйелер құру болып табылады. Программалау тілдері мен әдістері туралы қызу пікірталастар кезінде олар жайлы ұмытып кету оңай. Осыны есіңізде сақтаңыз! Егер сіздің есіңізге салу керек болса, онда 1 тарауды қайтадан оқып шығыңыз.

22.1.1 Программалау тілінің мақсаттары мен философиясы

Программалау тілі деген не? Олар не үшін қажет? Төменде бірінші сұрақтың кеңінен таралған жауаптары көрсетілген.

- Машиналарға нұсқаулықтар беруге арналған құрал.
- Алгоритмдерді жазу тәсілі.
- Программалаушылардың қатынасу құралы.
- Эксперимент, яғни тәжірибе жасауға арналған құрал.
- Компьютерлендірілген құрылғыларды өрнектеу тәсілі.
- Түсініктер арасындағы қатынастарды беру әдісі.
- Жоғарғы деңгейдегі жобалау шешімдерін өрнектеу құралы.

Біздің жауабымыз мынадай: "Бәрі де бірге және одан да көп!". Әрине, мұнда әмбебап программалау тілдері туралы әңгіме болады. Олардан басқа, шағын және нақты тұжырымдалған есептерге арналған арнайы және пәндерге бағытталған программалау тілдері бар. Программалау тілінің қандай қасиеттері қажетті болып саналады?

- Ауысымдылығы.
- Типтік қауіпсіздігі.
- Нақты анықталғандығы.
- Жоғары жұмыс өнімділігі.
- Идеяларды дәлме-дәл көрсету қабілеттілігі.
- Жеңіл жөнделіп түзетілуі (debugging).
- Жеңіл тесттен өткізілуі.
- Барлық жүйелік ресурстарға қол жеткізуі.
- Платформадан (тағандардан) тәуелсіздігі.
- Барлық платформада орындалу мүмкіндігі.
- Он жылдықтарға созылатын тұрақтылығы.
- Қолданбалы аймақта болып жататын өзгерістерге жауап ретінде тұрақты жетілдірілуі.
- Оқытудың жеңілдігі.
- Мөлшерінің (көлемінің) шағындығы.
- Кең таралған программалау стильдерін сүйемелдеуі (мысалы, объектіге бағытталған және жалпылама программалау).
- Программаны талдау мүмкіндігі.
- Көптеген мүмкіндіктерінің болуы.
- Ірі қоғамдастықтар жағынан қолдау табуы.

- Жаңадан қосылушы адамдардан (студенттер, оқушылар) қолдау табуы.
- Сарапшылар (мысалы, инфрақұрылым конструкторлары) үшін толыққанды мүмкіндіктерінің болуы.
- Программаны құруға қажетті көптеген қолжетімді құралдардың болуы.
- Программалық жабдықтаманың көптеген қолжетімді компоненттерінің болуы (мысалы, кітапханалар).
- Ашық код жазу қоғамдастығы жағынан қолдау табуы.
- Негізгі платформаларды берушілер жағынан қолдау табуы (Microsoft, IBM және т.б.).

Өкінішке орай, осы мүмкіндіктердің барлығын бір мезетте алуға болмайды. Бұл өкінішті-ақ, өйткені осы қасиеттердің әрқайсысы объективті түрде оң нәтиже береді: олардың әрқайсысы пайда әкеледі, ал осы қасиеттері жоқ тіл программалаушыларды қосымша жұмыс жасауға итермелеп, олардың өмірін күрделендіреді. Осы мүмкіндіктердің барлығын да бір мезетте алуға болмайтындығы себебінің іргелі түсінігі бар: олардың кейбіреулері өзара қарама-қарсы сипатта болып табылады. Мысалы, тіл толығымен платформадан тәуелсіз бола алмайды, ол сондықтан да бір мезетте барлық жүйелік ресурстарға қол жеткізе алмайды; мысалы, программа нақты бір платформада жоқ ресурсты сұрағанымен, олармен мүлде жұмыс жасай алмайды. Осыған орай, біз, әрине, тілдің шағын әрі үйренуге жеңіл болғанын қалар едік, бірақ бұл программаның барлық жүйелерде және кез келген пәндік аймақтарда қолдау табуының талаптарына сәйкес келмейді ғой.

Программалаудағы идеалдар маңызды рөл атқарады. Олар техникалық шешімдер мен әрбір тілді, кітапхананы және құралдарды құрастыру кезіндегі сәйкестіктерді таңдау кезінде бағыттаушы ретінде болады. Негізінде, программа жазған кезде, сіз жобалаушы рөлін атқарасыз және жобалық шешімдерді қабылдауға міндеттісіз.

22.1.2. Программалау мұраттары (идеалдары)

The C++ Programming Language кітабының кіріспесі мынадай сөздерден басталады: "C++ тілі – байсалды программалаушылар өз жұмыстарынан ләззат алу үшін құрылған әмбебап программалау тілі". Бұл нені білдіреді? Программалау дайын өнімді жасауды қарастырмай ма? Неліктен программаның дұрыстығы, сапасы және сүйемелденуі жөнінде ештеңе айтылмаған? Неліктен жаңа программаның бастапқы жоспарлануы мен оның нарықта пайда болуы арасындағы мерзім жайлы айтылмаған? Ал, программалық жабдықтаманы сүйемелдеу маңызды емес пе? Мұның барлығы, әлбетте, маңызды шығар, бірақ біз программалаушы туралы ұмытпауымыз керек. Басқа мысал қарастырайық. Дональд Кнут (Don Knuth) былай деп айтқан екен: "Alto компьютерінің ең жақсысы сол, ол түнде жылдам жұмыс істемейді". Alto – бұл ең алғашқы дербес компьютерлердің бірі, Xerox Palo Alto

Research Center (PARC) орталығының компьютері. Оның біріге отырып жұмыс істеуге арналған қарапайым компьютерлерден айырмашылығы, ол күндіз жұмыс істейтін программалаушылардың арасында өткір бақталастық тудырған болатын.

Біздің құралдарымыз бен программалау әдістеріміз программалаушының жақсырақ жұмыс істеп, ең жоғарғы нәтижелерге қол жеткізулеріне арналған. Бұл туралы ұмытпауларыңызды сұраймын. Біз программалаушыға энергияны аз жұмсай отырып, жақсы программалық жабдықтама құруға көмектесу үшін қандай қағидаларды ұсынуымызға болады? Біз өз ойымызды кітап ішінде айтып өттік, сондықтан да осы бөлім, негізінде, солардың түйіндемесі ретінде көрсетілген.

Кодтың жақсы құрылымын жасауға бізді итермелейтін негізгі себеп – оған көп күш жұмсамай, өзгерістер енгізуге ұмтылу. Код құрылымы неғұрлым жақсы болса, онда соншалықты кодты өзгерту оңай, қатені тауып жөндеу, жаңа қасиет енгізу, жаңа архитектураға бағыттау, программаның жұмыс жылдамдығын арттыру және т.б. әрекеттер жылдам жүзеге асады. "Жақсы" деп айтқан кезде, біз дәл осыларды ескереміз.

Бөлімнің қалған бөліктерінде келесі сұрақтарды қарастырамыз:

- Біз кодтан оның не істегенін қалаймыз?
- Программалық жабдықтаманы құруға деген жалпы екі көзқарас болса, солардың бір-бірімен араласуы, әрқайсысын жеке пайдаланғаннан гөрі, жақсырақ нәтиже алуды жабдықтамасыз етеді.
- Код түрінде өрнектелген программа құрылымының өзекті аспектілері:
  - Идеяларды тікелей түрде айтып өту.
  - Абстракция деңгейі.
  - Модульділік.
  - Қисындылығы мен минимализмі.

Идеалдар өмірде нақты түрде жүзеге асуы керек. Олар менеджерлер мен сарапшылардың ермек үшін айтатын сөздері емес, ойланудың негізі болып табылады. Біздің программаларымыз идеалдарға жақындауы тиіс. Біз тұйыққа тірелгенде, артқа қарай ораламыз (кейде бұл көмектеседі) да, біздің мәселеміз принциптерден ауытқудың салдарынан емес пе екен деп, соны байқауға тырысамыз. Біз программаны бағалаған кезде (қолданушыларға беруге дейін жасалғаны дұрыс), принциптердің бұзылуынан болашақта мәселелер туындамас па екен деп, соны іздейміз. Идеалдарды мүмкіндігінше жиі қолданыңыз, бірақ есіңізде болсын, практикалық концепциялар (мысалы, жұмыс өнімділігі мен қарапайымдылық), соған қоса тілдің әлсіздігі (бірде-бір тіл барынша жетілген болып табылмайды) идеалға жетуге емес, тек оған жақындау мүмкіндігін береді.

Идеалдар бізге нақты техникалық шешімдер қабылдауға көмектесе алады. Мысалы, біз кітапхана үшін тек өзіміз басқалардан тәуелсіз күйде (14.1 бөлімін қ.) интерфейсті таңдау туралы шешім қабылдай алмаймыз. Мұның нәтижесінде

шатасу туындауы мүмкін. Осының орнына біз өзіміздің алғашқы принципімізді еске алып, берілген нақты кітапхана үшін ненің маңызды екенін шешіп, сонан соң барып интерфейстің логикалық жиынын құруымыз керек. Ал ең негізгісі – әрбір жоба құжаттамасында жобалау принциптерін қалыптастырып, бір-бірімен сәйкестендіріліп келісілген компромисті шешімдер қабылдау және оларға программа кодында түсініктеме беру.

Жобаны бастағанда принциптерді қадағалап, олардың алдағы міндеттермен және шығарған есебіңіздің бұрын алынған шешімдерімен қалай байланысқанын қарап шығыңыз. Бұл – идеяларды анықтап айқындаудың жақсы тәсілі. Кейінірек, есептерді жобалау және программалау кезеңінде, сіз тұйыққа тірелетіндей жағдай туындаса, артқа оралып, өзіңіздің кодыңыз қай жерде идеалдардан ауытқыды, міне, дәл сол жерде жобалауға байланысты мәселелер туындап, қателер шығуы мүмкін. Бұл көзқарас келісім бойынша қабылданған программалаушының қатені бір жерден және оны іздеп табудың да бір тәсілін ғана пайдаланатын кодты жөндеудің баламалы бір жолы болып табылады. "Қате әрқашанда сіз күтпеген жерде тұрады, – немесе сіз оны тауып қойғансыз".

22.1.2.1 Біз нені қалаймыз?

Көбінесе біз төмендегілерді қалаймыз:

- *Дұрыстығы.* Иә, "дұрыс" деген сөздің астарында не жатқанын анықтау өте қиын, бірақ бұл жұмыстың маңызды бөлігі. Бұл түсінікті жиі нақты бір жобаның аясында біз үшін басқа адамдар анықтайды, бірақ осы жағдайда олардың айтқанын біз түсіндіріп беруге міндеттіміз.
- *Сүйемелдеудің (программаны) жеңілдігі.* Кез келген табысты программа уақыт өте келе өзгеріске ұшырайды; ол жаңа аппараттық жабдықтама мен платформаға бейімделеді, жаңа мүмкіндіктермен толықтырылады және осыған орай шығатын жаңа қателерді жою қажет. Келесі бөлімдерде біз программаның құрылымы арқылы осыған қалай жетеміз, соны көрсетеміз.
- *Жұмыс өнімділігі.* Жұмыс өнімділігі (тиімділігі) – салыстырмалы түсінік. Ол программаның мақсатына сәйкес болуы керек. Тиімді код қажеттілігіне қарай төменгі деңгейлі болуы керек, ал жоғары деңгейлі құрылым программаның тиімділігін азайтады деп программалаушылар жиі тұжырым жасайды. Біз бұған қарсы пікірдеміз, біз ұсынған принциптерді қолдану көбінесе кодтың жоғары тиімділігін жабдықтамасыз етеді деп санаймыз. Әрі абстрактілі, әрі өте тиімді STL кітапханасы осындай кодтың мысалы болып табылады. Жұмыс өнімділігінің азаюы көбінесе төменгі деңгейлі әрекеттермен тым қызығушылықтың және де керісінше, оларды елемеудің де салдары болуы мүмкін.

- *Дер кезінде жеткізу.* Жақсы деңгейде жасалған программаны жоспарланған мерзімнен бір жыл кейін беру – онша жақсы нәрсе емес. Әрине, адамдардың мүмкін емес нәрсені де күтуі мүмкін, бірақ біз сапалы программалық жабдықтаманы мүмкін болатын мерзімде жасап беруіміз қажет. Дер кезінде аяқталған программаның сапасы жоғары болмайды деген жалған сенім бар. Біз бұған қарсы пікірдеміз, жақсы құрылымды (мысалы, ресурстарды басқаруды, инварианттарды және интерфейс жобасын) негізге алып, тесттен өткізуі жоспарланған және өзіне сәйкес келетін қажетті кітапханаларды қолдану (көбінесе нақты бір қосымшаға немесе пәндік аймаққа есептелген) арқылы дайындалған программаны көрсетілген уақытта толығымен жасап үлгіруге болады.

Осы айтылғандардың барлығы біздің кодтың құрылымына деген қызығушылығымызды туындатады, олар:

- Егер программа құрылымы айқын болса, программада кездескен қатені (әрбір үлкен программа қателер болады) табу оңайға түседі.
- Егер программаны бөтен адамға түсіндіру немесе толықтырып өзгерту керек болса, төменгі деңгейдегі кодтардың араласуына қарағанда, айқын құрылымды түсіну жеңіл болады.
- Егер программаның жұмыс өнімділігіне қатысты мәселелері болса, онда жоғары деңгейлі программаны баптау төменгі деңгейлі программаға қарағанда анағұрлым жеңіл орындалады (өйткені ол жалпы принциптерге сәйкес келеді және жақсы анықталған құрылымнан тұрады). Жұмысты жаңадан бастаған программашыларға жоғары деңгейлі құрылым түсініктірек, оған қоса, жоғары деңгейлі код төменгі деңгейлі кодқа қарағанда, тесттен жеңіл өткізіледі және оны баптау да қиын емес.

Программа міндетті түрде түсінікті болуы тиіс. Программаны бізге түсінуге және ол туралы ойлануға көмектесетін нәрселердің барлығы да жақсы болып саналады. Негізінде, тәртіппен жасалған заттар тәртіп сақтамай жасалғандардан әлдеқайда жақсы, тек тәртіп тым қысқартудың нәтижесі болып кетпесін.

22.1.2.2 Жалпы көзқарастар

Дұрыс программалық жабдықтама құруға деген екі көзқарас бар.

- *Төменнен-жоғары.* Жүйе дұрыстығы дәлелденген шағын бөліктерден ғана құралады.
- *Жоғарыдан-төмен.* Жүйе қателері бар деп саналатын, тексерілмеген бөліктерден құралады, сонан соң барып барлық қателері айқындалады.

Ең сенімді жүйелердің осы екі көзқарастың араласуы арқылы құрылуы қызықты нәрсе, әйтсе де олар бір біріне қарама-қарсы сипатта болып келеді. Мұның себебі қарапайым: нақты жұмыс атқаратын ірі жүйелер үшін осы екі көзқарастың ешқайсысы да талап етілген дұрыстықты, бейімделуді және сүйемелдеуді қолайлы түрде орындауға кепілдік бере алмайды.

- Біз алдын ала барлық қателіктер көздерін жойып, негізгі компоненттерді құрып және тексере алмаймыз.
- Біз негізгі компоненттердің (кітапханалардың, ішкі жүйелердің, иерархиялардың және т.б.) кемшіліктерін, оларды аяқталған бір жүйеге біріктіре отырып, толығымен өтей алмаймыз.

Дегенмен, осы екі тәсілдің араласуы, олардың әрқайсысын жеке алғаннан гөрі, көп мүмкіндік бере алады: біз сапасы жоғары компоненттер құра аламыз (немесе сұрап аламыз, сатып аламыз), қалған мәселелерді қателерді өңдеу арқылы тұрақты тесттен өткізе отырып жоя аламыз. Оның үстіне, егерде біз жақсы компоненттерді құруды жалғастыра беретін болсақ, онда олардағы қажетті "ретсіз арнайы" кодтың үлесін қысқарта отырып, жүйенің үлкен бөліктерін құруға болады.

Тесттен өткізу программалық жабдықтаманы құрудың көрнекті бөлігі болып табылады. Ол 26-тарауда толығырақ талқыланады. Тесттен өткізу – ол қателерді тұрақты түрде іздеу деген сөз. Тесттен өткізуді, мүмкіндігінше, ертерек бастап, жиі-жиі орындап отыру керек. Мысалы, біз программаларымызды тесттен өткізуді барынша қарапайым етіп құрып, кодтар ішінде қателердің айқындалмай қалуына кедергі жасайтындай етіп құруға тырысамыз.

22.1.2.3 Идеяларды тікелей өрнектеу

Біз қандай да бір идеяны жоғары немесе төменгі деңгейде өрнектегенде, мәселені айналып өтетін жолмен емес, оны тікелей кодта көрсеткен жөн. Идеяларды тікелей кодта өрнектеудің негізгі принципі бірнеше спецификалық (арнайы) нұсқалардан тұрады:

- *Идеяларды тікелей кодта өрнектеу.* Мысалы, аргументті жалпы емес (мысалы, `int`), арнайы типтің көмегімен көрсеткен жөн (мысалы, `Month` немесе `Color`).
- *Тәуелсіз идеялар кодында тәуелсіз бейнелеу.* Мысалы, стандартты `sort()` функциясы, кейбіреулерін қоспағанда, кез келген қарапайым типтегі кез келген стандартты контейнерді реттей алады; сұрыптау концепциялары, контейнерді сұрыптау критерилері және қарапайым типтер тәуелсіз түсініктер болып табылады. Егер де біз бос жады аймағында орналасқан объектілер векторын құруымыз керек болса және оның элементтері `vector::sort()` функциясынан шақыруға анықталған `before()` функция-мүшесі бар

`Object` класынан шығарылған класқа жататын болса, онда бізде `sort()` функциясының шағындау бір нұсқасы болуы тиіс, өйткені сақтау туралы, кластар иерархиясы, қолжетімді функция-мүшелері, реттеу тәртібі және т.б. жайлы болжамдар жасаған болатынбыз.

- *Идеялардың арасындағы қатынастарды тікелей кодта бейнелеу.* Кодта тікелей көрсетуге болатын ең ортақ қатынастарға мұрагерлік (мысалы, `Circle` класы `Shape` класының бір түрі болып табылады) және параметрлеу (мысалы, `vector<T>` класы элементтерінің типінен тәуелсіз барлық векторлар үшін ортақ бір нәрсені өрнектейді) жатады.
- *Кодта бейнеленген идеялардың еркін араласуы, мұнда кодтар ішіндегі кездесетін комбинациялардың өзіндік мағынасы болуы тиіс.* Мысалы, `sort()` функциясы элементтердің әртүрлі типтерін және контейнерлердің түрлерін де пайдалануға мүмкіндік береді, бірақ осы элементтер `<` операциясын сүйемелдей алуы керек (әйтпесе салыстыру критериін беретін `sort()` функциясын қосымша аргументімен пайдаланған жөн), ал біз реттейтін контейнерлер кез келген түрде (қатынас құра) пайдаланыла алатын итераторларды сүйемелдей алуы қажет.
- *Қарапайым идеялардың қарапайым түрде бейнеленуі.* Жоғарыда айтылған принциптерді сақтап, оларды қолдану тым жалпы сипаттағы кодтарға әкелуі мүмкін. Мысалы, біз талап етілгеннен гөрі, кластар иерархиясының күрделірек таксономиясына (мұрагерлік құрылымы) тап боламыз немесе әрбір қарапайым класс үшін жеті параметрмен кездесуіміз мүмкін. Осындай қиындықтардан құтылу үшін біз кең тараған немесе маңызды жағдайлар үшін қарапайым нұсқалар құруға тырысамыз. Мысалы, элементтерін `op` операторы арқылы реттейтін `sort(b,e,op)` функциясының жалпы нұсқасынан басқа, реттеуді "кіші" қатынасы арқылы жанамалы түрде орындайтын `sort(b,e)` нұсқасы бар. Егер біз жасай алсақ (немесе C++0x тілін пайдалануға мүмкіндігіміз болса; 22.2.8 бөлімін қ.), онда стандартты контейнерді "кіші" қатынасы арқылы сұрыптау үшін `sort(c)` нұсқасын және стандартты контейнерді `op` операторы арқылы сұрыптайтын `sort(c, op)` функциясын қарастырар едік.

22.1.2.4 Абстракция деңгейі

Біз *абстракцияның мүмкін болатын ең жоғарғы (максималды) деңгейінде жұмыс жасағанды* дұрыс көреміз, басқаша айтқанда, өзіміздің шешімдерімізді барынша жалпы түрде көрсетуге тырысамыз.

Мысалы, сіздің ұялы телефоныңызда сақтауға болатын жазбалар телефон кітапшасында қалай бейнелетіндігін қарастырайық. Біз көптеген жұптарды (аты, мәні) `vector<pair<string, Value_type>>` класы арқылы көрсете алар едік. Бірақ егер біз адамның атын іздеу үшін әрқашан да осы жиынды

пайдаланатын болсақ, онда абстракцияның бұдан жоғарылау деңгейін бізге `map<string, Value_type>` класы жабдықтамасыз етеді. Бұл жазбаларға қол жеткізу (пайдалану) функциясын жазбауға (және түзетуге де) мүмкіндік береді. Басқа жағынан алғанда, `vector<pair<string, Value_type>>` класының өзі `string[max]` және `Value_type[max]` сияқты тіркестер мен мәндер арасындағы қатынастары жанамалы сипатта болатын екі жиымға қарағанда, абстракцияның жоғарғы деңгейінде орналасады. Абстракцияның ең төменгі деңгейінде `int` типі (элементтер саны) мен екі `void*` нұсқауышының (компиляторға емес, программалаушыға белгілі жазбаның кез келген бір формасына сілтеме жасайтын) араласуы орын алар еді. Біздің мысалымызда ұсынылған шешімдердің әрқайсысын төменгі деңгейлік абстракцияға жатқызуға болады, өйткені олардың әрқайсысында да біздің негізгі назарымыз олардың функцияларына емес, мәндер жұбын бейнелеуге арналған. Мұны нақты бір қосымшаға (программаға) жақындату үшін, оның пайдалану тәсілін тікелей көрсететін класты анықтап алған жөн. Мысалы, біз қолайлы интерфейсі бар `Phonebook` класын бірге пайдалана отырып, қосымшаның кодын жазып шыға алар едік. `Phonebook` класын жоғарыда сипатталған мәліметтерді бейнелеу жолдарының бірі арқылы жүзеге асыруға болар еді.

Абстракцияның жоғарғы деңгейінде қалуды қалайтын себебіміз (егер біздің құзырымызда абстракцияның сәйкес механизмі бар болса және біздің тіл оны тиімділіктің қолайлы деңгейінде қолдаса), компьютердің аппараттық жабдықтамасы терминдерімен айтылған шешімдерге қарағанда, мұндай тұжырымдау біздің есептер туралы ойымызға және оның шешімдеріне жақын болып табылады.

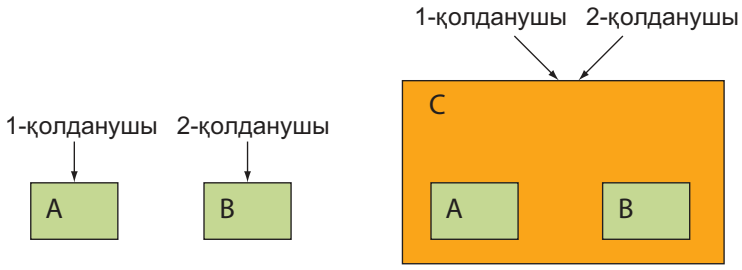
Көбінесе, абстракцияның төменгі деңгейіне көшудің негізгі себебі тиімділік деп аталып жүр. Дегенмен, мұны нақты қажеттілік пайда болған жағдайда ғана жасаған жөн (25.2.2 бөлімі). Төмен деңгейлі тілдердің қасиетін пайдалану (өте қарапайым) программаның жұмыс өнімділігін әрқашан да көтере бермейді. Кейде ол оңтайландыру (оптималдау) мүмкіндігін алып тастайды. Мысалы, `Phonebook` класын пайдалана отырып, оның жүзеге асу жолын, мысалы, `string[max]` және `Value_type[max]` жиымдарының араласуы арқылы немесе `map<string,Value_type>` класы көмегімен деп таңдай аламыз. Кейбір қосымшалар үшін ең тиімдісі болып бірінші нұсқа саналса, ал басқалары үшін – екінші нұсқа тиімді болады. Әрине, егер сіз өз телефон кітапшаңыздағы жазбаларды сақтау үшін программа жазсаңыз, өнімділік негізгі фактор болып табылмайды. Бірақ егер де миллиондаған жазбаларды сақтай отырып, оларды өңдеу қажет болса, онда ол маңызды болады. Одан да гөрі маңыздырағы, төменгі деңгейдегі құралдарды пайдалану жұмыс уақытын көп жұмсаумен тығыз байланысты болып келеді, ал программалаушыға өз программаларын жетілдіру үшін уақыт онсыз да жетпей жатады (өнімділікті арттыру немесе басқалары).

22.1.2.5 Модульділік

Модульділік – бұл принцип. Біз өзіміздің жүйелерді жеке құруға болатын, талдау жасайтын және тесттен өткізуге болатын компоненттерден (функциялар, кластар,

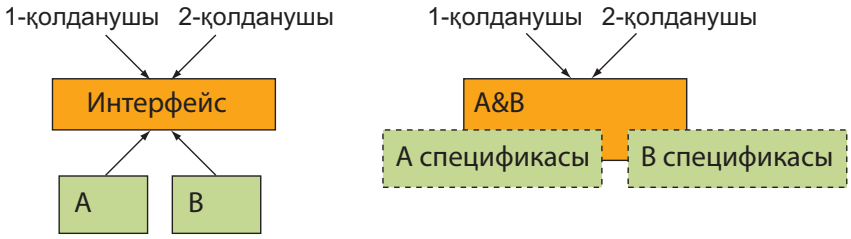
кластар иерархиясы, кітапханалар және т.б.) құрастырғымыз келеді. Негізінде, біз осындай компоненттерді оларды қайталай отырып, бірнеше программаларда пайдалануға болатындай етіп жобалап, жүзеге асырғымыз келеді. *Қайталап пайдалану* (reuse) – бұл кез келген бір жерде бұрын қолданылып тесттен өткізілген жүйелерді құру және де осындай компоненттерді жобалау және қолдану істері. Біз кластарды, кластар иерархиясын, интерфейстерді жобалау және жалпыланған программалауды талқылай отырып, бұл тақырыпты бұрын қарастырғанбыз. 22.1.3 бөлімінде қарастырылған программалау стилі туралы айтылғандардың көбісі қайталап пайдалануға болатын компоненттерді жобалау, жүзеге асыру және қайта қолдануға байланысты болып келеді. Әрбір компонентті бірнеше программаларда қолдануға бола бермейтінін айтып кеткен жөн; кейбір программалар тым арнайы болып табылады, оларды басқа жағдайларда қолдануға бейімдеу өте қиынға соғады.

Кодтың модульдігі қосымшаның негізгі логикалық бөлімдерін бейнелеуі тиіс. Қайталанып қолданылатын C компонентіне жеке-дара жасалған екі A және B кластарын жай ғана жүктей салып, қайталап қолдану деңгейін арттырудың қажеті жоқ. A және B кластарының интерфейсін жаңа C модуліне біріктіру кодты күрделендіріп жібереді:



Мұнда екі қолданушы да C модулін қолданады. Сіз C модулінің ішіне үңілмейінше, екі қолданушы да қолжетімді компонентті ортақ пайдалана алатындықтан, олардың артықшылықтары бар деп ойлайсыз. Бірігіп қолданудың (қайталап) пайдасына (мұнда ол орын алмайды) оны жеңіл тесттен өткізуді, кодтың шағын көлемін, қолданушы базасын кеңейтуді, т.с.с. жатқызуға болады. Өкінішке орай, жай ғана қарапайымдылықтан басқа, мұнда ешнәрсе жоқ, бұл жиі кездесетін жағдай.

Немен көмектесуге болады? Мүмкін, A және B кластарына ортақ интерфейс құру керек шығар?



Бұл диаграммалар сәйкесінше мұрагерлік пен параметрлеуді қолдануды ойға салады. Екі жағдайда да, жұмыстың мағынасы болуы үшін, A және B кластарының интерфейстерін жай біріктіргеннен гөрі, интерфейс шағын ғана болуы керек. Басқаша айтқанда, қолданушы қабылданған шешімнен пайда болуы үшін, A және B кластарының іргелі ортақтастығы болуы керек. Біз қайтадан интерфейске (9.7 және 25.4.2 бөлімдерін қ.) және, оның жалғасы ретінде, инварианттарға (9.4.3 бөлімін қ.) оралғанымызға назар аударыңыз.

22.1.2.6 Қисындылық пен минимализм

Қисындылық пен минимализм – идеяларды өрнектеудің негізгі принциптері. Осыған орай, сыртқы пішінге қатысты сұрақтар сияқты, біз оларды ұмытып кетуіміз мүмкін. Дегенмен, шатысқан шым-шытырық жобаны әдемі етіп жасап шығару өте қиын, сондықтан қисындылық пен минимализм талаптарын программаның көптеген шағын нақтылықтарына әсер ететін жобалаудың критерилері ретінде қарастыруға болады:

- Егер қажеттілігіне күмәндансаңыз, жаңа қасиетті қоспаңыз.
- Егер ұқсастықтарының іргелі сипаттары болса, ұқсас интерфейстердің (олардың аттары да) ұқсас қасиеттері болуы керек.
- Егер айырмашылықтарының іргелі сипаттары болса, ұқсамайтын қасиеттердің атаулары да ұқсамауы тиіс (мүмкіндігінше интерфейстері де әртүрлі болады).

Қисынды атау беру, интерфейстің стилі және жүзеге асыру стилі программаны пайдалануды жеңілдетеді. Егер код қисынды болса, онда программалаушы ірі жүйенің әрбір бөлігіне қатысты жаңа келісімдер топтамасын оқып үйренуге мәжбүр болмайды. Мұның мысалы STL кітапханасы (20-21 тарауларды, Б.4-6 бөлімін қ.) болып табылады. Егер қисындылық жабдықтамасыз етілмесе (мысалы, ескі кодтың болуына қарай немесе басқа тілде жазылған код бар болса), онда стильдің программаның басқа бөлігімен үйлесімділігін жабдықтамасыз ететін интерфейс құрған жөн. Әйтпесе, бұл бөтен код ("келеңсіз", "жаман") оны пайдалануға мәжбүр болған программаның әрбір бөлігіне өзінің залалын (зиянкестігін) "жұқтырады".

Минимализм мен қисындылықты жабдықтамасыз ету үшін әрбір интерфейсті мұқият (және тізбекті түрде) құжаттау керек. Мұндайда кодтың үйлеспегенін және қайталануын аңғару оңай болады. Алдыңғы, соңғы шарттарды және инварианттарды құжаттау аса пайдалы болуы мүмкін, өйткені олар ресурстарды басқару мен қателерді хабарлауға аса назар аударады. Қателерді қисынды (логикалық) түрде өңдеу мен ресурстарды басқарудың үйлесімді стратегиясы программаның қарапайымдылығын жабдықтамасыз ету жолында маңызды рөл атқарады (19.5 бөлімін қ.).

Кейбір программалаушылар жобалаудың KISS ("Keep It Simple, Stupid" - "Ақымағым-ау, қарапайым етіп жаса") принципін ұстанады. Бізге KISS принципі – жобалаудың назар аударарлық жалғыз ғана принципі дегенді де естуге тура келген. Дегенмен біз мұнан гөрі бәсеңдеу тұжырымға басымдық бергенді дұрыс көреміз, мысалы, "Keep simple things simple" ("Қарапайым нәрсені қиындатпа") және "Keep it simple: as simple as possible, but no simpler" ("Барлығы да қарапайымырақ емес, барынша қарапайым болуы тиіс"). Соңғысы – Альберт Эйнштейннің айтқаны (Albert Einstein). Ол адами ақыл-ой шегінен тысқары тұрған және жобаны бұзатын, шамадан тыс қарапайымдылыққа назар аударады. Бұдан мынадай сауал туындайды: "Кім үшін жасаймыз және де немен салыстырамыз?"

22.1.3 Стильдер мен парадигмалар

Біз программаны жобалау және оны жүзеге асыру кезінде тізбекті түрдегі стильді ұстанғанымыз жөн. C++ тілі іргелі деп санауға болатын төрт негізгі стильдерден тұрады.

- Процедуралық программалау.
- Мәліметтер абстракциясы.
- Объектіге бағытталған программалау.
- Жалпыланған программалау.

Кейде бұларды программалаудың (аздап әсірелеп айтылған сияқты) парадигмалары деп атайды. Тағы бірнеше парадигмалар бар, мысалы: функционалдық программалау (functional programming), логикалық программалау (logic programming), ережелерге бағытталған программалау (rule-based programming), шектелген программалау (constraints-based programming) және аспектіге бағытталған программалау (aspect-oriented programming). Дегенмен C++ тілі бұл парадигмаларды тікелей сүйемелдемейді және біз бір кітап аясында олардың барлығын қарастыра да алмаймыз, сондықтан оларды болашақтың үлесіне қалдырамыз.

- *Процедуралық программалау.* Бұл парадигманың негізгі идеясы – программаны аргументтерге қолданылатын функциялардан құрастыру. Мұның мысалы ретінде **`sqrt()`** және **`cos()`** сияқты математикалық функциялар кітапханасы қарастырылады. C++ тілінде бұл программалау стилі функцияларды қолдануға негізделген (8 тарауды қ.). Мүмкін, мұның ең бағалысы болып, аргументтерді мәндері бойынша, сілтеме бойынша және **`константалық`** сілтеме бойынша беру механизмдерін таңдау мүмкіндігі табылады. Көбінесе мәліметтер құрылымға **`struct`** конструкциясы арқылы ұйымдастырылады. Абстракцияның тікелей механизмдері (мысалы, жабық мәлімет-мүшелер және кластың функция-

мүшелері) қолданылмайды. Осы программалау стилі және функциялар кез келген басқа стильдің интегралдық бір бөлігі болып табылады.

- *Мәліметтер абстракциясы.* Бұл парадигманың негізгі идеясы – алдымен пәндік аймақ үшін типтер топтамасын құру, сонан соң оларды пайдаланатын программалар жазу (24.3-24.6 бөлімдері). Мәліметтерді тікелей түрде жасыру тәсілі кеңінен қолданылады (мысалы, кластың жабық мүшелерін қолдану). Мәліметтер абстракциясының кең тараған мысалы болып, жалпыланған программалауда қолданылатын мәліметтер абстракциясы мен параметрлеудің арасындағы тығыз тәуелділікті көрсететін `string` және `vector` стандартты кластары саналады. "Абстракция" сөзінің осы парадигманың атауында қолданылатын себебі, мұндағы типтермен өзара әрекеттесу оның жүзеге асуына тікелей қатынасу арқылы емес, интерфейстің көмегімен жасалады.

- *Объектіге бағытталған программалау.* Программалаудың осы парадигмасының негізгі идеясы – иерархиядағы типтердіің бір-бірімен қатынасын тікелей кодта көрсету үшін ұйымдастыру. Мұның классикалық мысалы – 14-тарауда сипатталған `Shape` иерархиясы. Бұл тәсілдің айқын бағасы типтердің нақты иерархиялық өзара әрекеттесуі орын алған жағдайда анық көрінеді. Дегенмен оны шамадан тыс көбірек қолдануға деген тенденция бар; басқаша айтқанда, адамдардың бұған қажетті себептері болмаса да, типтер иерархиясын құруға талпынып жатады. Егер адамдар туынды типтерді құрып жатса, онда мынандай сұрақ қойыңыз: "Неге?" Бұл нені өрнектейді? Базалық және туынды кластардың арасында айырмашылық осы нақты жағдайда маған қандай көмек көрсете алады?

- *Жалпыланған программалау.* Программалаудың бұл парадигмасының негізгі идеясы – нақты бір алгоритмдерді алып, алгоритмнің болмысын өзгертпей, олардағы типтерді өзгертуге мүмкіндік беретін параметрлерді қосып, абстракцияның ең жоғарғы деңгейіне көтеру. Осындай абстракцияның деңгейін көтерудің қарапайым мысалы 20-тарауда сипатталған `high()` функциясы болып табылады. Кітапхананың `find()` және `sort()` алгоритмдері жалпыланған программалау арқылы жалпы түрде көрсетілген іздеу мен сұрыптаудың классикалық алгоритмдері болып табылады. 20-21-тарауларда көрсетілген мысалдарды қараңыз.

Сонымен, қорыта келе айтарымыз, адамдар көбінесе программалау стильдері (парадигмалары) туралы, олар бір-біріне қарама-қарсы баламалардан тұрады дегенді жиі айтады: яғни сіз жалпыланған программалауды немесе объектіге бағытталған программалауды қолданасыз. Егер есептің шешімін ең жақсы түрде көрсеткіңіз келсе, онда осы стильдердің араласқан комбинациясын қолданыңыз. "Ең жақсы түрде" деген сөз, сіз жазған программаны жеңіл оқып, жазып, жеңіл пайдалануға болады және ол белгілі дәрежеде өте тиімді деген мағынаны береді.

Мысал қарастырайық: объектіге бағытталған программалаудың жүзеге асуының жақсы бір мысалы болып саналатын Simula тілінде туындаған классикалық `Shape` класы (22.2.6 бөлім). Оның бірінші шешімі былай көрсетіледі:

```
void draw_all(vector<Shape*>& v)
{
      for(int i = 0; i<v.size(); ++i) v[i]->draw();
}
```

Бұл кодтың фрагменті "жеткілікті түрде объектіге бағытталған" болып көрінеді. Ол кластардың иерархиясы мен виртуалды функцияны шақыруға негізделген, мұнда **draw()** функциясы әрбір нақты **shape** класының объектісі үшін автоматты түрде табылады; басқаша айтқанда, **Circle** класының объектісі үшін ол **Circle::draw()** функциясын шақырады, ал **Open_polyline** класының объектісі үшін **Open_polyline::draw()** функциясын шақырады. Дегенмен **vector<Shape*>** класы, негізінде, жалпыланған программалаудың конструктивті элементі болып табылады: ол компиляциядан өту кезеңінде анықталатын параметрді (элемент типі) пайдаланады. Барлық элементтері бойынша итерациялау үшін мұнда стандартты кітапханадан алынған алгоритм қолданылатындығын атап айтқан жөн.

```
void draw_all (vector<Shape*>& v)
{
      for_each(v.begin() ,v.end() ,mem_fun(&Shape::draw));
}
```

for_each() функциясының үшінші аргументі болып, алғашқы екі аргументпен берілген (Б.5.1 бөлімі), тізбектің әрбір элементі үшін жеке шақырылатын функция саналады. Үшінші функция, **p->f()** синтаксистік конструкциясының көмегімен шақырылатын функция-мүше емес, **f(x)** синтаксистік конструкциясы арқылы шақырылатын қарапайым функциядан (немесе функция-объектіден) тұрады деп есептеледі. Сондықтан біз шынымен де функция-мүшені шақырғымыз келетінін көрсету үшін (**Shape::draw()** виртуалды функциясын) стандартты кітапхананың **mem_fun()** (Б.6.2 бөлімі) функциясын қолдану қажет. **for_each()** және **mem_fun()** функциялары шаблондық болғандықтан, негізінде, объектіге бағытталған парадигмаға онша жақсы сәйкес келмейді; олар толығымен жалпыланған программалауға жатады. Одан да қызықтысы – **mem_fun()** функциясы кластың объектісін қайтаратын автономды (шаблонды) функция болып табылады. Басқаша айтқанда, оны мәліметтердің қарапайым абстракциясына (мұралау жоқ) немесе тіпті процедуралық програмалауға (мәліметтерді жасыру жоқ) жатқызған жөн. Сонымен, біз C++ тілі сүйемелдейтін төрт іргелі программалау стилінің барлығын да кодтың тек қана бір жолы пайдаланып тұрғандығын айта аламыз.

Неліктен біз барлық фигуралардың суретін салу үшін мысалдың екінші нұсқасын жаздық? Негізінде, оның бірінші нұсқадан ешқандай айырмашылығы жоқ, оған қоса ол бірнеше символға ұзынырақ! Өз әрекетіміздің дұрыстығын көрсету үшін, цикл концепциясын **for_each()** функциясы арқылы өрнектеу

жалпы дұрыс болғанымен, ол **for** циклына қарағанда қателерге онша төтеп бере алмайтынын айтайық, бірақ осы аргумент көптеген адамдар үшін онша сенімді болып табылмайды. Бұл жерде **for_each()** функциясының біз мұны қалай жасайтынымызды емес, мұны қалай жасағымыз келетінін (тізбекпен жүріп өту) көрсетеді деп айтқан дұрыс. Дегенмен адамдардың көбіне "Бұл пайдалы" деп айтқан жеткілікті. Осындай жазба көптеген мәселелерді шешуге мүмкіндік беретін жалпылау жолын (жалпыланған программалаудың жақсы дәстүрлері түрінде) көрсетеді. Неліктен барлық фигуралар, тізімде немесе жалпыланған тізбекте емес, векторда сақталады? Бұған жауап беру үшін біз кодтың үшінші нұсқасын (бұрынғыдан анағұрлым жалпы түрде) жаза аламыз:

```
template<class Iter> void draw_all(Iter b, Iter e)
{
for_each(b,e,mem_fun(&Shape::draw));
}
```

Енді бұл код фигуралардың барлық тізбектері түрлерімен жұмыс жасайды. Мысалы, біз оны тіпті **Shape** класының объектілері жиымының барлық элементтері үшін шақыра аламыз:

```
Point p(0,100);
Point p2(50,50) ;
Shape*a[]={newCircle(p,50),newTriangle(p,p2,Point(25,25))};
draw_all(a,a+2);
```

Жақсы термин болмағандықтан, біз мұны қолайлы стильдердің қоспасын пайдаланатын *мультипарадигмалық программалау* (multi-paradigm programming) деп атаймыз.

22.2 Программалау тілдерінің тарихына шолу

Адамзат дамуының басталуы кезеңінде программалаушылар нөлдер мен бірліктерді таста қашап жаза бастады. Жарайды, біз аздап асыра сілтеп жібердік! Осы тарауда біз бас жаққа қайта ораламыз да, программалау тілдерінің қысқаша тарихын C++ тілімен байланыстыра отырып сипаттаймыз.

Көптеген программалау тілдері бар. Олардың пайда болу жылдамдығы, шамамен он жыл ішінде 2000 тілдей болады, олардың жоғалу жылдамдығы да осындай шамада. Бұл бөлімде біз соңғы алпыс жылдың ішінде пайда болған он шақты тілдерді қарастырамыз. Толығырақ ақпаратты мына веб-парақтан: http//research.ihost.com/hopl/HOPL.html табуға болады, сонда *ACM SIGPLAN HOPL* (*History of Programming Languages* – программалау тілдерінің тарихы) үш конференциясында жарияланған барлық мақалаларға сілтемелер көрсетілген. Бұл

мақалалар қатаң сын-пікірден өткен, яғни олар веб желісінде көрсетілген орташа статистикалық ақпарат көздеріне қарағанда, анағұрлым толық және сенімді болып табылады. Біз талқылайтын тілдердің барлығы да HOPL конференциясында көрсетілген. Мақаланың толық атын іздеу веб-машинасында теру жолымен сіз оны оңай таба аласыз. Оған қоса, осы бөлімде көрсетілген компьютер ғылымдарының көптеген мамандарының, олардың жұмыстары туралы көптеген ақпаратты табуға болатын үй парақтары бар.

Біз бұл тарауда тілдердің өте қысқаша сипаттамаларын ғана келтіреміз, өйткені әрбір аталған тіл (және жүздеген аты аталмаған) жеке кітаптан тұруға лайық. Әрбір тілде біз ең бастысын ғана таңдадық. Оқырмандар мұны "X тілі туралы айтылатынның бәрі осы!" деп ойламай, өз бетінше іздеуге шақыру деп түсінеді деп үміттенеміз. Мұнда әрбір еске алынатын тіл өз заманында үлкен жетістік болып саналған болатын және программалауға маңызды үлесін қосқанын есіңізге саламыз. Орынның жетіспеушілігінен осы тілдерге қатысты лайықты өз бағасын бере алмаймыз, бірақ оларды атап өтпеу әділетсіздік болар еді. Біз осы тілдердің әрқайсысында жазылған бірнеше жол кодты да келтіргіміз келді, бірақ, өкінішке орай оған да орын жеткіліксіз болды (5-және 6-жаттығуды қ.).

Көбінесе артефактілер туралы (мысалы, программалау тілдері туралы) олар нені көрсетеді немесе анонимді процесті құрудың нәтижелері сияқты деп қана айтылады. Бұл тарихты дұрыс баяндау емес: көбінесе, әсіресе бірінші кезеңде тілге идеялар, жұмыс орны, жеке таңдаулар және бір немесе бірнеше (басқаларынан жиірек) адамның ішкі шектеулері әсер етеді. Осылайша әрбір тілдің артында нақты бір адамдар тұрады. IBM және Bell Labs компаниясы да, Cambridge University мекемесі де, басқа ұйымдар да программалау тілдерін жасамайды, оны осы ұйымдарда жұмыс істейтін адамдар, әдетте олар өзінің достарымен және әріптестерімен ынтымақтаса отырып жасайды.

Тарихты бұрмалаған көзқарасқа жиі әкелетін күлкілі бір феноменді атап өткен жөн болар. Атақты оқымыстылардың және инженерлердің суреттері, олар белгілі болып, ұлттық академияның, Корольдік қоғамның, Киелі Джонның серілері, Тьюринг сыйлығынының лауреаттары және т.с.с. қоғамдастықтардың атақты мүшелері болғанда ғана жасалатын болған. Басқаша айтқанда, олар түскен суреттерде өздерінің елеулі жаңалықтарын ашқан кезге қарағанда, анағұрлым (ондаған жастарға) үлкен болып келеді. Олардың көбісі қартайғанша өнімді еңбек етті. Дегенмен, бізге ұнаған тілдер мен программалау әдістерінің пайда болған жылдарына көз сала отырып, мынадай жайттарды айтуға болады. Осы жаңалықтарды ашқан ғалымдар байырғы кезде, өзінің құрбысын жақсы мейрамханаға шақыру үшін ақшасы жететіндігін анықтауға талпынған жас жігіт (ғылым мен техникада әлі күнге дейін әйел адамдар аз) болғанын немесе конференцияда маңызды жұмысының презентациясын жасау мен еңбек демалысын қалай біріктіретінін ойлап жүрген жас отбасының иесі болғанын көзге елестетуге тырысыңыз. Ақ сақалдылар, шашы селдір бастар және сәнсіз костюмдер киген сурет иелері кезінде көзді тартқан көрікті жандар болғаны естеріңізде болсын.

22.2.1 Алғашқы программалау тілдері

1948 жылы программаларды есте сақтай алатын алғашқы электронды компьютерлер пайда болған кезде, олардың әрқайсысының өзінің жеке программалау тілі болдцы. Алгоритмді бейнелеу (мысалы, планета орбитасын есептеу) мен нақты бір машинаның нұсқаулары арасында өзара тікелей сәйкестік болды. Әрине, ғалымдар (көбінесе қолданушылар болып ғалымдар отыратын) математикалық формулалар жазатын еді, ал программа машинаның нұсқаулар тізімінен құрастырылатын. Алғашқы қарапайым тізімдер машина жадында көрсетіліміне тікелей сәйкес келетін ондық немесе сегіздік сандардан тұрды. Кейінірек ассемблер мен "автокодтар" пайда болды; басқаша айтқанда, адамдар машиналық нұсқаулар мен құралдар (мысалы, регистрлер) үшін символдық атаулардан тұратын тілдерді жасап шығарды. Сонымен, программалаушы компьютер жадының 123 адресінде орналасқан мәліметті 0-регистрге жүктеу үшін, "LD R0 123" деп жазуы керек еді. Бірақ әрбір машина бұрынғыдай өзінің жеке нұсқаулар жиынынан және өз программалау тілінен тұратын болатын.

Сол кездегі программалау тілдерін құрастырушылардың айқын өкілі ретінде Кембридж университетінің компьютерлік зертханасындағы (University of Cambridge Computer Laboratory) Дэвид Уилер (David Wheeler) болған еді. 1948 жылы ол компьютер жадында сақталып орындалған алғашқы нақты программаны жазып шығарған болатын (квадраттар кестесін есептейтін программа; 4.4.2.1 бөлімін қ.). Ол алғашқы компиляторды (машинаға тәуелді автокод үшін) жасап шығарғанын хабарлаған он адамның біреуі болды, функцияны шақыруды да ойлап тапты (иә, осындай анық және қарапайым түсінік те бір кезде алғашқы рет ойлап табылған еді). Ол 1951 жылы кітапханаларды құрастыру туралы өз уақытынан жиырма жылға алға кеткен керемет мақала жазды! Морис Уилкс (Maurice Wilkes, жоғарыға қ.) және Стенли Гиллмен (Stanley Gill) біріге отырып программалау туралы алғашқы кітап жазып шығарды. Ол компьютерлік ғылымдар (Кембридж университетінде 1951 жылы) аймағында бірінші болып философия докторы дәрежесін алды, ал кейінірек аппараттық жабдықтамалар (кэш-архитектура және алғашқы жергілікті

желі) мен алгоритмдердің (мысалы, TEA шифрлеу алгоритмі, 25.5.6 бөлімін қ.) және Бэрроуз-Уилер түрлендіруінің дамуына (Burrows-Wheeler transform – bzip2 архиваторында қолданылған қысу алгоритмі) зор үлес қосты. Дэвид Уилер Бьярне Страуструптың (Bjarne Stroustrup) докторлық диссертациясының ғылыми жетекшісі болды. Тарихқа көз салсаңыз, компьютерлік ғылымдар – кейінірек пайда болған жас пәндер тобына жатады. Дэвид Уилер өзінің асқан үздік жұмысының басым бөлігін аспирант болып жүріп-ақ орындаған еді. Кейіннен ол Кембридж университетінің профессоры және Корольдік қоғамның мүшесі болып (Fellow of the Royal Society) тағайындалды.

Сілтемелер

Burrows, M., and David Wheeler. "A Block Sorting Lossless Data Compression Algorithm." Technical Report 124, Digital quipment Corporation, 1994.

Bzip2 link: www.bzip.org.

Cambridge Ring website: http://koo.corpus.cam.ac.uk/proj ects/earlyatm/cr82.

Campbell-Kelly, Martin. "David John Wheeler". *Biographical Memoirs of Fellows of the Royal Society*, Vol. 52,2006. (Оның қысқаша биографиясы).

EDSAC: http: //en.wikipedia.org/wiki/EDSAC

Knuth, Donald. *The Art of Computer Programming*. Addison-Wesley, 1968, and many revisions. Look for "David Wheeler" in the index of each volume

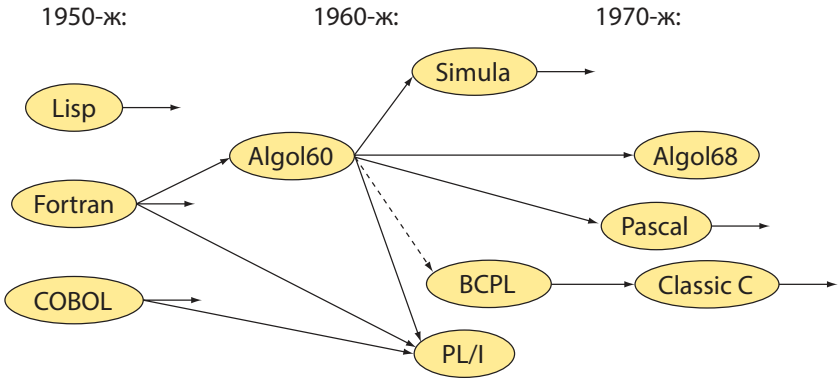
TEA link. http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm.

Wheeler, D J. "The Use of Sub-routines in Programmes.' Proceedings of the 1952 ACM National Meeting. (Бұл 1951 жылдан басталған техникалық есептер кітапханасы)

Wilkes, M. V., D. Wheeler, and D.J. Gill. *Preparation of Programs for an Electronic Digital Computer.* Addison-Wesley Press, 1951; 2nd edition 1957. Программалау туралы бірінші кітап.

22.2.2 Қазіргі программалау тілдерінің түп-тамыры

Төменде алғашқы негізгі тілдер диаграммасы келтірілген:



Осы тілдердің маңыздылығы олардың кең қолданылуымен түсіндіріледі (олардың кейбіреулері қазір де қолданылады), бірсыпыра тілдер қазіргі маңызды тілдердің негізін қалаушылар болып табылады, мұрагер тілдер көбінесе сол бұрынғы атауларын алып қалды. Бұл бөлім алғашқы үш тілдің – Fortran, COBOL және Lisp сияқты тілдердің шығуына себепші болған, программалау тілдерінің атасы тәрізді тілдерге арналған.

22.2.2.1 Fortran программалау тілі

1956 жылы Fortran тілінің пайда болуы программалау тілдерін жасау ісіндегі ең елеулі қадам болды. Fortran сөзі "Formula Translation" сөздерінің қысқаша жазылуынан шыққан. Оның негізгі идеясы компьютерге емес, адамдарға арналған тиімді машиналық код құру (генерациялауға) болып табылады. Fortran тілінде қабылданған белгілеу жүйесі, электрондық компьютерлердің машиналық нұсқауларынан (сол кезде енді ғана пайда болған) гөрі математикалық есептерді шығаратын ғалымдар мен инженерлердің қолданған жүйесін еске салады.

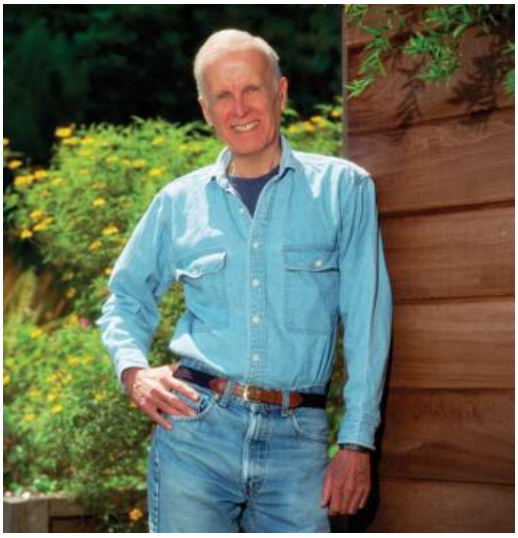
Қазіргі кездегі көзқарас бойынша, Fortran тілін пәндік аймақтарды код арқылы көрсетудің алғашқы тікелей әрекеті ретінде қарастыруға болады. Ол программалаушыларға, сызықты алгебраның операцияларын олардың дәл оқулықта сипатталғанындай түрде орындауға мүмкіндік берді. Fortran тілінде жиымдар, циклдар және стандартты математикалық формулалар (`x+y` және `sin(x)` сияқты стандартты математикалық белгілеулерді қолданатын) бар. Тілде математикалық функциялардың стандартты кітапханасы, енгізу-шығару механизмдері болды, оған қоса қолданушы өздігінен қосымша функциялар мен кітапханаларды да жазып анықтай алатын еді.

Белгілеулер жүйесі де жоғарғы дәрежеде машинадан тәуелсіз болды, сондықтан Fortran тілінде жазылған кодты қиындықсыз бір компьютерден екінші компьютерге аз ғана өзгерістер арқылы ауыстыруға болатын. Бұл сол кездегі ең үлкен жетістік еді. Осындай себептерге байланысты Fortran тілі ең алғашқы жоғарғы деңгейдегі программалау тілі болып саналады.

Fortran тілінде жазылған бастапқы кодты негізге ала отырып жасалған машиналық код, тиімділік тұрғысынан алғанда, оптималды кодқа жақын болса, ол маңызды болып саналды: машиналар аса үлкен және өте қымбат болды (программалаушылар ұжымының еңбекақысынан көптеген есеге артық), тым (қазіргі өлшемдер тұрғысынан) баяу (секундына шамамен 100 мың операция) жұмыс істеді және өте аз жады көлемін пайдаланды (8 Кб). Дегенмен, адамдар осындай машиналарға арнап пайдалы программалар жаза білді және бұл белгілеу жүйесінің жақсарған түрін қолдануға (программалаушы жұмысының өнімділігін арттыруға және программаларды басқа жерлерге тасымалдаудың нығаюына) шектеу қойып отырды.

Fortran тілі ғылыми-инженерлік есептерді шығару үшін өте қолайлы тіл ретінде қолданылды, ол негізінде, осындай мақсаттар үшін жасалған болатын. Өзінің пай-

да болған кезінен бастап ол үнемі дамып отырды. Fortran тілінің негізгілері болып II, IV, 77, 90, 95 және 03 нұсқалары саналады, осы кезге дейін тілдің қай нұсқасы (Fortran77 немесе Fortran90) жиі қолданылады деген дау-дамай әлі жалғасуда.



Fortran тілінің алғашқы анықталуы мен жасалуы Джон Бэкустің (John Backus) жетекшілігімен IBM компаниясының ұжымында жүзеге асырылды: "Біз нені қалайтынымызды және қалай жасайтынымызды білмедік. Тіл біртіндеп дами берді". Осы уақыт кезеңіне дейін ешкім мұндайды жасамаған еді, олар ақырындап компилятордың негізгі құрылымын: лексикалық, синтаксистік және семантикалық талдау арқылы, соған қоса тілді оңтайландыру жолымен де жетілдіріп отырды. Осы күнге дейін Fortran тілі математикалық есептеулерді орындау аймағының көш басшысы болып отыр. Fortran тілінен кейін ашылған жаңалықтар ішінде арнайы грамматика үшін жасалған белгілеулер жүйесі – Бэкус-Наур формасы (Backus-Naur Form – BNF) болды. Ол алғаш рет Algol-60 тілінде пайдаланылды (22.2.3.1 бөлімін қ.) және ол осы уақытта да қазіргі тілдердің көбінде қолданылады. Біз 6 және 7-тарауларда сипатталған BNF формасының бір нұсқасын өз грамматикамызда пайдаланған болатынбыз.

Бірсыпыра уақыттан кейін Джон Бэкус программалау тілдерінің жаңа бір даму аймағын (функционалдық программалау) дүниеге келтірді, ол жады ұяшықтарының ішіндегі мәліметті оқу немесе жазуға негізделген машинаға бағытталған көзқарастан басқаша, программалаудағы математикалық тәсілдерді пайдалануға негізделген болатын. Таза математикада меншіктеу және оператор деген ұғымдардың жоқ екендігін атап өткен жөн. Оның орнына сіз "жай" ғана белгілі бір шарттар орындалғанда ненің ақиқат болуы керек екендігін көрсететін едіңіз. Функционалдық программалаудың кейбір негіздері Lisp тілінде (22.2.2.3 бөлімін қ.) өз жалғасын тапты, ал оның басқа бір идеялары STL кітапханасында пайдаланыла бастады (21-тарауды қ.).

Сілтемелер

Backus, John. "Can Programming Be Liberated from the von Neumann Style?" *Communications of the ACM, 1977*. (Тьюринг сыйлығының берілуіне орай оқылған дәрісі)

Backus, John. "The History of FORTRAN I, II, and III." *ACM SIGPLAN Notices,* Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

Hutton, Graham. *Programming in Haskell*. Cambridge University Press, 2007. ISBN 0521692695.

ISO/IEC 1539. *Programming Languages – Fortran*. (The "Fortran 95" standard)

Paulson, L. C. *ML for the Working Programmer*. Cambridge University Press, 1991. ISBN 0521390222.

22.2.2.2 COBOL программалау тілі

Бизнеске байланысты есептерді шығаратын программалаушылар үшін COBOL (Common Business-Oriented Language – коммерцияға және іске байланысты есептерге арналған программалау тілі) тілі (кейбір жерлерде осы уақытқа дейін қолданыста), ғылыми есептерді шығаратын программалаушыларға арналған Fortran тілі (бұл да қолданыста) сияқты рөл атқарды. Бұл тілдің негізгі әрекеттері мәліметтер тобымен мынандай жұмыс түрлерін орындауға арналған болатын:

- Көшіру.
- Сақтау және іздеу (жазбаларды сақтау).
- Баспаға шығару (есеп берулер).

Мұнда санаулар мен есептеулер екінші дәрежелі мәселелер ретінде қарастырылды (COBOL тілін пайдаланатын қосымшалар үшін толығымен ақталған). Кейбіреулер, тіпті COBOL тілін бизнестік ағылшын тіліне жақын, мұнда тіпті менеджерлер де программалай алады және жақында программалаушылар қажет болмай қалады деген тұжырымдар жасады. Менеджерлер көп жылдар бойы осы үміт жетегінде жүріп, программаушылар көмегін керек етпейтін кезге жетіп, қаржы үнемдейміз деп күтіп жүрді. Бірақ ол еш уақытта жүзеге аспады және оған деген талпыныс та болған жоқ.

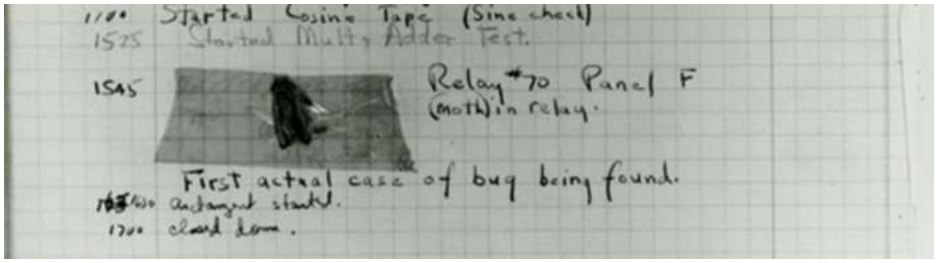
Бастапқыда COBOL тілін АҚШ қорғаныс министрлігінің (U.S. Department of Defense) бастамасымен 1959-60 жылдары CODASYL комитеті және коммерциялық есептерге байланысты есептеулерді орындайтын негізгі компьютерлер шығаратын топтар жасап шығарды. Бұл жоба Грейс Хоппер жасап шығарған FLOW-MATIC тіліне негізделген еді. Тілді дамытуға оның қосқан үлесінің бірі ағылшын тіліне жақын синтаксисті қолдану болды (Fortran тілінде қабылданған математикалық белгілеулерден айырмашылығы және осы уақытқа дейін қолданыста болуы). COBOL тіліне де, табысты түрде дамыған Fortran программалау тілі тәрізді, үздіксіз өзгерістер мен толықтырулар енгізіліп отырды. Тілдің негізгі нұсқалары 60, 61, 65, 68, 70, 80, 90 және 04 сияқтылар болды.

Грейс Мюррей Хоппер (Grace Murray Hopper) Иель университетінде (Yale University) математика бойынша философия докторы атағын алды. Екінші дүниежүзілік соғыс кезінде ол АҚШ-тың әскери-теңіз флотындағы алғашқы компьютерлерде жұмыс істеді. Компьютерлік өндірістің жаңа ғана туындаған ортасында бірнеше жыл жұмыс істегеннен кейін, ол әскери-теңіз флотындағы қызметіне қайта оралды.

"Контр-адмирал доктор Грейс Мюррей Хоппер (АҚШ-тың әскери-теңіз флоты) алғашқы компьютерлердегі программалауда үлкен жетістікке жеткен керемет әйел еді. Өзінің бүкіл өмір жолында ол программалық жабдықтаманы жобалау концепциясын құру аймағында басшы болды және қарапайым программалау әдістерінен күрделі компиляторларды қолдануға өту ісіне үлкен үлес қосты. Ол "біз әрқашан осылай жасайтынбыз" деген ұран ештеңені ауыстырмау үшін жасалған іске жақсы негіздеме емес екендігіне сенді.

— Анита Борг (Anita Borg) "Grace Hopper Celebration of Women in Computing" конференциясындағы сөзінен, 1994 ж.

Грейс Мюррей Хоппер – компьютердегі қатені ең бірінші болып "қоңыз" (bug) деп атаған адам. Ол осы терминді жиі пайдаланды және құжаттарда да оның орынды екенін дәлелдеп шықты.



Шынында да, қоңыз (күйе) программалық және аппараттық жабдықтамаға тікелей әсер етті. Қазіргі кездерде де "қоңыздардың" көбі программалық жабдықтамаларға ұя салып, сырт көзге онша тиімді болып көрінбей келеді.

Сілтемелер

Г.М.Хоппер биографиясы:" http://tergestesoft.com ~eddysworld/hopper.htm. ISO/IEC 1989 2002. *Information Technology – Programming Languages – COBOL.*
Sammet, Jean E. "The Early History of COBOL". *ACM SIGPLAN Notices*, Vol. 13 No.8,1978. Special Issue: History of Programming Languages Conference.

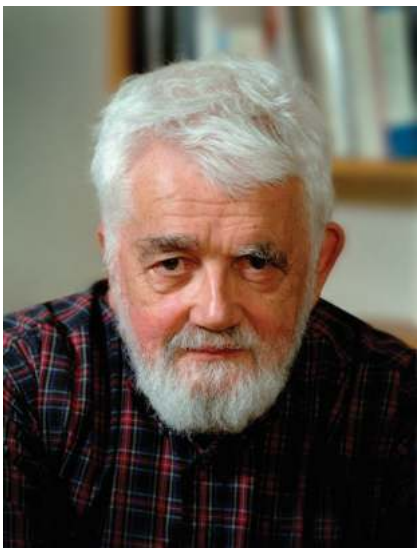
22.2.2.3 Lisp программалау тілі

Lisp тілін алғашқы рет 1958 жылы Массачусет технологиялық институтында (MIT) Джон Маккарти (John McCarthy) байланысқан тізімдер мен символдық ақпаратты өңдеу (сондықтан оны LIST Processing деп атады) үшін жасап шығарған болатын. Басында ол компилятор емес, интерпретатор (әлі де солай жалғасуда) арқылы орындалған еді. Lisp тілінің ондаған (жүздеген болуы да мүмкін) диалектілері бар. Көбінесе Lisp тілінің әртүрлі жүзеге асырылған нұсқалары бар деп айтылады. Оның қазіргі кездегі ең белгілі диалектілері болып Common Lisp және Scheme тілдері саналады.

Тілдердің осы тобына жататындар жасанды интеллект аймағындағы зерттеулердің негізі болды және болып келеді де (бірақ жұмыс істеп жүрген программалардың көбісі C немесе C++ тілдерінде жазылған). Lisp тілін құрушылар үшін негізгі шабыт көздерінің бірі лямбда-есептеуі болды (анығырағы, оның математикалық сипатталуы).

Fortran және COBOL тілдері өздеріне сәйкес пәндік аймақтарда нақты мәселелерді шешу үшін арнайы құрылған болатын. Lisp тілін құрастырушылар мен қолданушылар программалау мен программаның сәнділігіне көп көңіл бөлді. Олардың еңбегі жиі-жиі жетістіктерге жеткізіп тұрды. Lisp тілі аппараттық жабдықтамадан тәуелсіз тілдердің алғашқысы болды, оның үстіне оның семантикасы математикалық формада берілді. Қазіргі кезде Lisp тілінің қолдану аймағын анықтау қиын, жасанды интеллект және символдық есептеулерді, бизнестік есептер немесе ғылыми программалаудағы сияқты, нақты бір есептерге айқын түрде жобалауға болмайды. Lisp тілінің идеясын (Lisp тілін құрушылар қоғамдастығы және пайдаланушылар) көптеген қазіргі программалау тілдерінде, әсіресе функционалдық программалауда жиі кездестіруге болады.

Джон Маккарти математика саласы бойынша бакалавр дәрежесін Калифорнияның технологиялық институтында (California Institute of Technology), ал математика бойынша философия докторының дәрежесін Принстон университетінде (Princeton University) алған болатын.



Программалау тілдерін құрастырушылардың арасында математиктердің көп болғандығын айта кету керек. Маккарти MIT-тегі жемісті еңбегінен кейін, 1962 жылы жасанды интеллекті зерттейтін зертхананың (Stanford AI lab) негізін салуға қатысу үшін Станфордқа көшіп келді. Ол *"жасанды интеллект"* (artificial intelligence) терминін ойлап тапты, сонымен қатар осы аймақтағы зерттеулерде зор жетістіктерге жетті.

Сілтемелер

Abelson, Harold, and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*, Second Edition. MIT Press, 1996. ISBN 0262011530.

ANSI INCITS 226-1994 (formerly ANSI X3.226:1994). *American National Standard for Programming Language - Common LISP.*

McCarthy, John. "History of LISP." *ACM SIGPLAN Notices,* Vol. 13 No. 8,1978. Special Issue: History of Programming Languages Conference.

Steele, Guy L.Jr. *Common Lisp: The Language*. Digital Press, 1990. ISBN 1555580416.

Steele, Guy L.Jr., and Richard Gabriel. "The Evolution of Lisp". Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

22.2.3 Algol тілдерінің топтамасы

1950 жылдардың соңында көптеген мамандар программалаудың өте қиындап кеткендігін, оның арнайы сала болғанымен, аса ғылыми емес жұмыс екендігі

сезіле бастады. Программалау тілдерінің тым көп түрлі болып кеткені айтылып, енді оның ортақ элементтерін біріктіре отырып, іргелі принциптер негізінде, барлығын да бір тілге біріктіру керек деген сенім пайда болды. Осы идея ойда жүргенде, бір топ ғалымдар IFIP (International Federation of Information Processing – Ақпаратты өңдеудің халықаралық федерациясы) аясында жинала отырып, бірнеше жыл ішінде программалау аймағында төңкеріс жасаған жаңа тіл жасап шығарды. Қазіргі тілдердің көбісінің, оның ішінде C++ тілі де бар, кең қолданылып отырғаны осы жобаның арқасы деп айтуға болады.

22.2.3.1 Algol-60 программалау тілі

IFIP 2.1 тобы жұмысының нәтижесі болған "Алгоритмдік тіл" ("ALGOrithmic Language" – Algol), қазіргі программалау тілдерінің жаңа бір концепцияларын ашты.

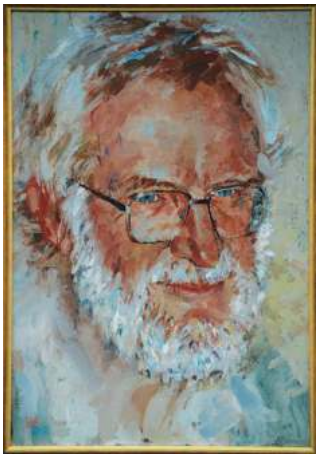
- Лексикалық талдаудың ішкі мәтіні (контексті).
- Тілді анықтау үшін грамматиканы қолдану.
- Синтаксистік және семантикалық ережелерді айқындап бөлу.
- Тілдің анықтамасы мен оның жүзеге асырылуын айқындап бөлу.
- Типтерді жүйелі түрде қолдану (статикалық түрде, яғни компиляция кезеңінде).
- Құрылымдық программалауды тікелей түрде сүйемелдеу.

"Әмбебап программалау тілі" деген ұғымның өзі Algol тілімен бірге келді. Осы уақытқа дейін тілдер ғылыми есептеулерге (мысалы, Fortran), бизнестік есептеулерге (мысалы, COBOL), тізімдерді өңдеуге (мысалы, Lisp), модельдеуге және т.б. арналған болатын. Algol 60 тілі осы аталғандардың барлығының ішіндегі Fortran тіліне жақын болып шықты.

Өкінішке орай, Algol-60 тілі ешқашан да академиялық ортадан тысқары ортада көп пайдаланылмады. Ол көп адамдарға қызықты тіл болып көрінді. Fortran тілін қалайтын программалаушылар Algol-60 тіліндегі программалар өте баяу жұмыс істейді деді; Cobol тілінде жұмыс істейтін программалаушылар Algol-60 тіліндегі программалар бизнестік ақпаратты өңдеуді қолдамайды десе, Lisp тілінде жұмыс істейтін программалаушылар Algol-60 тілінде икемділік жетіспейді деп, ал қалған адамдардың көбісі (программалық жабдықтаманы құруға берілетін инвестицияны басқаратын менеджерлерді қоса) ол мүлдем академиялық деп есептеді; ақырында көптеген американдықтар оны европалық тіл деп атай бастады. Сындық ескертулердің көбісі әділ болды. Мысалы, Algol-60 тілінің есеп беруінде енгізу-шығару механизмінің біреуі де анықталмаған болып шықты! Дегенмен осы ескертулерді қазіргі программалау тілдерінің көбісіне де айтуға болады, өйткені

көптеген программалау аймақтарындағы жаңа стандарттарды Algol тілі орнатқан еді.

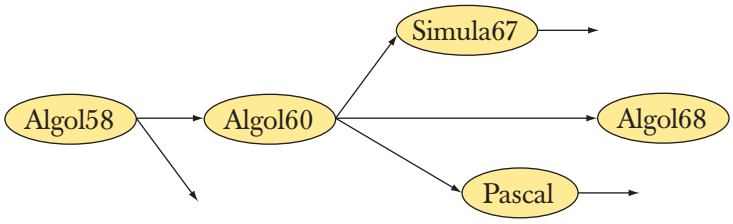
Algol-60 тілімен байланысты басты мәселе, оны қалай жүзеге асыруға болатынын ешкім білмеді. Осы мәселе Питер Наур (Peter Naur) жетекшілік еткен программалаушылар тобының, Algol-60 тілінің есеп беру редакторының және Эдсгер Дейкстраның (Edsger Dijkstra) көмегі арқылы шешілді.



Питер Наур Копенгаген университетінде (University of Copenhagen) астроном білімін алған болатын және Копенгагеннің Техникалық университетінде (Technical University of Copenhagen – DTH) жұмыс істеді, сонымен қоса, компьютер шығаратын Regnecentralen атты дат компаниясында қызмет етті. Ол программалауды 1950-1951 жылдары Данияда компьютерлер болмағандықтан, Кембридж компьютерлік зертханасында (Computer Laboratory in Cambridge) жүріп үйренді, соның нәтижесінде кейіннен академиялық және өндірістік салаларда аса биік лауазымға ие болды. Ол грамматиканы сипаттау үшін қолданылған BNF (Backus-Naur Form – Бэкус-Наур формасы) формасын құрудағы авторлардың бірі болды және де программалар туралы (Бьярне Страуструп алғаш рет, шамамен 1971 жылдары, инварианттарды қолдану туралы Питер Наурдың техникалық мақалаларынан білді) формальды пікірталас жүргізуді жақтаушылардың белсендісі болды. Наур программалаудағы адам факторын әрқашанда ескере отырып, есептеулерге деген терең ойлы көзқарасты ұстанды. Оның кейінгі жұмыстары философиялық сипатта жазылды (әйтсе де ол дәстүрлі академиялық философияны әшейін бос сөз деп санады). Ол даталогиядан Копенгаген университетіндегі бірінші профессор болды ("даталогия" деген дат термині (datalogi)) негізінде "информатика" деп аударылатын; Питер Наур "компьютерлік ғылым" (computer scienses) деген терминді ұнатпайтын, оны дұрыс емес деп санайтын, өйткені есептеулер – бұл компьютерлер туралы ғылымға жатпайды деп айтатын еді.

Эдсгер Дейкстра (Edsger Dijkstra) – компьютерлік ғылымдар аймағындағы атақты ғалым болды. Ол физиканы Лейденде оқып, бірақ өзінің алғашқы жұмыстарын Амстердамдағы Математикалық орталықта (Mathematisch Centrum) орындады. Кейінірек ол, Эйндховен технологиялық университетінде (Eindhoven University of Technology), Burroughs Corporation компаниясында және Остиндегі Техас университетін (University of Texas (Austin)) қоса алғанда, бірнеше жерлерде жұмыс істеді. Algol тілімен жемісті жұмысынан басқа, ол программалауда математикалық логиканы және алгоритмдер теориясын қолданудың пионері және жақтаушысы болды, оған қоса THE операциялық жүйесінің конструкторы және құрастырушыларының бірі, THE – жүйелі түрде параллелизмді қолданатын алғашқы операциялық жүйелердің бірі болған еді. THE атауы Эдсгер Дейкстра сол кезде жұмыс істеген университет – Technische Hogeschool Eindhoven сөздерінің алғашқы әріптерінен алынған болатын. Оның ең танымал мақаласы "Go-To Statement Considered Harmful" деп аталды, ол осы мақаласында құрылымдық емес басқару ағындарын қолданудың қауіптілігін ашып көрсетті.

Algol тілінің генеалогиялық бұтақ тәрізді құрылымы төменде бейнеленген.



Мұндағы Simula67 және Pascal тілдеріне назар аударыңыздар. Олар көптеген (мүмкін, басым бөлігінінің) қазіргі программалау тілдерінің негізін қалаушылар болып табылады.

Сілтемелер

Dijkstra, EdsgerW. "Algol 60 Translation. An Algol 60 Translator for the x1 and Making a Translator for Algol 60". Report MR 35/61. Mathematisch Centrum (Amsterdam), 1961

Dijkstra, Edsger "Go-To Statement Considered Harmful" *Communications of the ACM*, Vol. 11 No. 3, 1968.

Lindsey, C H. "The History of Algol68". Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol 28 No. 3, 1993.

Naur, Peter, ed. *"Revised Report on the Algorithmic Language Algol 60".* A/S Regnecentralen (Copenhagen), 1964

Naur, Peter "Proof of Algorithms by General Snapshots". *BIT*, Vol 6, 1966, pp. 310-316. Вероятно, первая статья о том, как доказать правильность программы.

Naur, Peter. "The European Side of the Last Phase of the Development of ALGOL 60". *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.

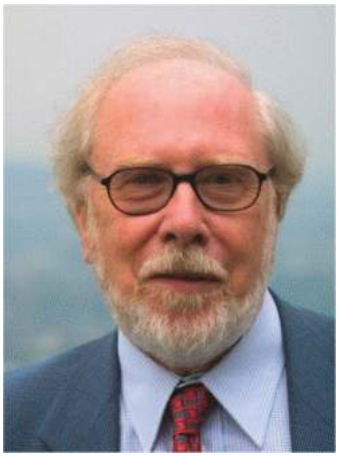
Perlis, Alan J. "The American Side of the Development of Algol". *ACM SIGPLAN Notices*, Vol. 13 No 8, 1978. Special Issue: History of Programming Languages Conference.

van Wijngaarden, A., B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C. H. Lindsey, L. G L. T. Meertens, and R. G. Fisker, eds. *Revised Report on the Algorithmic Language Algol 68* (Sept. 1973). Springer-Verlag, 1976.

22.2.3.2 Pascal программалау тілі

Algol тілінің генеалогиялық бұтағында көрсетілген Algol-68 тілі ірі және зор үміт артылған жобаның бірі болды. Algol-60 тілі сияқты мұны да Algol тілінің (IFIP 2.1 жұмыс тобы) комитеті құрды, бірақ оның жүзеге асуы шексіз мерзімге созылып кетті, көптеген мамандардың шыдамы таусылып, осы жобадан бір пайда шыға ма деп күмәндана бастады. Тіл бойынша комитет мүшелерінің бірі Никлаус Вирт (Niklaus Wirth) Algol тілінің мұрагері ретінде өзінің жеке бір тілін ұсынып, соны жүзеге асыруға шешім қабылдады. Algol-68 тіліне қарсы бағытты ұстанған бұл тіл Algol-60 тілінің қысқартылған нұсқасы түрінде болып, Pascal деп аталды.

Pascal тілін құру 1970 жылы аяқталды, мұның нәтижесінде ол шынымен де қарапайым және әжептеуір икемді тіл болып шықты. Көбінесе ол оқып үйренуге арналған деп жиі айтылғанмен, бірақ алдыңғы жарияланған мақалаларда ол сол кездегі суперкомпьютерлерге арналған Fortran тілінің баламасы ретінде көрсетілді. Pascal тілін оқып үйрену шынымен де қиын емес, жеңіл тасымалданатын нұсқалары шыққаннан кейін, ол программалау тілін оқыту үшін пайдаланылатын жалпыға түсінікті тіл болып қалыптасты, бірақ ол Fortran тіліне бәсекелес бола алмады.



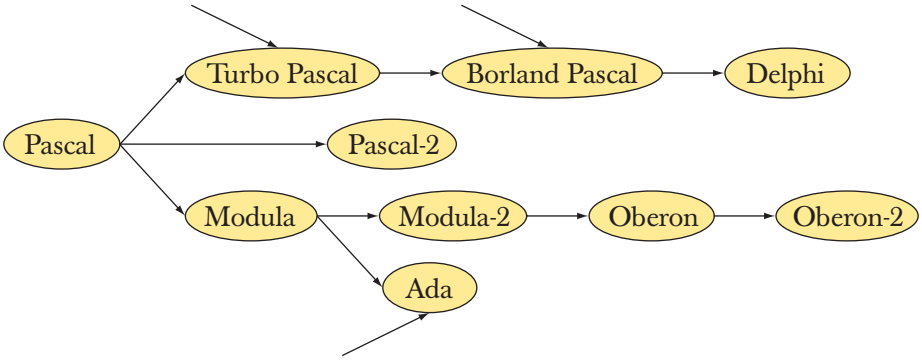
Pascal тілін Цюрихтегі Швейцария Техникалық университетінің (Technical University of Switzerland in Zurich – ETH) профессоры Никлаус Вирт (Niklaus Wirth) жасады. Жоғарыда оның 1969 және 2004 жылы түскен суреттері көрсетілген. Ол өзінің философия докторы дәрежесін Берклидегі Калифорния университетінде (University of California at Berkeley) алды және өзінің ұзақ өмірінде Калифорниямен тығыз байланыста болды. Профессор Вирт программалау тілдерін құратын кәсіби маманның идеалына толық сәйкес келетін жан еді. Жиырма бес жыл ішінде ол келесі программалау тілдерін құрып, жүзеге шығарды.

- Algol W.
- PL/360.
- Euler.
- Pascal.
- Modula.
- Modula-2.
- Oberon.
- Oberon-2.
- Lola (аппараттық жабдықтаманы сипаттау тілі).

Никлаус Вирт өз жұмысын қарапайымдылықты шексіз іздеу деп сипаттады. Оның жұмыстары программалауға көп әсерін тигізді. Осы программалау тілдерін оқып үйрену қызықты жұмыс болып табылады. Профессор Вирт – HOPL (History of Programming Languages) конференциясында екі программалау тілін көрсеткен жалғыз адам.

Соңында Pascal тілі өндірісте кең қолданысқа ие бола алмайтын өте қарапайым және икемсіз тіл болып шықты. Бұл тілді ұмытып кетуден сақтап қалған 1980-

ші жылдардағы Borland компаниясының негізін салушы үш адамның бірі Андерс Хейльсбергтің (Anders Hejlsberg) жұмысы болды. Ол бірінші болып Turbo Pascal тілін жасап, оны жүзеге асырды (басқа да көптеген мүмкіндіктерімен қатар, аргументтерді берудің икемді механизмдерін жүзеге асырды), ал кейінірек оған C++ тілінің моделіне ұқсас объектілі модельді қосты (тек бір ғана мұрагерлікке жол беретін және өте әдемі модульдік механизмі бар). Ол Питер Наур кейде дәріс беретін, Копенгагеннің Техникалық университетінде (Technical University in Copenhagen) білім алды, әлем тығыз байланыста деген осы шығар. Кейінірек, Андерс Хейльсберг Borland компаниясы үшін Delphi тілін және Microsoft компаниясы үшін C# тілдерін жасады. Pascal тілі топтамаларының қысқаша түрдегі генеалогиялық бұтағы төменде көрсетілген.



Сілтемелер

Borland/Turbo Pascal. http://en.wikipedia.org/wiki/Turbo_Pascal.

Hejlsberg, Anders, Scott Wiltamuth, and Peter Golde. *The C# Programming Language, Second Edition.* Microsoft.NET Development Series. ISBN 0321334434.

Wirth, Niklaus. "The Programming Language Pascal". *Acta Informatics*, Vol. 1 Fasc 1, 1971.

Wirth, Niklaus. "Design and Implementation of Modula". *Software-Practice and Experience*, Vol. 7 No. 1, 1977.

Wirth, Niklaus. "Recollections about the Development of Pascal". Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

Wirth, Niklaus. *Modula-2 and Oberon*. Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III). San Diego, CA, 2007. http://portal.acm.org/toc.cfm?id=l238844.

22.2.3.3 Ada программалау тілі

Ada программалау тілі АҚШ-тың қорғаныс министрлігінде туындайтын программалаудың кез келген есебін шығаруға арналған. Ол программалаудың

құрамдас жүйелерін оқуды және жеңіл сүйемелдеуді жабдықтамасыз ететін код жазуға арналған тіл болуы қажет еді. Оның ең жақын келетін тегі болып Pascal және Simula (22.2.6 бөлімін қ.) тілдері саналады. Ada тілін жасаушы топтың жетекшісі Жан Ишбиа (Jean Ichbiah) болды, ол бұрын Simula Users› Group тобының төрағасы қызметін атқарған. Ada тілін жасау кезінде мыналарға:

- мәліметтерді абстракциялауға (1995 жылға дейін мұралаусыз);
- статикалық типтерді қатаң тексеруге;
- параллелизмді тікелей тілдік сүйемелдеуге аса назар аударылды.

Ada жобасының мақсаты программалық жабдықтамалар құру қағидаларын нақты түрде жүзеге асыру болды. Осыған орай Қорғаныс министрлігі тілді емес, тілді жобалау процесін құрастырды. Осы процеске көптеген адамдар мен ұйымдар қатынасты, олар жеңімпаз спецификация идеяларын жүзеге асыратын, ең жақсы спецификация және ең жақсы тіл құру үшін бір-бірімен қатаң бәсекеге түсті. Осы жиырма жылға жалғасқан аса ірі жобаны 1980 жылдан бастап (1975-1998 жж.) AJPO (Ada Joint Program Office) бөлімі басқарды.

1979 жылы бұл тілге ақын лорд Байронның (Byron) қызы леди Аугуста Ада Лавлейстің (Augusta Ada Lovelace) құрметіне Ada тілі атауы берілді. Леди Лавлейсті қазіргі заманның бірінші программалаушысы десе де болады (егер қазіргі заман ұғымын аздап кеңейтсек), өйткені ол 1840-шы жылдарда Кембриджджегі лукасиандық математиканың профессоры (яғни бұрын Ньютон атқарған қызметте) болып істеген Чарльз Бэббиджбен (Charles Babbage) бірге алғашқа механикалық компьютерді құру процесіне араласты. Өкінішке орай, Бэббидж машинасы тәжірибе жүзінде өзін ақтай алмады.

Ойланып ұйымдастырылған құру процесінің арқасында Ada тілі Комитет құрған ең жақсы тіл болып саналады. Бірақ бұл шешімді француз компаниясының осы жобаны жеңген құрастырушылар ұжымының көшбасшысы Жан Ишбиа

қуаттамай, оны батыл түрде теріске шығарды. Алайда, егер ол берілген шарттарымен шектелмеген болса, онда одан да жақсы тіл жасап шығаратын еді деп ойлаймын.

АҚШ-тың қорғаныс министрлігі көп жылдар бойы әскери қосымша программаларда Ada тілін ғана пайдалануды қуаттап отырды, бұл осыдан кейін мынадай нақыл сөздің шығуына себепші болды: "Ada тілі – бұл тек жақсы идея емес, бұл – заң!" Алдымен, Ada тілі қолдануға "тиянақты түрде ұсынылған" еді, бірақ көптеген жобалаушыларға басқа тілдерді (көбінесе C++ тілін) қолдануға тиым салынған кезде, АҚШ Конгресі көптеген әскери программаларда тек ғана Ada тілі қолданылсын деген заң қабылдады. Нарықтық және техникалық мүмкіндіктердің әсер етуіне орай бұл заң ақырында күшін жойды. Осылайша, Бъярне Страуструп бірсыпыра адамдар ішіндегі, жұмыстарын АҚШ Конгресінің шешімімен қолдануға тыйым салынған мамандардың бірі болды.

Басқаша айтқанда, біз Ada тілін өзінің алған беделінен едәуір жақсы тіл деп айта аламыз. Егер де АҚШ-тың қорғаныс министрлігі оны қолдануда икемсіздік танытпай, тілдің негізіне жатқан қағидаларды (қосымшаларды жобалау процессорының стандарттары, программалық жабдықтаманы құру құралдары, құжаттама және т.б.) нақты ұстанған болса, онда табыс әлдеқайда көрнекті болар еді. Қазіргі кезде Ada тілі аэроғарыш программаларында және сол сияқты құрамдас жүйелерді жасауға байланысты аймақтарда маңызды рөл атқарады.

Ada тілі 1980 жылы әскери стандарт болып бекітілді, ANSI стандарты 1983 жылы (бірінші нұсқасы 1983 жылы алғашқы стандарт шығарылғаннан үш жыл өткен соң, пайда болды), ISO стандарты – 1987 жылы қабылданды. ISO стандарты 1995 жылғы шығарылымында біраз (әрине, салыстырмалы түрде) қайта қаралды. Стандартқа бірсыпыра жақсы өзгертулер енгізу параллелизм механизмдерінің икемділігін және мұрагерлікті сүйемелдеуді арттырды.

Сілтемелер

Barnes, John. *Programming in Ada 2005*. Addison-Wesley, 2006. ISBN 0321340787.

Consolidated Ada Reference Manual, consisting of the international standard (ISO/IEC 8652:1995). *Information Technology - Programming Languages - Ada*, as updated by changes from Technical Corrigendum 1 (ISO/IEC 8652:1995:TC1:2000).

Ada тілінің ресми бастапқы парағы: www.usdoj.gov/crt/ada/.

Whitaker, William A. *ADA - The Project: The DoD High Order Language Working Group.* Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

22.2.4 Simula программалау тілі

Simula тілін 1960-шы жылдардың бірінші жартысында Норвегиялық есептеу орталығында (Norwegian Computing Center) және Осло университетінде (Oslo

University) істейтін Кристен Нюгорд (Kristen Nygaard) пен Оле-Йохан Дал (Ole-Johan Dahl) жасап шығарды. Simula тілі күмәнсіз Algol тілі топтамалары құрамына жатады. Негізінде, Simula тілі Algol-60 тілін толық қамтитын, одан жоғарғы топтағы тіл болып саналады. Біздің Simula тіліне ерекше көңіл бөліп отырған себебіміз, өйткені ол қазіргі кездегі объектіге бағытталған программалау деп аталып жүрген көптеген іргелі идеялардың қайнар көзі болып табылады. Ол мұралау мен виртуалды функцияларды жүзеге асырған алғашқы тіл болды. Базалық кластың интерфейсі арқылы шақыруға және ауыстыруға болатын пайдаланушы типіндегі class сөзі және функциядағы *virtual* сөзі C++ тіліне осы Simula тілінен келген.

Simula тілінің қосқан үлесі тек тілдік қасиеттермен шектелмейді. Ол программа кодының нақты құбылысын модельдеу идеясына негізделген объектіге бағытталған жобалаудың төмендегідей айқын өрнектелген ұғымдарынан тұрады.

- Класс және кластар объектілері түріндегі идеяларды бейнелеу.
- Иерархиялық қатынастарды кластар иерархиясы (мұралау) түрінде бейнелеу.

Осылайша, программа біртұтас (монолит) нәрсе емес, өзара әрекеттесетін объектілер жиыны түрінде болады.

Кристен Нюгорд – Simula 67 тілін құрушылардың бірі (Оле-Йохан Далмен бірге, суретте сол жақта көзілдірікпен) – жігерлі және жомарт алып болды (соның ішінде бойымен де). Ол объектіге бағытталған программалау мен жобалаудың іргелі идеяларының негізін қалаушылардың бірі болды, әсіресе мұралауды қолдады және үздіксіз осы ұстанымды ондаған жылдар бойынша қуаттап жүрді. Оны ешқашанда қарапайым, қысқа және алысқа бармайтын жауаптар қанағаттандырмады. Оны, сонымен қатар, әлеуметтік мәселелер де ондаған жылдар бойы толғандырып

жүрді. Ол Норвегияның Еуропалық одаққа кіруіне шынайы қарсы болды, өйткені ол мұнан шектен тыс орталықтандырудың, төрешілдіктің әсерін сезіп, сол одақтың шалғай шетінде орналасқан кішкентай елдің ішкі жағдайы ескерілмейді деп қауіптенді. 1970-ші жылдар ортасында Кристен Нюгорд Даниядағы Аархус университетіндегі (University of Aarhus) компьютерлік ғылымдар факультетінде істеген жұмыстарына көп уақыт бөлді, осы кездерде мұнда Бъярне Страуструп та магистрлік программа бойынша оқып жүрген болатын.

Кристен Нюгорд математика бойынша магистрлік дәрежесін Осло университетінде (University of Oslo) алды. Ол 2002 жылы компьютерлік ғылымдар аймағындағы көрнекті жаңалықтарды атап өтетін есептеу техникасы Ассоциациясының (Association for Computing Machiner – ACM) ең жоғарғы құрмет белгісі – Тьюринг сыйлығын (өзінің өмір бойы әріптес досы болған Оле-Иохан Далмен бірге) алғаннан кейін бір айдан соң қайтыс болды.

Оле-Иохан Дал оған қарағанда, дәстүрлі академиялық ғалым болды. Оны тілдердің спецификациясы мен жасанды әдістер өте қызықтыратын еді. Ол 1968 жылы информатика саласы бойынша Осло университетіндегі (компьютерлік ғылымдар бойынша) алғашқы профессор болды.

2000 жылдың тамыз айында Норвегия королі Дал мен Нюгордты Қасиетті Олаф орденінің командорлары (Commanders of the Order of Saint Olav) деп жариялады. Дегенмен де өз отанының да даналарын танитын кездері болады емес пе!

Сілтемелер

Birtwistle, G., O-J. Dahl, B. Myhrhaug, and K. Nygaard: *SIMULA Begin*. Student-litteratur (Lund. Sweden), 1979. ISBN 9144062125.

Holmevik, J. R. "Compiling SIMULA: A Historical Study of Technological Genesis". *IEEE Annals of the History of Computing*, Vol. 16 No. 4, 1994, pp. 25-37.

Kristen Nygaard's homepage: *http://heim.ifi.uio.no/~kristen/*.
Krogdahl, S. "The Birth of Simula". Proceedings of the HiNC 1 Conference in Trondheim, June 2003 (IFIP WG 9.7, in cooperation with IFIP TC 3).
Nygaard, Kristen, and Ole-Johan Dahl. "The Development of the SIMULA Languages". *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
SIMULA Standard. *DATA processing – Programming languages – SIMULA*. Swedish Standard, Stockholm, Sweden (1987). ISBN 9171622349.

22.2.5 C программалау тілі

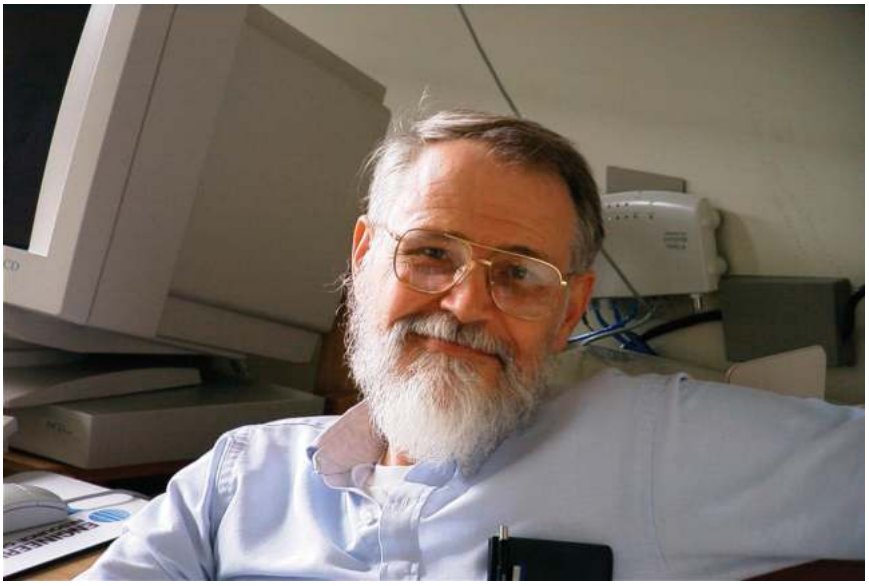
1970-ші жылдары жүйелік программалауды, оның ішінде операциялық жүйені жасау – ассемблерлік кодта орындалуы тиіс екені және оны тасымалдауға болмайтыны белгілі болған еді. Бұл Fortran тілі пайда болу қарсаңындағы ғылыми программалау ортасында орын алған жағдайды еске түсірді. Бірсыпыра мамандар және топтар осындай кезеңдегі туындаған жағдайдың бір шешімін табуға бел байлады. Сол кездерде пайда болған C программалау тілі ұзақ мерзімге есептеліп шешімін тапқан осындай еңбектің жақсы бір нәтижесі (бұл туралы толығырақ 27-тарауда) болып шықты.

Деннис Ритчи (Dennis Ritchie) Нью-Джерси штаты, Мюррей-Хиллдағы (Murray Hill, New Jersey) Bell Telephone Laboratories компаниясының компьютерлік ғылымдар бойынша Зерттеу орталығында C программалау тілін жасап, оның шығуын жүзеге асырды. C тілінің бір ерекшелігі ол аппараттық жабдықтаманың іргелі аспектілерін тікелей есепке алуға мүмкіндік беретін, қарапайым программалау тілі болып шықты. Көптеген күрделіліктер (негізінде C++ тілінен сәйкестікті жабдықтамасыз ету үшін алынған) кейінірек және көбінесе Деннис Ритчидің ойына қарамастан енгізіле бастады. Бір жағынан алғанда, C тілінің жетістігі оның кең қолжетімділігімен түсіндіріледі, бірақ оның нақты қуаттылығы тілдің мүмкіндіктері аппараттық жабдықтаманың қасиеттерін тікелей бейнелеуде өте жақсы көрініс тапты (25.4-25.5 бөлімдерін қ.). Деннис Ритчи C тілін тексеру механизмі әлсіз болғанмен, қатаң типке келтірілген, жинақы тіл ретінде сипаттады; басқаша айтқанда, C тілі статикалық типтер (компиляциялау кезеңінде анықталатын) жүйесінен тұрды, ал оның анықтамасына қарамастан, объектіні қолданатын программа дұрыс емес деп есептелді. Дегенмен, C тілінің компиляторы мұндай жағдайларды анықтай алмайтын. C тілінің компиляторы 48 K көлемнен тұратын компьютер жадында орындалатын болғандықтан, бұл қисынды еді. Көп ұзамай C тілі практика жүзінде қолданылып, мамандар кодтың типтер жүйесіне сәйкестігін компилятордан бөлек тексере алатын *lint* программасын жазып шықты.

Кен Томпсон (Ken Thompson) мен Деннис Ритчи Unix операциялық жүйесінің авторлары атанды, бұл жүйе сол кездерге дейінгі барлық уақыт аралығындағы ең маңызды жүйе болып есептелетін де шығар. C тілі осы уақытқа дейін және қазіргі күндерде де Unix операциялық жүйесімен тығыз байланысты болып саналады, сол арқылы Linux жүйесімен де байланысып, ашық код үшін қозғалыста да көзге түсіп келеді.

Деннис Ритчи Lucent Bell Labs компаниясынан зейнетке шықты. Ол қырық жыл бойы Bell Telephone компаниясының Компьютерлік ғылымдар зерттеу орталығында жұмыс істеді. Ол "физика" мамандығы бойынша Гарвард университетін (Harvard University) бітірген, қолданбалы математика бойынша философия докторы атағын да осы университетте алған.

1974-1979 жылдарда C++ тілін дамытуға және бейімдеуге Bell Labs компаниясының көптеген мамандары әсерін тигізді. Солардың ішінде, жалпыға бірдей ұнамды, әрі сыншы, әңгімелесуге де шебер және де идеялар генераторы Дут Макилрой (Doug McDroy) болатын. Ол тек C және C++ тілдеріне ғана емес, Unix операциялық жүйесіне, сонымен қатар басқа да көптеген жұмыстарға әсерін тигізді.



Брайан Керниган (Brian Kernighan) – программалаушы, әрі ерекше жаза білетін жан еді. Оның программалары мен прозалары – айқындықтың үлгісі. Осы кітаптың стилінде де оның үздік туындысы – *The C Programming Language* оқулығына (оның авторлары – Брайан Керниган мен Деннис Ритчи фамилияларының алғашқы әріптерінен алынған K&R атымен белгілі) аздап еліктеу бар екенін айта кету керек.

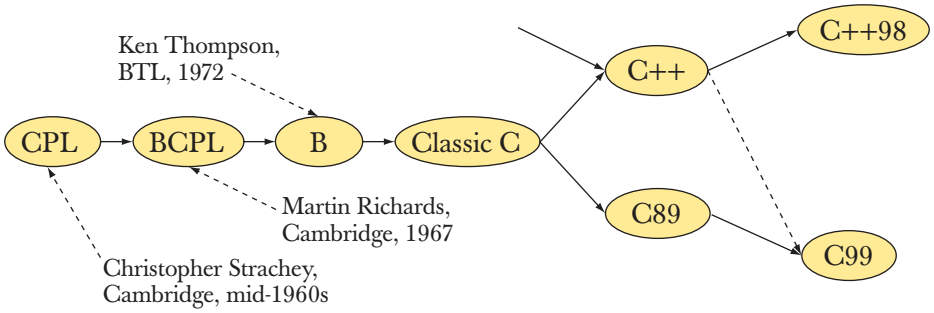
Пайдасы көрнекті болуы үшін жақсы идеялар ұсыну жеткіліксіз, оларды көп адам түсінетіндей етіп, қарапайым түрде көрсетіп, анық тұжырымдау қажет. Көп сөзділік – айқындылықтың қас дұшпаны; оған қоса ұзын сонар баяндау мен шектен тыс абстрактылықтың да болмауы тиіс екенін атап өткен жөн. Пуристер осындай көпшілікке түсінікті түрде жеткізудің нәтижелерін жиі күлкі етіп, сарапшыларға ғана қонымды формада көрсетілген "керемет нәтижелерді" дұрыс деп есептейді. Біз пуристерге жатпаймыз: жаңадан келгендерге жай сөздерден тұрмайтын бағалы идеяларды меңгеру қиын, бірақ бұл олардың кәсіби өсуіне және жалпы қоғам үшін де қажетті болып табылады.

Брайан Керниган көп жылдар бойы көптеген маңызды программалау және баспа жобаларына қатысты. Мысал ретінде, өзінің авторларының инициалдары бойынша қысқаша атқа ие болған (Aho, Weinberger және Kernighan), сценарийлерді дайындайтын алғашқы тілдердің бірі – AWK тілін, сонымен қатар AMPL – (A Mathematical Programming Language – математикалық программалауға арналған тіл) тілін айтуға болады.

Қазіргі кезде Брайан Керниган – Принстон университетінің (Princeton University) профессоры; ол күрделі тақырыптарды түсінікті етіп баяндайтын тамаша оқытушы. Ол отыз жылдан астам уақыт кезеңінде Bell Telephone компаниясының Компьютерлік ғылымдар зерттеу орталығында жұмыс істеді. Кейінірек Bell Labs компаниясы AT&T Bell Labs деп аталып жүрді де, сонан кейін AT&T Labs және Lucent Bell Labs компанияларына бөлініп кетті. Брайан Керниган физика

мамандығы бойынша Торонто университетін (University of Toronto) бітірген; ол электротехника бойынша философия докторы атағын Принстон университетінде алды.

С тілі топтарының генеалогиялық бұтақ тәрізді даму жолы төменде көрсетілген:



С тілінің түп тамыры Англиядағы аяқталмай қалған CPL тілін құрастыру жобасынан бастау алып, Кембридж университетінің (Cambridge University) қызметкері Мартин Ричардстің (Martin Richards) Массачусет технологиялық институтында (MIT) болған кезінде жасаған BCPL (Basic CPL) тіліне, сонымен қатар Кен Томпсон құрған тікелей аударылатын (интерпретатор түрінде) B тіліне жалғасады. Кейінірек C тілі ANSI және ISO институттары арқылы стандартталды және оған C++ тілі күшті әсер етті (мысалы, онда функция аргументтерін тексеру мен `const` түйінді сөзі пайда болды).

CPL тілін жасау Кембридж университеті мен Лондондағы Империя колледжінің (Imperial College) бірлескен жобасының мақсаты болды. Алдымен жобаны Кембриджде орындау жоспарланған болатын, сондықтан "C" әріпі ресми түрде "Cambridge" сөзін білдіреді. Жобаның серіктесі болып Империя колледжі қосылған кезде, "C" әріпінің ресми түрдегі түсіндірмесі "Combined" ("бірлескен") болып өзгерді. Негізінде (бізге айтылғандары бойынша), оны CPL тілінің негізгі құрастырушысы болған Кристофер Стрэчидің (Christopher Strachey) құрметіне Christopher атымен де жиі байланыстырады.

Сілтемелер

Брайан Кернигандың бастапқы веб-парағы (home page): *http://cm.bell-labs.com/ cm/cs/who/bwk.*

Деннис Ритчидің бастапқы веб-парағы: *http://cm.bell-labs.com/cm/cs/who/dmr.*

ISO/IEIC 9899:1999. *Programming Languages – C* (The C standard.)

Kernighan, Brian, and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 1978. Second Edition, 1989. ISBN 0131103628.

Bell Labs компаниясының Компьютерлік ғылымдар зерттеу орталығы қызметкерлерінің тізімі: *http://cm.bell-labs.com/cm/cs/alumnl.html.*

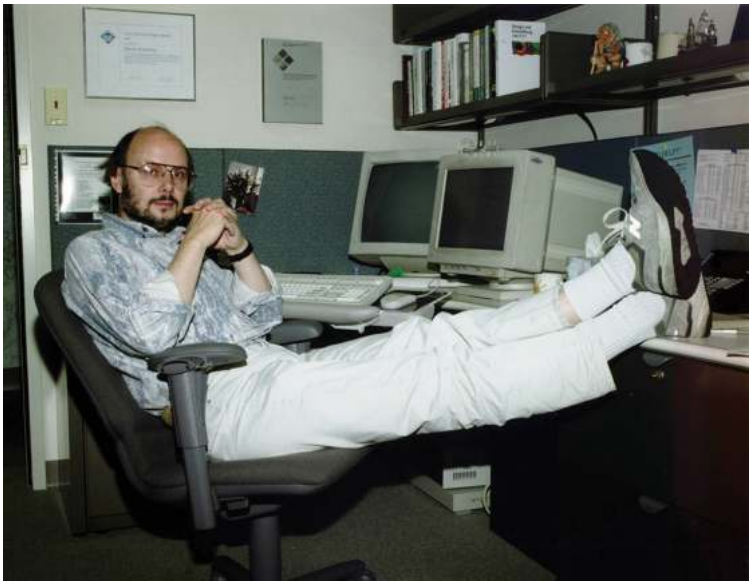
Ritchards, Martin. *BCPL – The Language and Its Compiler*. Cambridge University Press, 1980. ISBN 0521219655.
Ritchie, Dennis. "The Development of the C Programming Language. Proceedings of the ACM History of Programming Languages Conference (HOPL-2).
ACM SIGPLAN Notices, Vol. 28 No. 3,1993.
Salus, Peter. *A Quarter Century of UNIX*. Addison-Wesley, 1994. ISBN 0201547775.

22.2.6 C++ программалау тілі

C++ тілі – жүйелік программалауға бейімделген әмбебап программалау тілі. Оның негізгі қасиеттерін атап өтейік:

- Ол C тілінен жақсы.
- Мәліметтер абстракциясын сүйемелдейді.
- Объектіге бағытталған программалауды сүйемелдейді.
- Жалпыланған программалауды сүйемелдейді.

C++ тілі Нью-Джерси (New Jersey) штатының Мюррей-Хиллдағы (Murray Hill) Bell Telephone Laboratories компаниясының Компьютерлік ғылымдар зерттеу орталығынан Бьярне Страуструппен жасалып, жүзеге асырылды, мұнда сонымен қатар Деннис Ритчи, Брайан Керниган, Кен Томпсон, Дуг Макилрой және тағы да басқа Unix жүйесінің алыптары жұмыс істеді.



Бьярне Страуструп математика және компьютерлік ғылымдар бойынша магистр дәрежесін Даниядағы өзінің туған қаласы Эрхуста (Erhus) алды. Сонан

соң ол Кембриджге (Cambridge) көшіп келді де, сол жерде Дэвид Уилермен (David Wheeler) бірге жұмыс жасай отырып, компьютерлік ғылымдар бойынша философия докторы дәрежесін алды. C++ тілін құру мақсатының мәнісі мынада болды.

- Кең таралған жобалар аясында абстракциялау әдістерін қолжетімді және басқарылатын етіп жасау.
- Тиімділік табыстың негізгі критериі болып саналатын қолданбалы аймақтарға объектіге бағытталған және жалпылама программалауды енгізу.

C++ тілі пайда болғанға дейін осы әдістер (көбінесе негізсіз жалпы "объектіге бағытталған программалау" атауына біріктірілген) іс жүзінде индустрияда белгісіз болды. Ғылыми программалаудағы Fortran тілі пайда болғанға дейінгі кез сияқты, жүйелік программалауда да C тілі пайда болғанға дейін осы технологияларды нақты қосымшаларда қолдану өте қымбат және қарпайым программалаушылар үшін өте күрделі деп саналды.

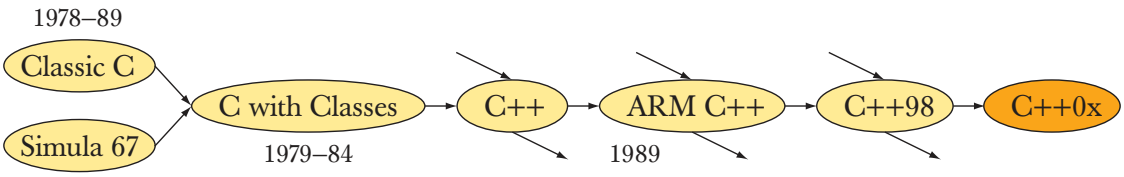
C++ тілін құру бағытындағы жұмыс 1979 жылы басталды, ал 1985 жылы ол коммерциялық түрде пайдалануға шығарылды. Сонан кейін де Бъярне Страуструп, оның Bell Labs компаниясындағы достары және де басқа көптеген ұйымдар C++ тілін жетілдіруді жалғастыра берді де, 1990 жылы оны стандарттау рәсімі басталды. Содан кейін C++ тілін анықтау ісі алдымен ANSI (АҚШ-тың Ұлттық стандарттау институты) орталығында, 1991 жылдан бастап – ISO-да (Стандарттау бойынша халықаралық ұйым) жүргізілді.

Бьярне Страуструп тілдің жаңа қасиеттерін құруға жауапты түйінді топшаның төраға қызметін атқарушы ретінде осы процесте басты рөл атқарды. Алғашқы халықаралық стандартты (C++98) рәсімдеу (ратификациялау) 1998 жылы болды, ал екінші стандартпен (C++0x) жұмыс істеу әлі күнге дейін жалғасуда.

C++ тілінің тарихындағы ең елеулі оқиға, ол пайда болғаннан кейін он жылдан соң шыққан контейнерлер мен алгоритмдердің стандартты STL кітапханасы болды. Ол, негізінде, өзінің әдемілігімен және математикаға жақындығымен ерекшеленіп, барынша әмбебап және тиімді программалық жабдықтама құруға бағытталып, Александр Степановтың (Alexander Stepanov) жетекшілік етуімен орындалған көп жылғы еңбектің нәтижесі болды.

Алекс Степанов – STL кітапханасын жасаушы және жалпылама программалаудың пионері. Ол Мәскеу мемлекеттік университетін бітіріп, әртүрлі программалау тілдерін (Ada, Scheme және C++ қоса алғанда) қолдана отырып, робототехника және алгоритмдер аймағында жұмыс істеген. 1979 жылдан бастап ол АҚШ-тың академиялық ұйымдарында және де GE Labs, AT&T Bell Labs, Hewlett-Packard, Silicon Graphics және Adobe сияқты өнеркәсіптік компанияларда жұмыс жасады.

C++ тілінің генеалогиялық бұтақшасы төменде көрсетілген.



C with Classes тілін Бьярне Страуструп C және Simula тілдерінің идеяларын синтездеу нәтижесі ретінде құрды. Бұл тіл оның мұрагері, яғни C++ тілі жүзеге асырылғаннан кейін қолданыстан алынып тасталынды.

Программалау тілдерін талқылау ісіне олардың әдемілігі мен жаңа қасиеттеріне байланысты жиі назар аударылады. Әйтсе де, C және C++ тілдері компьютерлік технологияның бүкіл тарихындағы ең сәтті шыққан программалау тілдері болды: олардың күші икемділігінде, өнімділігінде және тұрақтылығында болды. Программалық жабдықтама жүйелерінің көпшілігі ондаған жылдар бойы жұмыс істейді де, көбінесе өзінің аппараттық ресурстары мүмкіндігін жеткен жеріне дейін пайдаланып, қандай да бір күтпеген өзгерістерге ұшырай бастайды.

C және C++ тілдері осы ортада өте табысты болды. Біз Деннис Ритчидің мына нақыл сөзін жақсы көреміз: "Адамдар кейбір тілдерді өз ойларының дұрыстығын дәлелдеу үшін жасады, ал басқаларын есептер шығару үшін жасады". C тілі екінші санаттағы тілдерге жатады. Бьярне Страуструп былай деп айтқанды жақсы көреді: "C++ тілінен әдемі тілді қалай жасау керек екенін, тіпті мен де білемін". C++ тілінің мақсаты C тілінің мақсаты сияқты – абстрактілі түрдегі әдемілік (біз де оны өте бағалаймыз) емес, оның пайдалылығы.

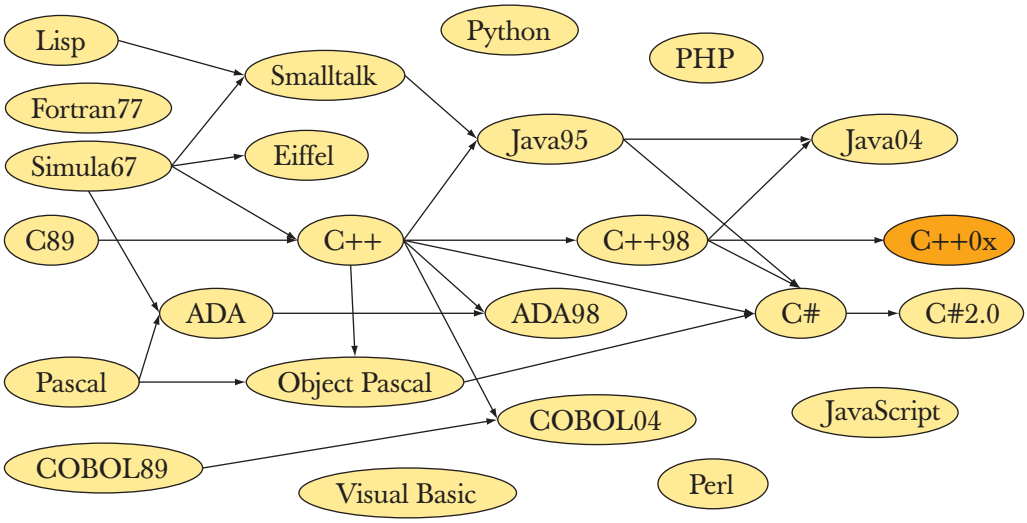
Мен осы кітапта C++0x тілі нұсқасының мүмкіндіктерін пайдалана алмағаныма жиі өкінетінмін. Ол көптеген мысалдар мен түсіндірулерді қарапайым ететін еді. C++0x нұсқасындағы стандартты кітапхана компоненттерінің мысалы болып **`unordered_map`** (21.6.4 бөлімін қ.), **`array`** (20.9 бөлімін қ.) және **`regexp`** (23.5-23.9 бөлімін қ.) кластары саналады. C++0x нұсқасында шаблондарды мұқият тексеру, өте қарапайым және әмбебап инициалдау, сонымен қатар белгілеулердің түсінікті жүйесі болады (Б.Страуструптың HOPL-III конференциясындағы мақаласын қ.).

Сілтемелер

Александр Степановтың жарияланымдары: *www.stepanovpapers.com.*
Бъярне Страуструптың алғашқы веб-парағы: *www.research.att.com/~bs.*
ISO/IEC 14882:2003. *Programming Languages - C++*. (C++ тілінің стандарты).
Stroustrup, Bjarne. "A History of C++: 1979-1991. Proceedings of the ACM
History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3,1993.
Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
Stroustrup, Bjarne. *The C++ Programming Language (Special Edition)*. Addison-Wesley, 2000. ISBN 0201700735.
Stroustrup, Bjarne. "C and C++: Siblings"; "C and C++: A Case for Compatibility"; and "C and C++: Case Studies in Compatibility". *The C/C++ Users Journal*. July, Aug., and Sept. 2002.
Stroustrup, Bjarne. "Evolving a Language in and for the Real World: C++ 1991-2006". Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III). San Diego, CA, 2007. *http://portal.acm.org/toc.cfm?id=1238844.*

22.2.7. Жұмыстың қазіргі жай-күйі

Қазіргі кезде программалау тілдері қалай қолданылуда және олар не үшін қажет? Бұл сұраққа шынымен де жауап беру қиын. Қазіргі тілдердің генеалогиялық бұтақшасы, тіпті қысқартылған түрінің өзінде, бір-бірімен қым-қиғаш байланысып, шатасып жатқан күйде бейнеленеді.



Негізінде, веб-парақтардан (немесе басқа жерлерден) табылған нақтылы статикалық мәліметтердің көбісі әдеттегі алып қашпа сөздерден алыс кетпеген, өйткені олар пайдалану қарқындылығымен әлсіз байланысқан құбылыстарды бағалауға тырысады, мысалы, желі ішіндегі веб-парақтардағы кез келген бір программалау тілдерінің айтылу жиілігі, компиляторларды сату, академиялық мақалалар, кітаптардың сатылуы және т.б. Бұл көрсеткіштер, бұрынғыларға қарағанда, жаңа программмалау тілдерінің кең таралғандығын көтеріңкі түрде бейнелейді. Қандай болғанымен де, программалаушы деген кім? Программалау тілін күн сайын қолданатын адам ба? Мүмкін тілді үйрену үшін шағын программаларды жазатын студент пе? Мүмкін тек программалау туралы пікірін айтатын профессор болар? Мүмкін әр жыл сайын программалар құратын физика маманы ма? Бірнеше программалау тілдерін әр апта сайын бірнеше немесе тек бір рет қана пайдаланатын адам кәсіби программалаушы болып табыла ма? Әртүрлі статикалық көрсеткіштер әртүрлі жауаптарға алып келеді.

Дегенмен де біз осы сұраққа жауап беруге тиіспіз, өйткені 2008 жылы әлемде он миллионға жуық кәсіби программалаушылар болған. Оған куә ретінде IDC компаниясының C89 C++ есеп беруін (мәліметтерді жинақтау ісімен айналысып жүрген), компиляторларды сатушылар мен баспагерлермен өткізілген пікірсайыс нәтижелерін, сонымен қатар веб-парақтардағы әртүрлі дерек көздерін келтіруге болады. Бізбен сөз таластыруыңызға да болады, дегенмен бізге нақты белгілісі, бірден жүз миллионға дейінгі адамдардың жартылай саны программалаушы сөзінің анықтамасына сәйкес келеді. Олар қандай тілдерді пайдаланады? Мүмкін (жай ғана мүмкін) олардың 90%-дан астам программалары Ada, C, C++, C#, COBOL, Fortran, Java, PERL, PHP және Visual Basic тілдерінде жазылған шығар.

Жоғарыда аты аталған тілдерден басқа біз ондаған, тіпті жүздеген атауларды да көрсетуімізге болады. Бірақ біз тек қызықты немесе маңызды тілдерді ғана атап өту қажет деп санаймыз. Егер сізге қосымша ақпарат қажет болса, онда оларды өзіңіз іздеп тауып алуыңызға болады. Кәсіпқой мамандар бірнеше тілді біледі және қажет болып жатса, жаңа тілді де үйреніп алады. Барлық адамдар үшін және барлық қосымшалар үшін жалғыз бір тілдің дұрыс болып табылуы мүмкін емес. Шынында да, бізге белгілі негізгі жүйелердің барлығы бірнеше тілдерді пайдаланады.

22.2.8 Ақпараттардың дерек көздері

Әрбір тілді сипаттау өзінің жеке сілтемелер тізімінен тұрады. Төменде бірнеше тілдер үшін сілтемелер келтірілген:

Программалау тілдерін құрушылардың парақтары мен суреттері:
www.angelfire.com/tx4/cus/people/.

Программалау тілдерінің бірнеше мысалдары
http://dmoz.org/Computers/Programming/Languages/.

Оқулықтар

Scott, Michael L. *Programming Language Pragmatics*. Morgan Kaufmann, 2000. ISBN 1558604421.

Sebesta, Robert W. *Concepts of Programming Languages*. Addison-Wesley, 2003. ISBN 0321193628.

Программалау тілдерінің тарихы туралы кітаптар

Bergin, T.J., and R.G. Gibson, eds. *History of Programming Languages – II.* Addison-Wesley, 1996. ISBN 0202295021.

Hailpern, Brent, and Barbara G. Ryder, eds. Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III).

San Diego, CA, 2007. http://portal.acm.org/toc.cfm?id=1238844.

Lohr, Steve. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts - The Programmers Who Created the Software Revolution*. Basic Books, 2002. ISBN 9780465042265.

Sammet, Jean. *Programming Languages: History and Fundamentals*. Prentice-Hall, 1969. ISBN 0137299885.

Wexelblat, Richard L., ed. *History of Programming Languages*. Academic Press, 1981. ISBN 0127450408.

БАҚЫЛАУ СҰРАҚТАРЫ

1. Тарих не үшін қажет?
2. Программалау тілдері не үшін қажет? Мысалдар келтіріңіз.
3. Жақсы программалау тілдерінің кейбір іргелі қағидаларын атап көрсетіңіз.
4. Абстракция деген не? Абстракцияның жоғарғы деңгейі деген не?
5. Программалаудың жоғарғы деңгейдегі идеалдарын атап өтіңіз.
6. Жоғарғы деңгейдегі программалаудың әлеуетті артықшылықтарын көрсетіңіз.
7. Кодты қайталап пайдалану деген не және оның пайдасы неде?
8. Процедуралық программалау деген не? Нақты мысал келтіріңіз.
9. Мәліметтер абстракциясы деген не? Нақты мысал келтіріңіз.
10. Объектіге бағытталған программалау деген не? Нақты мысал келтіріңіз.
11. Жалпыланған программалау деген не? Нақты мысал келтіріңіз.
12. Мультипарадигмалық программалау деген не? Нақты мысал келтіріңіз.
13. Мәліметтерді компьютер жадында сақтауға мүмкіндік беретін алғашқы программа қашан орындалған?
14. Дэвид Уилер қандай көрнекті жұмысты орындады?

15. Алғашқы программалау тілін құруға Джон Бэкустің қосқан негізгі үлесі туралы айтып беріңіз.
16. Грейс Мюррей Хоппер алғашқы болып қай тілді жасап шығарды?
17. Джон Маккарти өзінің ең басты жұмысын компьютерлік ғылымдардың қай аймағында орындады?
18. Питер Наур Algol-60 тілін құруға қандай үлес қосты?
19. Эдсгер Дейкстра қандай көрнекті жұмысты орындады?
20. Никлаус Вирт қандай тілді жобалап, жүзеге асырды?
21. Андерс Хейльсберг қандай тілді жасап шығарды?
22. Жан Ишбианың Ada жобасындағы рөлі қандай?
23. Simula тілі алғашқы болып программалаудың қандай стилін ашты?
24. Кристен Нюгорд (Ослодан басқа) қай жерде оқытушы болып істеді?
25. Оле-Иохан Дал қандай көрнекті жұмысты орындады?
26. Кен Томпсонның жетекшілігімен қандай операциялық жүйе жасалды?
27. Дуг Макилрой қандай көрнекті жұмысты орындады?
28. Брайан Керниганның ең атақты кітабын атаңыз.
29. Деннис Ритчи қай жерде жұмыс істеді?
30. Бьярне Страуструп қандай көрнекті жұмысты орындады?
31. Алекс Степанов STL кітапханасын жобалау үшін қандай тілдерді пайдалануға тырысты?
32. 22.2 бөлімінде сипатталмаған он программалау тілін атап көрсетіңіз.
33. Scheme тілі қандай программалау тілінің диалектісі болып табылады?
34. C++ тілінің танымал екі мұрагерін атаңыз.
35. Неліктен C тілі C++ тілінің бөлігі болып қалды?
36. Fortran қысқартылған сөз (аббревиатура) болып табыла ма? Егер солай болса, онда қандай сөздер қолданылған?
37. COBOL сөзі аббревиатура болып табыла ма? Егер солай болса, онда қандай сөздер қолданылған?
38. Lisp сөзі аббревиатура болып табыла ма? Егер солай болса, онда қандай сөздер қолданылған?
39. Pascal сөзі аббревиатура болып табыла ма? Егер солай болса, онда қандай сөздер қолданылған?
40. Ada сөзі аббревиатура болып табыла ма? Егер солай болса, онда қандай сөздер қолданылған?
41. Ең жақсы программалау тілін атаңыз.

ТЕРМИНДЕР

Осы тарауда "Терминдер" бөлімі тілдердің атауынан, адамдардың атынан және ұйымдардың атауынан тұрады.

- Тілдер
  - Ada
  - Algol
  - BCPL
  - C
  - C++
  - COBOL
  - Fortran
  - Lisp
  - Pascal
  - Scheme
  - Simula
- Адамдар
  - Чарльз Бэббидж
  - Джон Бэкус
  - Оле-Иохан Дал
  - Эдсгер Дейкстра
  - Андерс Хейльсберг
  - Грейс Мюррей Хоппер
  - Жан Ишбиа
  - Брайан Керниган
  - Джон Маккарти
  - Дуг Макилрой
  - Питер Наур
  - Кристен Нюгорд
  - Деннис Ритчи
  - Алекс Степанов
  - Бъярне Страуструп
  - Кен Томпсон
  - Дэвид Уилер
  - Никлаус Вирт
- Ұйымдар
  - Bell Laboratories
  - Borland
  - Cambridge University (England)
  - ETH (Швейцариялық федералды техникалық университет)
  - IBM
  - MIT
  - Norwegian Computer Center
  - Princeton University
  - Stanford University
  - Technical University of Copenhagen
  - U.S. Department of Defense
  - U.S. Navy

ЖАТТЫҒУЛАР

1. *Программалау* түсінігіне анықтама беріңіз.
2. *Программалау тілі* түсінігіне анықтама беріңіз.
3. Кітапты парақтап, тарауларға берілген эпиграфтарды оқып шығыңыз. Олардың қайсыларын компьютерлік ғылымның мамандары айтқан? Солардың айтқандарын жинақтайтын бір абзац жазып шығыңыз.
4. Кітапты парақтап, тарауларға берілген эпиграфтарды оқып шығыңыз. Олардың қайсыларын компьютерлік ғылымға қатысы жоқ мамандар айтқан? Солардың туған елдерін және әрқайсысының жұмыс істеу аймақтарын атап көрсетіңіз.
5. Осы тарауда қарастырылған тілдердің барлығында да "Hello, World!" программасын жазып шығыңыз.
6. Осы тарауда қарастырылған тілдердің әрқайсысы туралы жазылған танымал оқулықты және сол тілде жазылған алғашқы аяқталған программаны табыңыз. Осы программаны тарауда көрсетілген барлық қалған тілдерде жазып шығыңыз. Ескерту: сізге шамамен жүз программа жазу керек болады.
7. Әрине, біз көптеген маңызды тілдерді қарастырмай кеттік. Біз C++ тілінен кейін пайда болған барлық программалау тілдерін сипаттаудан бас тартуға мәжбүр болдық. Қазіргі кездегі назар аударуға тұрарлық бес тілді таңдап алыңыз да, солардың үшеуі туралы бір жарым бет мәлімет жазыңыз.
8. C++ тілі не үшін қажет? 10-20-беттен тұратын шығарма жазыңыз.
9. C тілі не үшін қажет? 10-20-беттен тұратын шығарма жазыңыз.
10. Бір программалау тілін (C және C++ емес) таңдап, оның тарихы, мақсаты және мүмкіндіктері туралы 10-20-беттен тұратын шығарма жазыңыз. Көп нақты мысалдар келтіріңіз. Осы тілдерді кімдер және неліктен қолданады?
11. Қазіргі кезде Кембридждегі Лукасиан кафедрасын кім басқарады (Lucasian Chair in Cambridge)?
12. Осы тараудағы программалау тілдерін жасаушылардың қайсыларының математика ғылымы бойынша дәрежесі бар, ал кімдерде жоқ?
13. Осы тараудағы программалау тілдерін жасаушылардың қайсыларында философия докторының дәрежесі бар, ал кімдерде жоқ? Қай аймақта?
14. Осы тараудағы программалау тілдерін жасаушылардың қайсылары Тьюринг сыйлығының иегері болып табылады? Қандай жетістіктері үшін? Тарауда көрсетілген Тьюринг сыйлығы иегерлеріне осы атақты беру туралы ресми хабарландыруларды табыңыз.
15. Файлдан мәліметтер жұбын (аты, жылы) оқып, мысалы, (Algol,1960) және (C, 1974) түрінде, соларға сәйкес график салатын программа жазып шығыңыз.

16. Алдыңғы жаттығудағы программаны толықтырып, сол файлдан кортеждерді (аты, жылы (негізі болған тіл)), мысалы, (Fortran, 1956, ()), (Algol, 1960, (Fortran)) және (C++, 1985, (C, Simula)) сияқты түрде оқитын және негізі болған тілден оның ізбасарына тілсызықпен бағытталған граф салатындай етіп түрлендіріңіз. Осы программаны пайдалана отырып, 22.2.2 және 22.2.7 бөлімдердеріндегі диаграммалардың жақсартылған нұсқаларын салып шығыңыз.

СОҢҒЫ СӨЗ

Әрине, біз тек программалау тілдерінің тарихы мен программалық жабдықтаманың идеалдарын атүсті қарастырып өттік. Осы сұрақтарды өте маңызды деп санағанымызбен, біз үлкен өкінішке орай, оларды бұл кітапта тереңдетіп баяндай алмадық. Программалау тілдерін жобалау және жүзеге асыру кезіндегі ең жақсы программалық жабдықтаманы және программалау әдістерін іздеудің шексіздігіне қатысты өз сезімдеріміз бен идеяларымызды жеткізе білдік деп ойлаймыз. Басқаша айтқанда, ең бастысы программалау, яғни сапалы жабдықтама құру, ал программалау тілі – оны жүзеге асырудың жай ғана құралы екені есіңізде болсын.

Мәтіндерді өңдеу

"Анық түсінікті болуы үшін, ешнәрсе
нақты түсінікті болмайды...
"Түсінікті болу" деген сөзді қолдану
логикалық аргументтің жоқтығын білдіреді".

- Эррол Моррис (Errol Morris)

Бұл тарауда негізінен, мәтіннен ақпарат алу жайлы сөз болады. Біз өз білімдерімізді кітаптар, электрондық пошта немесе кесте түрінде бейнеленіп, кейіннен есептеуге оңай болатындай формада басылып шығарылатын құжаттарға тіркелген сөздер түрінде сақтаймыз. Мұнда біз мәтіндерді өңдеуде басқаларынан көбірек қолданылатын **`string`**, **`iostream`** және **`map`** сияқты стандартты кітапхана мүмкіндіктерін қарастырамыз. Сонан соң мәтіннің шаблондық фрагменттерін өрнектеуге мүмкіндік беретін регулярлық өрнектерді (**`regex`** класын) енгіземіз. Соңында регулярлық өрнектер көмегімен мәтіннен пошта индексі және де мәтіндік файлдардың тексерілген (верификацияланған) форматтары сияқты мәліметтердің спецификалық элементтерін тауып, оларды шығарып алу жолдарын көрсетеміз.

23.1 Мәтін

Негізінде, біз әрқашанда мәтінмен жұмыс істейміз. Біздің кітаптарымыз мәтіндерден тұрады, біздің компьютер экранынан көретініміз де – бұл мәтіндер және программаларымыздың бастапқы кодтары, олар да мәтін болып табылады. Біздің байланыс арналарымыз (барлық түрлері) да сөздермен толтырылған. Екі адамның мәлімет алмасуы кезіндегі барлық ақпаратты да мәтін түрінде бейнелеуге болады, бірақ өте тереңге бармай-ақ қояйық. Бейнелер мен дыбыстарды әдетте сол дыбыстар мен бейнелер күйінде көрсеткен жақсы, бірақ қалғандарының бәрін программаларды талдау мен мәтіндердерді түрлендіру арқылы өңдеуге болады.

3-тараудан бастап, біз `iostream` және `string` кластарын пайдаландық, сондықтан мұнда солар жататын кітапханаларды қысқаша сипаттап шығамыз. Мәтіндерді өңдеу үшін әсіресе ассоциативті жиымдар (23.4 бөлім) пайдалы болып табылады, сондықтан біз оны электрондық поштаны талдау үшін пайдаланудың бір мысалын келтіреміз. Бұл тарауда, шолудан басқа регулярлық өрнектер (23.3-23.10 бөлімдер) арқылы мәтіндегі шаблондық фрагменттерді іздеу мәселелері қарастырылады.

23.2 Сөз тіркестері

`string` класында символдар тізбегі мен символды тіркеске қосатын тіркес ұзындығын анықтайтын және екі тіркестің конкатенациясы сияқты бірнеше пайдалы операциялар бар. Негізінде, стандартты `string` класында аздаған ғана операциялар бар, олардың да басым бөлігі күрделі мәтіндерді төменгі дәрежелі өңдеу кезінде пайдалы болып табылады. Мұнда біз өте пайдалы болып саналатын бірнеше операциялар жайлы ғана сөз етеміз. Қажет болса, олардың толық сипаттамасын (`string` класындағы операциялардың толық тізімін) анықтамалықтардан немесе күрделірек оқулықтардан табуға болады. Мұндай операциялар `string` (`string.h` емес) тақырыбында анықталған.

Тіркестермен орындалатын кейбір операциялар	
`s1 = s2`	`s2` тіркесін `s1` тіркесіне меншіктеу; `s2` тіркесі `string` класының объектісі немесе С тілі стиліндегі тіркес бола алады
`s += x`	тіркес соңына `x` объектісін қосу; `x` объектісі символ, `string` класының объектісі немесе С тілі стиліндегі тіркес бола алады
`s[i]`	индекстеу
`s1+s2`	конкатенация; символдар `string` класының мақсаттық объектісінде `s1` тіркесінің символдары көшірмесі болады, оларға жалғасып `s2` тіркесінің символдары көшірмесі орналасады
`s1==s2`	`string` класының объектілерін салыстыру; мұнда не `s1`, не `s2`, бірақ екеуі де қатар емес, С тілі стиліндегі тіркес бола алады. Және де `!=` операциясының сипаттамасын қ.
`s1<s2`	`string` класының объектілерін лексикографикалық түрде салыстыру; мұнда не `s1`, не `s2`, бірақ екеуі де қатар емес, С тілі стиліндегі тіркес бола алады. Қосымша `<=`, `>` және `>=` операцияларының сипаттамасын қ.
`s.size()`	`s` тіркесіндегі символдар саны
`s.length()`	`s` тіркесіндегі символдар саны
`s.c_str()`	С тілі стиліндегі `s` объектісінің нұсқасы
`s.begin()`	бірінші символға итератор
`s.end()`	`s` тіркесінің соңына жалғасқан ұяға итератор
`s.insert(pos,x)`	`s(pos)` тіркесі алдына `x` объектісін енгізу; `x` объектісі символ, `string` класының объектісі немесе С тілі стиліндегі тіркес бола алады
`s.append(pos,x)`	`s(pos)` тіркесінен кейін `x` объектісін енгізу; `x` объектісі символ, `string` класының объектісі немесе С тілі стиліндегі тіркес бола алады. `x` объектісіндегі символдарды сыйғызу үшін `s` тіркесі үлкейеді
`s.erase(pos)`	`s(pos)` позициясындағы символды өшіру; `s` тіркесінің көлемі бірге кішірейеді
`pos=s.find(x)`	`s` тіркесінен `x` объектісін іздеу; `x` объектісі символ, `string` класының объектісі немесе С тілі стиліндегі тіркес бола алады. `pos` айнымалысы – алғашқы табылған символдың индексі немесе `npos` мәні (`s` тіркесінің соңынан кейінгі ұя позициясы).
`in>>s`	`in` ағымынан `s` объектісіне бос орындармен бөлінген сөзді оқу
`getline(in,s)`	`in` ағымынан `s` объектісіне мәтін жолын оқу
`out<<s`	`s` объектісінен `out` ағымына мәліметтерді жазу

Енгізу-шығару операциялары 10-11-тарауларда және де 23.3 бөлімінде сипатталған. `string` класының объектісіне енгізу операциясы, қажет болғанда, оның

көлемін ұлғайтады да, оның толып кетуі ешқашанда болмайтынына назар аударыңыз.

`Insert()` және `append()` операциялары жаңа символдарға орын беру үшін символдарды жылжытады. `erase()` операциясы символ өшірілген соң, оның орнында қалған бос орынды толтыру үшін символдарды солға жылжытады.

Негізінде, кітапханада стандартты тіркес символдар жиынының көпшілігін сүйемелдейтін, мысалы, мыңдаған символдар (қарапайым символдардан басқа £, Ω, □, δ, ♬ таңбалары) да қарастырылған `Unicode` сияқты, `basic_string` шаблондық класымен сипатталады. Айталық, сізде `Unicode` жиынындағы, мысалы, `Unicode` символы бар қаріп (шрифт) болса, онда сіз келесі код фрагментін жаза аласыз:

```
basic_string<Unicode> a_unicode_string;
```

Біз қолданып жүрген стандартты `string` класы кәдімгі `char` типімен нақтыланған жай ғана `basic_string` класы болып табылады.

```
typedef basic_string <char> string;
                        // тіркес – бұл basic_string<char>
```

Біз `Unicode` кодының символдары мен сөз тіркестерін сипаттамаймыз, бірақ қажет болса, сіз олармен кәдімгі символдармен және тіркестермен жұмыс істегендегі сияқты істей бересіз (оларға да тілдің `string` класы, `iostream` класы ағымдары және регулярлық өрнектер сияқты конструкциялар қолданылады). Егер сізге `Unicode` кодының символдары керек болып жатса, онда тәжірибелі қолданушылардан кеңес сұрау қажет; сіздің программаңыз пайдалы болуы үшін сіз тек қана тіл ережелерін орындап қана қоймай, кейбір жүйелік талаптарды да орындауыңыз керек.

Мәтінді өңдеу мәселесі жайында айтар болсақ, жалпы барлығын да сөз тіркестері мен символдар түрінде бейнелеуге болатынын есте сақтаған жөн. Мысалы, осы бетте `12.333` саны бос орынмен қоршалған алты символдан тұратын тіркес түрінде көрсетілген.

Егер сіз оны сан деп есептейтін болсаңыз, онда алдымен бұл символдарды жылжымалы нүктелі санға айналдырып алуыңыз керек, тек сонан кейін ғана оған арифметикалық операциялар қолдануға болады. Бұл сандарды `string` класы объектілеріне және `string` класы объектілерін сандарға түрлендіруді қажет етеді. 11.4 бөлімде біз бүтін санды `stringstream` класын пайдалану арқылы `string` класы объектісіне айналдыруды көрдік. Осы тәсілді `<<` операторы бар кез келген типке қолдануға болады.

```
template<class T> string to_string(const T& t)
{
    ostringstream os;
    os << t;
    return os.str();
}
```

Мысал қарастырайық.

```
string s1 = to_string(12.333);
string s2 = to_string(1+5*6-99/7);
```

`s1` тіркесінің мәні "`12.333`", ал `s2` тіркесі мәні – "`17`". Іс жүзінде, `to_string()` функциясын тек қана сандық мәндерге емес, `<<` операторы бар кез келген `T` класына қолдануға болады.

`string` класынан санға айналдыратын кері түрлендіру де әрі жеңіл, әрі пайдалы түрде орындалады:

```
struct bad_from_string : std::bad_cast
// тіркесті түрлендіру кезінде қателер жайлы
// хабарлама беретін класс
{
    const char* what() const   // override bad_cast's what()
    {
       return "bad cast from string";
    }
};

template<class T> T from_string(const string& s)
{
    istringstream is(s);
    T t;
    if (!(is >> t)) throw bad_from_string();
    return t;
}
```

Мысал қарастырайық:

```
double d = from_string<double>("12.333");

void do_something(const string& s)
try
{
   int i = from_string<int>(s);
   // . . .
}

catch (bad_from_string e) {
   error ("енгізудің қате тіркесі",s);
}
```

`to_string()` функциясымен салыстырғанда, `from_string()` функциясының қосымша күрделілігі `string` класының көптеген типтердің мәндерін бейнелей алатындығымен түсіндіріледі. Мұнда біз әрбір мысал сайын `string` класының объектісінен қандай тип алғымыз келетінін көрсетуіміз керек. Бұған қоса айтарымыз, бұл қарастырылып отырған `string` класы біз күткендей тип мәнін сақтамауы мүмкін екендігін де білдіреді. Мысал қарастырайық:

```
int d = from_string<int>("Mary had a little lamb"); //ой!
```

Сонымен, біз `bad_from_string` типіндегі аластама түрінде көрсеткен мысалымызда қате болуы мүмкін. 23.9 бөлімде біздің `from_string()` функциясының (немесе соған парапар функцияның) мәтіндік өрістерінен сандық мәндер алуымыз керек болғандықтан, оның қажетті мәтіндік қосымша программаларда маңызды рөл атқаратынын көрсетеміз. 16.4.3 бөлімде `get_int()` эквивалентті функциясының графикалық қолданушы интерфейсінде пайдаланылғаны көрсетілген болатын.

`to_string()` және `from_string()` функцияларының бір-біріне өте ұқсас екеніне назар аударыңыз. Іс жүзінде, олар бір-біріне кері болып табылады; басқаша айтқанда (бос орын, дөңгелектеу, т.с.с. байланысты нақтылықтарды есепке алмасақ), "әрбір ақылға қонымды `T` типі үшін" былай жазуға болады:

```
s==to_string(from_string<T>(s)) // барлық s үшін
```

және

```
t==from_string<T>(to_string(t)) // барлық t үшін
```

мұнда "ақылға қонымды" дегеніміз `T` типінің келісім бойынша конструкторы, `>>` операторы және соған сәйкес `<<` операторы да болуы керек дегенді білдіреді.

`to_string()` және `from_string()` функцияларының жүзеге асырылған нұсқалары өзінің барлық жұмыстарын атқаруы үшін `stringstream` класын пайдаланатынын атап айтқан жөн. Бұл әрекет әмбебап түрлендіру амалын үйлестірілген `<<` және `>>` операцияларын кез келген екі тип үшін анықтау кезінде пайдаланылған болатын.

```
struct bad_lexical_cast : std::bad_cast
{
   const char* what() const { return "bad cast"; }
};

template<typename Target, typename Source>
Target lexical_cast(Source arg)
{
```

```
    std::stringstream interpreter;
    Target result;

    if (!(interpreter << arg)       // arg-ты ағымға оқу
      || !(interpreter >> result) // result-ты ағымнан оқу
      || !(interpreter >> std::ws).eof()) // ағым бос па?
            throw bad_lexical_cast();
    return result;
}
```

`!(interpreter>>std::ws)` нұсқауының `stringstream` ағымында нәтиже алғаннан кейін қалып кететін кез келген бос орын таңбасын оқитыны қызықты нәрсе. Бос орындар болуы заңды, бірақ енгізу ағымында олардан басқа ешқандай да символ қалмауы мүмкін, біз енді осы жағдайға файл соңын кездестіргендей болып қалмау үшін бір әрекет етуіміз керек. Сонымен, біз `string` класынан `lexical_cast` класын пайдаланып, `int` бүтін санын оқығымыз келсе, мұның нәтижесінде біз соңғы беске байланысты "`123 5`" тіркесін емес, "`123`" немесе "`123 `" тіркесін аламыз.

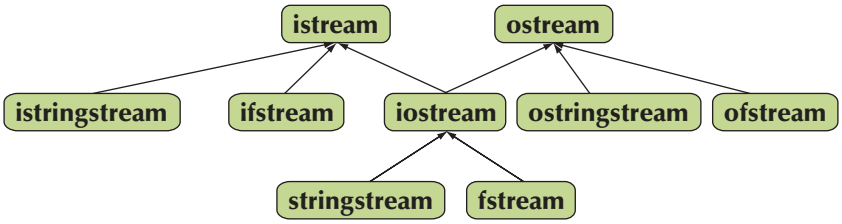
Мұндағы `lexical_cast` атауы `boost` кітапханасында қолданылады, біз оны регулярлық өрнектерді салыстыру үшін 23.6-23.9 бөлімдерінде пайдаланатын боламыз, бұл бір жағынан ыңғайлы болғанмен, бірақ біртүрлі сияқты. Болашақта ол C++ тілі стандартының жаңа нұсқасының бір бөлігі болып қалыптасады.

23.3 Енгізу-шығару ағымдары

Біз тіркестер мен басқа типтер арасындағы байланысты қарастыра отырып, енгізу-шығару ағымдарына келетін боламыз. Енгізу-шығару кітапханасы тек қана енгізу-шығару әрекеттерін орындап қана қоймай, ол компьютер жадындағы форматтар мен тіркестер типтері арасындағы түрлендіру ісін атқарады. Стандартты енгізу-шығару ағымдары символдар тіркестерін оқу, жазу және форматтауды орындайтын мүмкіндіктерді де қамтамасыз етеді. `iostream` кітапханасы 10-11-тарауларда сипатталған, сондықтан жай ғана қорытынды жасайық.

Stream I/O	
`in >> x`	мәліметтерді `x` объектісінің типіне сәйкес етіп, `in` ағымынан `x` объектісіне оқиды
`out << x`	`x` объектісін `out` ағымына `x` объектісінің типіне сәйкес етіп жазады
`in.getc(c)`	`in` ағымына `c` объектісіне символ оқиды
`getline(in,s)`	`in` ағымынан `s` тіркесіне сөз тіркесін оқиды

Стандартты ағымдар кластар иерархиясы (сатылары) түрінде ұйымдастырылған (14.3 бөлімді қ.)



Бұл кластар біріге отырып, файлдар мен тіркестерді пайдалану арқылы енгізу-шығару әрекеттерін орындауға мүмкіндік береді (және де файлдар мен тіркестер сияқты көрсетілетіндер, мысалы, пернетақта және экран). 10-11-тарауларда көрсетілген тәрізді `iostream` ағымы форматтауға үлкен мүмкіндіктер береді. Суреттегі тілсызықтар мұралауды бейнелейді (14.3 бөлімін қ.), сондықтан, мысалы, `stringstream` класын `iostream`, `istream` немесе `ostream` кластары орнына қолдануға болады.

Сөз тіркестері сияқты, енгізу-шығару ағымдарын мәліметтердің көптеген жиындарына және қарапайым символдарға да қолдануға болады. Егер сізге Unicode символдарын енгізу-шығарумен жұмыс істеу қажет болса, сарапшылардан кеңес сұрау керек; сіздің жазған программаңыз пайдалы болуы үшін тек тіл ережелеріне сәйкес келіп қана қоймай, белгілі бір жүйелік талаптарды да орындауға тиіс.

23.4 Ассоциативті контейнерлер (Maps)

Ассоциативті контейнерлер (ассоциативті жиымдар мен хеш-кестелер) мәтін өңдеуде өзекті рөл (каламбур) атқарады. Мұның себебі қарапайым – біз мәтін өңдегенде, ақпарат жинаймыз, ал ол көбінесе адам аттары, адрестері, әлеуметтік сақтандыру кәртішкелерінің нөмірі жұмыс орны, т.с.с. мәтін тіркестерімен байланысты болып келеді. Егер осы мәтін тіркестерінің кейбірін сандық мәндерге түрлендіруге болса да, оларды мәтін түрінде өңдеп, сол күйінда пайдаланған әрі қарайым, әрі оңай болып табылады. Осы тұрғыдан алғанда сөздер санын анықтау жақсы мысал бола алады (21.6 бөлімді қ.). Егер сізге `map` класымен жұмыс істеу ыңғайсыз болса, 21.6 бөлімді тағы бір оқып шығыңыз.

Электрондық пошта хабарламасын қарастырайық. Біз жиі-жиі электрондық пошта хабарламаларын және олардың тіркеу жазбаларын белгілі бір программа (мысалы, Thunderbird немесе Outlook) көмегімен іздейміз, талдауға тырысамыз. Көбінесе мұндай программалар хабарлама берушіні сипаттайтын нақты мәліметтерді бермейді, бірақ оны жіберген адамды, хабарды кімдер алғанын, оның қандай тораптар арқылы өткенін және де басқа мәліметтерді программадағы хат тақырыбында орналасқан мәтіннен білуге болады. Толық хабарлама осындай түрде

болады. Тақырыпты талдайтын мыңдаған құралдар бар. Олардың көпшілігі ақпарат алу үшін регулярлық өрнектерді (23.5-23.9 бөлімінде жазылған тәрізді) және де соларды өздеріне сәйкес хабарламалармен байланыстыру үшін ассоциативті жиымдар түрлерін пайдаланады. Мысалы, біз көбінесе бір адамнан келген, ортақ бір тақырыпқа арналған немесе нақты бір тақырыпты қамтитын хаттарды бөліп алу үшін электрондық пошта хабарламаларын іздейміз.

Мәтіндік файлдардан мәлімет алу үшін қолданылатын кейбір тәсілдерді көрсету үшін электрондық поштаның қарапайым файлын қарастырайық. Оның тақырыптары *www.faqs.org/rfcs/rfc2822.html* веб-парағынан алынған шынайы RFC2822 тақырыбы болып табылады. Мысал қарастырайық:

```
xxx
xxx
----
From:John Doe <jdoe@machine.example>
To: Mary Smith <mary@example.net>
Subject: Saying Hello
Date: Fri, 21 Nov 1997 09:55:06 -0600
Message-ID: <1234@local.machine.example>

This is a message just to say hello.
So, "Hello".
----
From:Joe Q. Public <john.q.public@example.com>
To: Mary Smith <@machine.tld:mary@example.net>
, , jdoe@test .example
Date:Tue, 1 Jul 2003 10:52:37 +0200
Message-ID: <5678.21-Nov-1997@example.com>

Hi everyone.
----
To: "Mary Smith: Personal Account" <smith@home.example>
From: John Doe <jdoe@machine.example>
Subject: Re: Saying Hello
Date: Fri, 21 Nov 1997 11:00:00 -0600
Message-ID: <abcd.1234@local.machine.tld>
In-Reply-To: <3456@example.net>
References:<1234@local.machine.example><3456@example.net>

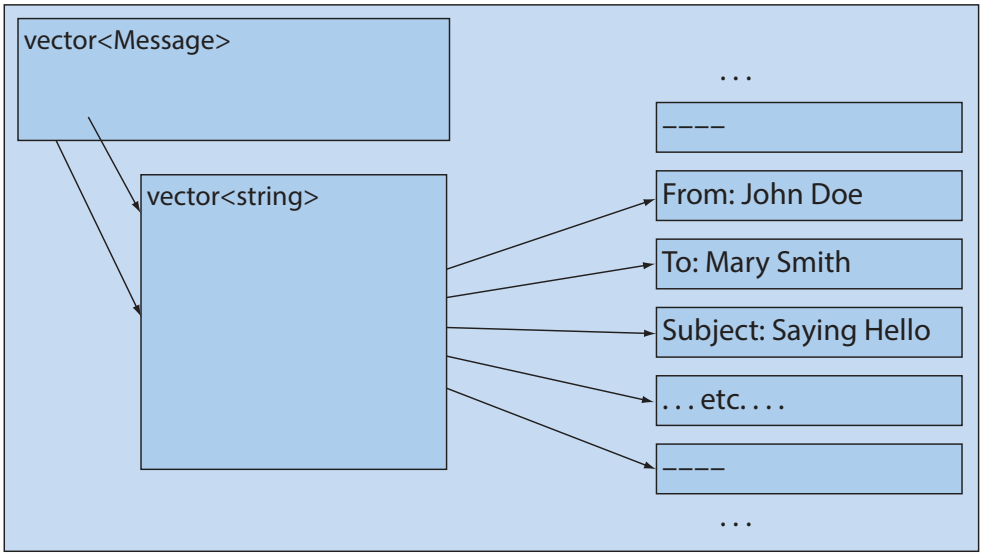
This is a reply to your reply.
----
----
```

Негізінде, біз көптеген ақпаратты алып тастадық та, талдауды жеңілдетіп, әрбір хабарламаны `----` (төрт пунктир сызық) символдары бар жолмен аяқтадық. Біз "ойын қосымшасын" жазбақшымыз, ол John Doe жіберген барлық хабарламаларды іздейді де, экранға солардың тақырыбын "Subject." Рубрикасынан кейін шығарады. Егер біз мұны жасай алсақ, онда көптеген қызықты нәрселерді де жасауды үйренеміз.

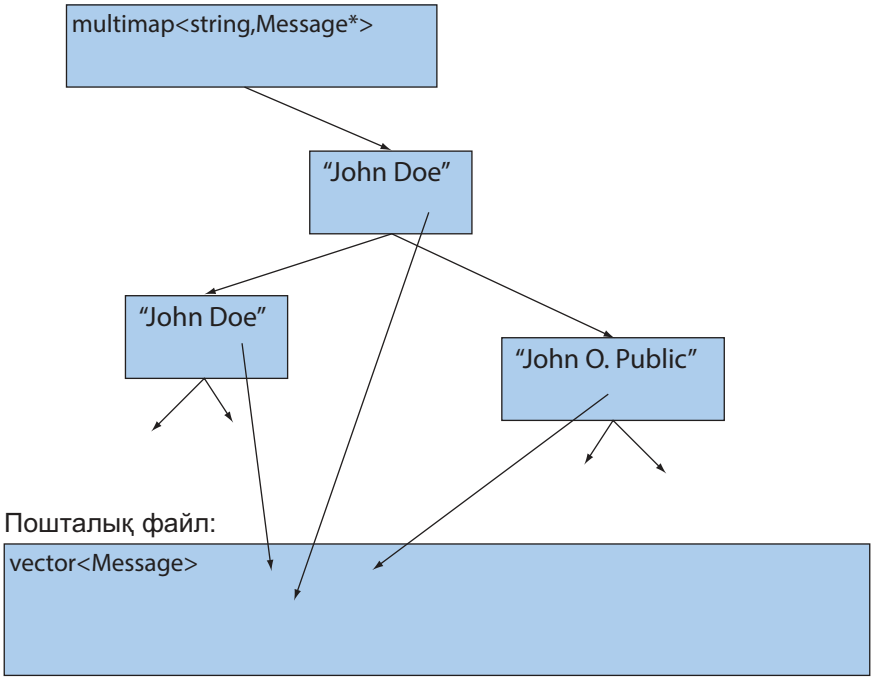
Біріншіден, біз мәліметтермен кез келген түрде қатынас құруды қалаймыз ба немесе оларды кіріс ағымдары деп талдауымыз керек пе, міне осыны шешіп алуымыз қажет. Біз бірінші нұсқаны таңдадық, өйткені шынайы программада бізді бірнеше хат жіберушілер немесе нақты бір адамнан келген ақпараттың бірнеше фрагменттері қызықтыруы мүмкін. Оның үстіне, бұл есепті шешу қиынырақ, сондықтан бізге үлкен шеберлік көрсету керек болады. Анығын айтсақ, біз қайтадан итераторларды пайдаланамыз.

Біздің негізгі идеямыз – пошта файлын толығынан `Mail_file` деп аталған құрылымға оқу болып табылады. Бұл құрылым пошталық файлдың барлық жолдарын (`vector<string>` класы объектісінде) және әрбір жеке хабарламаның басындағы және соңындағы индикаторларын (`vector<Message>` класының объектісінде) есте сақтайтын болады.

Пошталық файл:



Енді жолдар және хабарламалар бойынша әдеттегідей жылжу үшін біз итераторларды және де `begin()`, `end()` функцияларын қосамыз. Бұл схема бізге хабарламалармен жеңіл қатынас құруды қамтамасыз етеді. Мұндай құралды пайдаланып, біз бір адресаттан келген барлық хабарламаларды іздегенде, жеңіл табылатындай етіп, бәрін де біріктіріп жинақтауға мүмкіндік беретін "ойын қосымшасын" жазып шығамыз.



Қорытындылай келе, құрылымдармен қатынас құратын, өзіміз жасаған механизмді экранға шығарып, John Doe жіберген барлық хабарламалар тақырыптарын көрсетеміз. Ол үшін біз стандартты кітапхананың көптеген құралдарын пайдаланамыз.

```
#include<string>
#include<vector>
#include<map>
#include<fstream>
#include<iostream>
using namespace std;
```

vector<string> (біздің тіркестер векторы) класында **Message** класын итераторлар жұбы ретінде анықтаймыз.

```
typedef vector<string>::const_iterator Line_iter;

class Message {           // Message класының объектісі
//хабарламаның бірінші және соңғы жолдарына сілтеме жасайды
    Line_iter first;
    Line_iter last;
public:
    Message(Line_iterp1,Line_iterp2):first(p1), last(p2){}
```

```
    Line_iter begin() const { return first; }
    Line_iter end() const { return last; }
    // . . .
};
```

Mail_file класын мәтін жолдарынан және хабарламалардан тұратын құрылым ретінде анықтаймыз:

```
typedef vector<Message>::const_iterator Mess_iter;

struct Mail_file {
// Mail_file класының объектісі файлдағы барлық жолдардан
// тұрады және хабарламаларды пайдалануды жеңілдетеді

    string name;                // файл аты
    vector<string> lines;       // жолдар рет-ретімен
    vector<Message> m;          // хабарламалар рет-ретімен

    Mail_file(const string& n); // n файлын тіркеске оқу

    Mess_iter begin() const { return m.begin(); }
    Mess_iter end() const { return m.end(); }
};
```

Біз құрылым бойынша әрқашанда алға-артқа жылжу мүмкіндігіне ие болу үшін мәліметтер құрылымына итераторларды қосқанымызға назар аударыңыз. Негізінде, біз мұнда стандартты кітапханалық алгоритмдерді пайдаланғымыз келмейді, бірақ, қалап жатсақ, итераторлар мұны да істеуге мүмкіндік береді.

Хабарламадағы ақпаратты тауып алып, оқуымыз үшін екі қосымша функция қажет.

```
// Message класы объектісінде хат жіберушінің атын іздейді;
// егер аты табылса, true мәнін қайтарады
// егер аты табылса, оны s тіркесіне орналастырады:
bool find_from_addr(const Message* m, string& s);

// хабарлама тақырыбын қайтарады,
// егер ол жоқ болса, " " символын қайтарады:
string find_subject(const Message* m);
```

Сонымен, біз файлдан ақпарат шығарып алатын кодты жаза аламыз:

```
int main()
{
   Mail_file mfile("my-mail-file.txt");
   // mfile құрылымын файл мәліметтерімен инициалдаймыз
   // алдымен әрбір жіберушіден келіп түскен хабарламаларды
   // multimap класында жинаймыз:

   multimap<string, const Message*> sender;

   for(Mess_iter p = mfile.begin(); p!=mfile.end();++p) {
       const Message& m = *p;
       string s;
       if (find_from_addr(&m,s))
          sender.insert(make_pair(s,&m));
   }

   // енді multimap класының объектілері бойынша жылжып,
   // John Doe-дан келген хабарламалар тақырыптарын шығарып аламыз:
   typedef multimap<string, const Message*>::
                              const_iterator MCI;
   pair<MCI,MCI> pp =
      sender.equal_range("John Doe<jdoe@machine.example>");
   for(MCI p = pp.first; p!=pp.second; ++p)
        cout << find_subject(p->second) << '\n';
}
```

Ассоциативті жиымдарды пайдалануды толығырақ қарастырайық. Біз **multimap** (§20.10, B.4 бөлімдері) класын пайдаландық, өйткені бір жерде бір адрестен келген көптеген хабарламаларды жинап алғымыз келді. Стандартты **multimap** класы міне осыны (бір ғана кілт арқылы элементтермен қатынас құруды жеңілдете отырып) ғана істейді. Біз күткендей (көбінесе осылай), есебіміз екі шағын есепке бөлінеді:

- ассоциативті жиым құру;
- ассоциативті жиымды пайдалану.

Біз барлық хабарламаларды қарастыра келіп және оларды **insert()** функциясы арқылы енгізе отырып, **multimap** класы объектілерін құрамыз:

```
for (Mess_iter p = mfile.begin(); p!=mfile.end(); ++p) {
    const Message& m = *p;
    string s;
    if (find_from_addr(&m,s))
       sender.insert(make_pair(s,&m));
}
```

Ассоциативті жиымға `make_pair()` функциясы көмегімен құрылған қос параметр (кілт, мән) енгізіледі. Хат жіберушінің атын табу үшін, "қолдан жасалған" `find_from_addr()` функциясын пайдаланамыз. Егер адресі табылмаса, онда бос сөз тіркесін қайтарамыз.

Неге біз `m` сілтемесін пайдаланып, оның адресін жібереміз? Неге тікелей `p` итераторын пайдаланып, `find_from_addr(p,s)` функциясын шақырмасқа? Өйткені біз, тіпті `Mess_iter` итераторы `Message` класы объектісіне сілтеме жасап тұрғанын білсек те, оның нұсқауыш түрінде жасаламағанына ешқандай кепілдік берілмейді.

Біз неге алдымен `Message` класы объектісін жазып, сонан кейін ғана `multimap` класы объектісін жасадық? Неге бірден `Message` класы объектісін `map` класының ассоциативті жиымына қоспадық? Оның себебі қарапайым іргелі сипатта болып отыр:

- Алдымен біз көптеген заттар үшін қолдануға болатын әмбебап құрылым жасап аламыз.
- Сонан соң оны нақты бір қосымшада пайдаланамыз.

Сонымен, біз белгілі бір деңгейде қайталанып қолданылатын компоненттер коллекциясын жасаймыз. Егер біз бірден `Mail_file` класының объектісінде ассоциативті жиым жасасақ, онда оны басқа бір есепті шығару үшін қолданғымыз келсе, солардың әрқайсысы үшін оларды қайта анықтауға мәжбүр болатын едік. Бөле-жара айтар болсақ, біздің `multimap` класының объектісі (көп мәнді түрде senders деп аталған) `Address` өрісі арқылы сұрыпталған (реттелген) болып отыр. Басқа қосымшалардың басым көпшілігі сұрыптау кезінде басқа критерилерді қолдануы мүмкін, мысалы: `Return`, `Recipients`, `Copy-to`, `Subject fields` өрістері уақыт белгілері, т.б. бойынша сұрыпталуы мүмкін.

Қосымшаларды кезеңдер (немесе оларды кейде *қабаттар* – layers бойынша құру деп айтады) бойынша құру программаларды жобалауды, жүзеге асыруды, құжаттамалауды және қолдануды (сүйемелдеуді) бірсыпыра деңгейде қарапайым етуі мүмкін. Әңгіме мынада, қосымшаның әрбір бөлігі жеке бір есепті шығарады және оны біз күткендей дәрежеде орындайды. Басқа жағынан, барлығын да бірден жасау үшін үлкен ақыл керек. Әрине, электрондық поштаның ақпараттарын және хабарламалар тақырыптарын шығарып алу – бұл қосымша жасаудың шағын ғана мысалдары. Есептерді жекелеп бөліп алу, модульдерді де бір-бірінен бөліп алу, қосымшаларды жасау барысында оларды кеңейту істерінің мәндері іс барысында айқын көріне бастайды.

Ақпараттарды шығарып алу үшін біз "`John Doe`" түйінді сөзі кездесетін барлық тіркестерді `equal_range()` функциясын қолдана отырып, іздеп тауып аламыз (В 4.10 бөлімін қ.). Сонан кейін `equal_range()` функциясы қайтаратын `[first,second)` тізбегінің барлық элементтері бойынша жылжи отырып, `find_subject()` функциясы көмегімен хабарламалар тақырыптары шығарып аламыз.

```
typedefmultimap<string,constMessage*>::const_iteratorMCI;

pair<MCI,MCI> pp = sender.equal_range("John Doe");

for (MCI p = pp.first; p!=pp.second; ++p)
   cout << find_subject(p->second) << '\n';
```

map класы объектілерінің элементтері бойынша алға жылжи отырып, **pair** класының кез келген басқа бір объектісіндегі сияқты біз қос параметр (кілт, мәні) тізбегін аламыз, оның бірінші элементі (мұнда **string key** класының кілті) **first** деп, ал екіншісі (мұнда **Message** класының объектісі) – **second** деп аталады (21.6 бөлімді қ.).

23.4.1 Жүзеге асыру нақтылықтары

Енді, біз пайдаланып жүрген функцияларымызды жүзеге асыруымыз керек. Әрине, қағазды босқа шығын қылмай, ағашты да сындырмай, оқырмандар бұл есепті өздері шығарар дей салуға да болар еді, бірақ біз мысал толық болуы тиіс деп шештік.

Mail_file класының конструкторы файл ашып, **lines** және **m** векторларын жасайды:

```
Mail_file::Mail_file(const string& n)
 // "n" атты файл ашады
 // "n" файлындағы сөз тіркестерін lines векторына оқиды
 // lines векторындағы хабарламаларда тауып, соларды
 // m векторына орналастырады
 // қарапайымдылық үшін әрбір хабарлама "---- " line
 // тіркесімен аяқталады
{
   ifstream in(n.c_str());        // файл ашу
   if (!in) {
          cerr << "no " << n << '\n';
          exit(1);      // программаны орындауды тоқтату
   }
   string s;
   while getline(in,s)) lines.push_back(s);
   // тіркестер векторын құру

   Line_iter first = lines.begin();
   //хабарламалар векторын құру build the vector of Messages
```

```
        for (Line_iter p = lines.begin(); p!=lines.end(); ++p) {
            if (*p == "---- ") {        // хабарлама соңы
                m.push_back(Message(first,p));
                first = p+1;  // ---- тіркесі хабарлама бөлігі емес
            }
        }
    }
```

Қателерді өңдеу өте қарапайым сипатта болып отыр. Егер бұл программаны өз достарымыз үшін жазсақ, онда оны жақсылау етіп жазуға тырысар едік.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

"Қатені жақсырақ етіп өңдеу" деген нені білдіреді? `Mail_file` класының конструкторын "`----`" тіркесіне байланысты оған форматтау қателері әсер ететіндей түрде өзгертіңіз.

Біз файлдағы ақпаратты қалай идентификациялауды (регулярлық өрнектерді және 23.6–10 бөлімдерді пайдалану арқылы) анықтағанымызша, `find_from_addr()` және `find_subject()` функцияларының нақты мазмұны болмайды.

```
int is_prefix(const string& s, const string& p)
// p тіркесі s тіркесінің бірінші бөлігі болып табыла ма?
{
      int n = p.size();
      if (string(s,0,n)==p) return n;
      return 0;
}

bool find_from_addr(const Message* m, string& s)
{
     for(Line_iter p = m->begin(); p!=m->end(); ++p)
         if (int n = is_prefix(*p,"From: ")) {
            s = string(*p,n);
            return true;
         }
     return false;
}

string find_subject(const Message& m)
{
    for(Line_iter p = m.begin(); p!=m.end(); ++p)

       if (int n = is_prefix(*p,"Subject: "))
            return string(*p,n);
    return "";
}
```

Біздің ішкі тіркестерді қалай пайдаланғанымызға назар аударыңыз: `string(s,n)` конструкторы `s` тіркесінің `s[n]` элементінен бастап, оның соңына шейінгі (яғни `s[n]..s[s.size()-1]`) символдардан тұратын тіркес жасайды да, ал `string(s,0,n)` конструкторы `s[0]..s[n-1]` символдарынан тұратын тіркес құрады. Бұл операторлар жаңа тіркестер құрып, символдарды көшіретін болғандықтан, программаның жұмыс өнімділігін төмендетпес үшін, оларды өте сақ пайдаланған жөн.

Неге `find_from_addr()` және `find_subject()` функциялары бір-бірінен мұндай айырмашылықтары бар? Мысалы, олардың бірі `bool` типіндегі айнымалыны қайтарса, ал екіншісі – `string` класындағы объектіні қайтарады. Мұның себебі, біз келесі әрекеттерді жүзеге асырғымыз келген еді:

- `find_from_addr()` функциясы адрестің бос тіркесін (`""`) іздеу мен адрес көрсетілген тіркесі, яғни жолы мүлде жоқ мүмкіндікті іздеудің айырмасын береді. Мұның біріншісінде `find_from_addr()` функциясы `true` (өйткені ол адресті тапты) мәнін қайтарып, `""` мәнін (өйткені адрестік адрес бос болып шықты) `s` тіркесіне меншіктейді. Екінші жағдайда функция `false` (өйткені файлда адрестік сөз тіркесі тіпті жоқ болып шықты) мәнін қайтарады.

- `find_subject()` функциясы хабарламаның тақырып жолы бос болғанда және ол тіпті болмаған кезде де `""` тіркесін қайтарады.

`find_from_addr()` функциясы орындайтын мұндай айырмашылық қаншалықты пайдалы болып табылады? Бұл қажет пе? Біз мұны әрі қажет, әрі пайдалы деп санаймыз. Мәліметтер файлынан ақпарат іздегенде, бұл айырмашылық қайта-қайта көзге түседі: біз қажетті тіркесті таптық па және онда бізге керекті мәлімет бар ма? Нақты программада қолданушыларға осындай айырмашылық жасау мүмкіндігін беретін екі функцияны да, `find_from_addr()` және `find_subject()`, `find_from_addr()` функциясы стилінде жазып шығу керек еді.

Бұл программа жұмыс өнімділігі жағынан тиімді болып табылмайды, бірақ қалыпты жағдайда ол әжептеуір жылдам істейді деп үміттенеміз. Мысалы, ол кіріс файлын бір-ақ рет оқиды да, оның мәтіндерінің бірнеше көшірмелерін есте сақтамайды. Көлемді файлдар үшін `multimap` класын `unordered_multimap` класымен алмастырған тиімдірек болар еді, бірақ тәжірибе арқылы тексерусіз ол программаның тиімділігін қаншаға көтеретінін айту қиын.

Стандартты ассоциативті контейнерлерге кіріспені (`map`, `multimap`, `set`, `unordered_map` және `unordered_multimap`) 21.6 бөлімнен қ.

23.5 Мәселе

Енгізу-шығару ағымдары және `string` класы бізге символдар тізбегін оқуға, жазуға, есте сақтауға және олармен негізгі операциялар орындауға көмектеседі. Дегенмен мәтінмен жұмыс істейтін көп жағдайларда тіркестің ішкі бөлігін талдау немесе ұқсас тіркестерді қарастыру қажеттілігі туындап жатады. Қарапайым мысал қарастырайық. Электрондық пошта хабарламасын (сөз тізбектері) алып, онда U.S. қысқартылған сөз әріптерінің (аббревиатурасының) және ZIP пошталық кодының (екі әріп пен соған жалғасқан бес цифрлар тізбегі) бар немесе жоқ екендігін тексерейік:

```
string s;
while (cin>>s) {
    if (s.size()==7
    && isalpha(s[0]) && isalpha(s[1])
    && isdigit(s[2]) && isdigit(s[3]) && isdigit(s[4])
    && isdigit(s[5]) && isdigit(s[6]))
        cout << "found " << s << '\n';
}
```

Мұнда егер `x` – `letter` класының объектісі болса, `isletter(x)` мәні `true` болады, ал егер `x` – цифр болса, онда `isdigit(x)` мәні `true` болады (11.6 бөлімді қ.). Осы (өте) қарапайым шешімнің өзінде бірнеше проблема жатыр:

- Ол өте күрделі (төрт жол, функцияны сегіз рет шақыру бар).
- Біз өз ішкі мәтінінен бос орынмен бөлінбеген әрбір ZIP пошта индексін (әдейі) өткізіп жібереміз (мысалы, "`TX77845`", `TX77845-1234` және `TX77845`).
- Біз әріптер мен цифрлар арасы бос орынмен бөлінген пошта индексін (әдейі) өткізіп жібереміз (мысалы, `TX 77845`).
- Біз әріптері төменгі регистрде терілген әрбір ZIP пошта индекстерін (әдейі) қабылдаймыз (мысалы, `TX 77845`).
- Егер сіз басқа форматта берілген ZIP пошта индекстерін талдайтын болып шешім қабылдасаңыз (мысалы, `CB3 0FD`), онда сіз барлық кодты қайта жазуға мәжбүр боласыз.

Бұдан жақсы тәсіл болуы тиіс! Оны сипаттау алдында қойылған мәселелерді қарастырайық. Мысалы, біз көрсетілген жағдайларды өңдейтін мүмкіндік қосылған "бұрынғы жақсы қарапайым кодты" қалдыруды қалаған болдық делік.

- Егер біз бір форматты ғана өңдеуді қаласақ, онда `if` немесе `switch` нұсқауларын қосу керек.

- Егер біз жоғарғы және төменгі регистрлерді есепке алғымыз келсе, онда тіркестерді тікелей түрде түрлендіру (әдетте төменгі регистр) керек немесе қосымша `if` нұсқауын қосу қажет.
- Біз іздеу жүргізілетін ішкі мәтінді белгілі бір тәсілмен (қалай?) сипаттауымыз керек. Бұл біз сөз тіркестерімен емес, жеке символдармен жұмыс істеуіміз керек дегенді білдіреді, яғни `iostream` ағымдары беретін (7.8.2 бөлімді қ.) көптеген артықшылықтарды жоғалтып аламыз.

Егер қаласаңыз осы стильде код жазып көріңіз, бірақ мұндайда ерекше жағдайларды өңдеуге арналған `if` нұсқауларының көптігінен шатасып кету қаупі бар. Тіпті осы қарапайым мысалдың өзінде біз таңдау (мысалы, бес және тоғыз таңбалы пошта индекстерін есепке алу керек пе) алдында тұрамыз. Көптеген басқа мысалдарда бізге леп белгілерімен жұмыс істеуге тура келеді (мысалы, `123!` және `123456!` тәрізді леп белгісімен жалғасатын кез келген цифрлар тізбегі). Сөз соңында айтарымыз, префикстер мен суффикстер жайлы да ұмытуға болмайды. Біз бұрын көрсеткендей (11.1–2 бөлімді қ.) әртүрлі форматтарға қатысты қолданушылардың қалауы программалаушылардың жүйелілік пен қарапайымдылыққа ұмтылысымен шектелмесе керек. Тек бір ғана күн-ай (дата) мерзімінің әртүрлі жазылу тәсілдері туралы ойланып көріңіз:

```
2007-06-05
June 5, 2007
jun 5, 2007
5 June 2007
6/5/2007
5/6/07
  . . .
```

Осы сәтте, әрине, бұдан бұрын да болуы мүмкін, тәжірибелі программалаушы: "Бұдан жақсырақ тәсіл болуы тиіс!" деп (ординарлық кодты күрделендіре бергенше), оны іздей бастауы мүмкін. Бұл мәселенің қарапайым және кең таралған шешімі болып *регулярлық өрнектер* (regular expressions) деп аталатын нәрсені пайдалану болып табылады.

Регулярлық өрнектер көптеген мәтіндерді өңдеу тәсілдерінің және Unix жүйесіндегі grep (8-жаттығуды қ.) командасының негізі болып және де осындай мәселелерді шешуге екпінді түрде қолданылып келе жатқан программалау тілдерінің (AWK, Perl, және PHP сияқты) маңызды бөлігі болып табылады.

Біз пайдаланатын регулярлық өрнектер C++ (C++0x) тілінің келесі стандартының бір бөлігі болатын кітапханада жүзеге асырылған. Оларды Perl тілінің регулярлық өрнектерімен салыстыруға болады. Осы тақырыпқа арналған көптеген кітаптар, оқулықтар және анықтамалықтар бар, мысалы, C++ тілін стандарттау комитетінің жұмыс есебі, Джон Мэддокс (John Maddoc) `boost::regex` құжаттамасы және Perl тілінің оқулықтары. Мұнда біз іргелі түсініктер береміз және де регулярлық өрнектерді пайдалану тәсілдерін баяндаймыз.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Соңғы екі абзацта ешқандай түсініктемелерсіз "байқаусызда" бірнеше адам аттары мен қысқартылған сөздер аббревиатуралары пайдаланылған. Солар туралы веб-ақпарат іздеп көріңіз.

23.6 Регулярлық өрнектер идеясы

Регулярлық өрнектердің негізгі идеясы оның біз мәтін ішінен іздеген шаблонды (pattern) анықтауында болып табылады. Біз қарапайым `TX77845` тәрізді пошталық кодтың шаблонын қалай дәл сипаттай алатынымызды қарастырып көрейік. Алғашқы тәжірибе нәтижесі мынадай түрде болды:

```
wwddddd
```

мұндағы `w` символы кез келген әріпті білдіреді, ал `d` – кез келген символ. Біз `w` ("word" сөзінен) символын пайдаланамыз, өйткені `l` ("letter" сөзінен) символын `1` цифрымен шатастырып алу қаупі бар. Мұндай белгілеулер біздің қарапайым мысалымыз үшін сәйкес келеді, бірақ егер біз оны тоғыз цифрдан тұратын (мысалы, `TX77845-5629`) ZIP пошта коды форматын сипаттау үшін қолданғымыз келсе, не болар еді. Мұндай шешімге сіз не дер едіңіз?

```
wwddddd- dddd
```

Бұлар логикалық тұрғыдан дұрыс болып көрінгенмен, `d` символының "кез келген цифр" екенін, ал `-` таңбасының "тек қана" дефис екенін қалай түсінуге болады? Бізге белгілі бір тәсілмен `w` мен `d` арнайы таңбалар екенін түсіндіріп кету керек: олар өздерін емес (`w` символы "`a` немесе `b` немесе `c` немесе . . ." дегенді, ал `d` символы "`1` немесе `2` немесе `3` немесе . . ." дегенді білдіреді), символдар класын көрсетеді. Бұның бәрі күрделі нәрсе. Енді біз C++ тіліндегі сияқты символдар класы атауын білдіретін әріпке кері қиғаш сызықты қосайық (мысалы, `\n` тіркесі жаңа жолға көшуді білдіреді). Сонда келесі тіркесті аламыз:

```
\w\w\d\d\d\d\d-\d\d\d\d
```

Онша әдемі көрінбейді, бірақ біз сәйкессіздікті жойдық, ал кері қиғаш сызықтар олардан кейін "қалыптан тыс бір нәрсе" тұрғанын анық білдіріп тұр. Мұнда жай ғана қайталанатын символдар тізбегі көрініп тұр. Бұл бір жағынан жалықтырса, екінші жағынан қателер туындата алады. Сіз дефиске дейінгі кері қиғаш сызық алдарында бес цифр тұрғанын, ал одан кейін төрт цифр тұратынын тез санап айтып бере аласыз ба? Біз мұны жасап шықтық, бірақ мұнда жай ғана `5` және `4` деп *айту* жеткіліксіз, оған көз жеткізу үшін оларды санап шығу керек.

Әрбір символдан кейін оның неше рет қайталанатынын көрсететін санауыш қоюға да болар еді. Мысалы:

```
\w2\d5-\d4
```

Бірақ, негізінде, бізге бұл шаблондағы `2`, `5` және `4` сандары жай ғана `2`, `5` және `4` цифрлары ғана емес олар санауыш мәндері екенін көрсететін белгілі бір синтаксистік конструкция керек болады. Санауыш мәндерін жүйелік жақшалармен ерекшелейік:

```
\w{2}\d{5}-\d{4}
```

Енді `{` символы кері қиғаш сызық `\` сияқты арнайы символ болып табылады, бірақ мұны істемей кетуге болмайды, сондықтан біз енді осы әрекетті есепке алып отыруымыз керек.

Сонымен, бәрі дұрыс секілді, бірақ біз екі нәрсені ұмыттық: ZIP пошталық кодындағы соңғы төрт цифрдың болуы міндетті емес. Кейде екі нұсқа да: `TX77845` және `TX77845-5629` қолданыла алады. Мұны екі негізгі тәсілмен көрсетуге болады:

`\w{2}\d{5}` немесе `\w{2}\d{5}-\d{4}`

және

`\w{2}\d{5}` және міндетті емес `-\d{4}`

Дәлірек айтқанда, алдымен біз `\w{2}\d{5}` және `-\d{4}` тіркестері `\w{2}\d{5}-\d{4}` тіркесінің бөлігі болатынын айту үшін топтау (немесе жартылай шаблон) идеясын көрсетуіміз керек. Әдетте топтау әрекеті жай жақшалар көмегімен орындалады:

```
(\w{2}\d{5})(\d{4})
```

Енді біз шаблонды екі шағын *ішкі шаблонға* (sub-patterns) бөлуге тиіспіз, яғни олармен не істейтінімізді көрсетуіміз керек. Бұрынғыдай, жаңа мүмкіндікті енгізу жаңа арнайы символды пайдалану арқылы атқарылады: енді `(` символы `\` және `{` символдары сияқты арнайы символдарға жатады. Әдетте `|` символы "немесе" (баламаны, яғни альтернативаны) операциясын белгілеу үшін қолданылады, ал `?` символы белгілі бір шартты символды (міндетті емес) белгілеу үшін пайдаланылады. Сонымен, енді былай жаза аламыз:

```
(\w{2}\d{5})|(\w{2}\d{5}-\d{4})
```

және

```
(\w{2}\d{5})(-\d{4})?
```

Мұнда да жүйелі жақшалар сияқты санауыштарды (мысалы, `\w{2}`) белгілегенде, сұрақ белгісі (`?`) суффикс түрінде қолданылады. Мысалы, `(\d{4})?` тіркесі `"\d{4} бөлігінің міндетті емес" екендігін білдіреді; яғни біз алдарында дефис тұрған төрт цифрды суффикс деп түсінеміз. Негізінде, біз жай жақшаны белгілі бір операцияны орындау кезінде бес таңбалы ZIP пошталық кодын (яғни, `\w{2}\d{5}`) ерекшелеу үшін қолданбаймыз, сондықтан оларды алып тастауға болады:

```
\w{2}\d{5}(-\d{4})?
```

Біздің 23.5 бөлімінде көрсетілген шешімімізді аяқтау үшін, екі әріптен соң міндетті емес бос орын таңбасын қоса аламыз:

```
\w{2} ?\d{5}(-\d{4})?
```

Сұрақ белгісін "`?`" жазу таң қаларлық нәрсе секілді, бірақ бос орыннан кейінгі сұрақ белгісі бұл бос орынның міндетті емес екендігін білдіреді. Егер біз бос орын қате терілген болып көрінбесін десек, оны жақшаға алып жазуымыз керек еді:

```
\w{2}( )?\d{5}((-\d{4})?
```

Егер біреу бұлай жазу өте анық жазылмаған деп айтса, онда бізге бос орын үшін бір белгі ойлап табу керек болар еді, мысалы, `\s` (`s` "space" сөзінен). Мұнан кейін жазуымыз былай болар еді:

```
\w{2}\s?\d{5}(-\d{4})?
```

Ал егер біреу әріптен соң екі бос орын қойса ше? Жоғарыда анықталған шаблонға сәйкес бұл біздің `TX 77845` кодын емес, `TX77845` және `TX 77845` кодын қабылдағанымызды білдірер еді. Бұл дұрыс емес.

Бізге "ешқандай бос орын жоқ немесе бір бос орын немесе бірнеше бос орын" бар деп айтқызатын құрал керек, сондықтан біз `*` суффиксін енгіземіз.

```
\w{2}\s*\d{5}(-\d{4})?
```

Мұндағы әрбір кезеңді берілген тәртіптегі логикалық тізбекте орындаған дұрыс болады. Бұл шаблондық белгілеу жүйесі логикалық түрде қисынды әрі қысқа. Оған қоса, біз жобалық шешімдерді көктен алған жоқпыз: біз таңдап алған белгілеу жүйесі өте кең таралған. Мәтіндерді өңдеуге арналған көптеген есептерді шығару кезінде бізге осындай символдарды оқу және жазу керек. Иә, бұл жазбалар пернетақтаны билеген мысық басқан тәрізді бір жерден шыққан қате олардың мағынасын толығынан өзгертіп жібереді, бірақ мұнымен санасуға тура келеді. Біз

бұдан жақсы ешнәрсе ұсына алмаймыз, осы стиль соңғы отыз жылда кеңінен таралып кетті. Ол алғашқы рет Unix жүйесіндегі **grep** командасында қолданылған болатын, бірақ сол кездің өзінде оны жаңа деп айтуға болмайтын еді.

23.7 Регулярлық өрнектер арқылы іздеу

Енді алдыңғы бөлімдегі ZIP пошталық кодтарының шаблонын файлдағы пошталық кодтарды іздеу үшін пайдаланайық. Программа шаблонды анықтайды да, сонан кейін оны файлдағы сөз тіркестерін бірінен кейін бірін оқи отырып іздейді. Программа белгілі бір жолдан шаблонды тапқан кезде, ол сол жол нөмірін және табылған кодты экранға шығарады:

```
#include <boost/regex.hpp>
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int main()
{
   ifstream in("file.txt");       // енгізу файлы
   if (!in) cerr << "файл жоқ\n";
                                                  -
   boost::regex pat ("\\w{2}\\s*\\d{5}(  \\d{4})?");
                                 // ZIP кодының шаблоны
   cout << "шаблон: " << pat << '\n';

   int lineno = 0;
   string line;          //  енгізу буфері
   while (getline(in,line)) {
        ++lineno;
        boost::smatch matches;
        // мұнда сәйкес келген тіркестерді жазамыз
        if (boost::regex_search(line, matches, pat))
          cout << lineno << ": " << matches[0] << '\n';
   }
}
```

Бұл программа түсіндіруді керек етеді. Алдымен келесі фрагментті қарастырайық:

```
#include <boost/regex.hpp>
. . .
```

```
boost::regex pat("\\w{2}\\s*\\d{5}(-\\d{4})?");
// ZIP коды шаблоны
boost::smatch matches;
// мұнда сәйкес келген кодтарды жазамыз
if (boost::regex_search(line, matches, pat))
```

Біз болашақта стандартты кітапхананың бөлігі болып кететін **Boost.Regex** кітапханасының жүзеге асырылған нұсқасын пайдаланамыз. **Boost.Regex** кітапханасын пайдалану үшін оны орнатып іске қосу (инсталяциялау) керек. **Boost.Regex** кітапханасына қандай мүмкіндіктер қатысты екенін көрсету үшін біз квалификатор ретінде тікелей **boost** атаулар кеңістігін, яғни **boost::regex** деп көрсетеміз.

Регулярлық өрнектерге қайтып оралайық! Келесі кодты қарастырайық:

```
boost::regex pat ("\\w{2}\\s*\\d{5}(-\\d{4})?");
cout << "pattern: " << pat << '\n';
```

Мұнда біз алдымен **pat** (**regex** типінде) шаблонын анықтап алып, сонан кейін оны баспаға шығардық. Біздің жазғанымызға назар салыңыз.

```
"\\w{2}\\s*\\d{5}(-\\d{4})?"
```

Егер программаны іске қоссақ, экраннан мынадай жолды көрер едік:

```
pattern:\w{2}\s*\d{5}(-\d{4})?
```

C++ тілінің тіркестік литералдарында кері қиғаш сызық басқару символын көрсетеді, сондықтан литералдық тіркестерде бір кері қиғаш сызық **** орнына оның екеуін **** жазу керек.

Негізінде, **regex** типіндегі шаблон **string** класы объектісінің бір түрі болып табылады, сондықтан біз оны **<<** операторы арқылы баспаға шығара аламыз. **Regex** класы – бұл **string** класы объектісінің жай ғана бір түрі емес, оның **regex** класының объектісін инициалдау кезінде (немесе меншіктеу операторын орындау кезінде) жасалған шаблондарды салыстырыратын күрделі механизмі бұл кітап ауқымынан тысқары жатыр. Дегенмен, біз **regex** класының объектісін ZIP пошта кодының шаблонымен инициалдағандықтан, біз оны өз файлымыздың әрбір жолына қолдана аламыз:

```
boost::smatch matches;
if (boost::regex_search(line, matches, pat))
   cout << lineno << ": " << matches[0] << '\n';
```

regex_search(line, matches,pat) функциясы **line** жолындағы **pat** объектісінде сақталып тұрған регулярлық өрнекке кез келген сәйкестікті іздейді.

Егер ол белгілі бір сәйкестікті тапса, онда оны **matches** объектісінде сақтайды. Әрине, егер сәйкестік анықталмаса, **regex_search(line, matches, pat)** функциясы **false** мәнін қайтарады.

matches айнымалысының типі **smatch** болады. **s** әрпі "**sub**" дегенді білдіреді. Негізінде, **smatch** типі вектор бөліктерінің бірдей екендігін көрсетеді. Мұндағы **matches[0]** бірінші элементі толығынан бірдей. Егер **i<matches.size()** болса, біз **matches[i]** элементін тіркес ретінде қабылдаймыз. Сонымен, егер осы регулярлық өрнек үшін шаблон бөліктерінің максимал, яғни ең үлкен саны **N** болса, онда біз мынадай өрнек аламыз: **matches.size()==N+1**.

Мұндағы *шаблон бөлігі* (sub-pattern) деп отырғанымыз не? Бұған: "Шаблон ішіндегі жақшаға алынғанның барлығы" деген жауап айта аламыз.

\\w{2}\\s*\\d{5}(-\\d{4})? шаблонына қарап, біз ZIP төрт таңбалы кодының жақшаға алынғанын көреміз. Сонымен, біз тек бір шаблон бөлігін ғана, яғни **matches.size()==2** өрнегін көріп тұрмыз. Бұған қоса, біздің осы төрт цифрға ғана қол жеткізе алатынымыз байқалады. Мысал қарастырайық:

```
while (getline(in,line)) {
    boost::smatch matches;
    if (boost::regex_search(line, matches, pat)) {
         cout << lineno << ": " << matches[0] << '\n';
                                  // толығымен бірдей
         if (1<matches.size() && matches[1].matched)
               cout << "\t: " << matches[1] << '\n';
                                  // бөлігі ғана бірдей
         }
}
```

Дәлірек айтқанда, біз **1<matches.size()** өрнегін тексеруге міндетті емеспіз, өйткені шаблонды қарап шықтық, бірақ бұған бізді (біз pat объектісінде сақталған әртүрлі объектілермен тәжірибе жасап тұрғандықтан және олардың көбінде бір ғана шаблон бөлігі болмағандықтан) жеңіл сенімсіздік итермелейді. Біз оның **matches** мүшесін қарай отырып, мұнда ол **matches[i].matched** түрінде, шаблон бөлігінің табылғандығын тексере аламыз. Бізді мынадай жағдай: егер **matches[i].matched** мәні **false** болса, онда **matches[i]** шаблон бөлігі сәйкестігі жоқ бос тіркес түрінде шығарылады. Осылайша, егер шаблон бөлігі кездеспесе, мысалы жоғарыда келтірілген шаблон үшін **matches[17]** сияқты, онда ол сәйкестігі жоқ шаблон ретінде қарастырылады.

Біз бұл программаны келесідей жолдары бар файлға қолданып көрдік:

```
address TX77845
ffff tx 77843 asasasaa
ggg TX3456-23456
howdy
```

```
zzz TX23456-3456sss ggg TX33456- 1234
cvzcv TX77845- 1234 sdsas
xxxTx77845xxx
TX12345- 123456
```

Нәтижесі төменде келтірілген:

```
pattern: "\w{2}\s*\d{5}(  \d{4})?"
1:TX77845
2: tx 77843
5:TX23456-3456
:-3456
6:TX77845- 1234
:- 1234
7:Tx77845
8:TX12345- 1234
:- 1234
```

Бірнеше маңызды сәттерді атап өтейік:

- Біз `ggg` символдары (мұнда қандай қате бар) бар жолда дұрыс форматталмаған ZIP кодымен өзімізді шатастыруға жол бермедік.
- `zzz` символдары бар жолдан біз тек бірінші ZIP кодын таптық (біз жолдан тек бір кодты ғана іздейміз).
- 5- және 6-жолдан дұрыс суффикстер таптық.
- 7-жолдан `xxx` символдарының арасында жасырын тұрған ZIP кодын таптық.
- `TX12345- 123456` жолында жасырын тұрған ZIP кодын таптық (өкінішке орай).

23.8 Регулярлық өрнектер синтаксисі

Біз регулярлық өрнектерді салыстырудың өте қарапайым мысалын қарастырдық. Енді регулярлық өрнектерді (`regex` кітапханасында пайдаланылған түрде) толығырақ жүйелі түрде қарастыратын кез келді.

Регулярлық өрнектер (regular expressions, "regexps" or "regexs"), негізінде, символдық шаблондарды өрнектеуге арналған шағын тіл болып табылады. Бұл қуатты (көрнекті) және ықшам тіл кейде тым бүкпелі болып та көрінеді. Регулярлық өрнектерді он жылдан аса пайдалану нәтижесінде бұл тілде көптеген сезімтал қасиеттер пайда болды және оның бірнеше нұсқалары (диалектілері) бар. Мұн-

да біз регулярлық өрнектердің қазіргі кездегі ең кең таралған диалектісі (PERL тілі) болып табылатын оның ішкі бір жиынын (көлемді және пайдалы) сипаттаймыз. Оқырмандарға регулярлық өрнектер жайлы толығырақ мәлімет қажет болса немесе оны басқаларға түсіндіру керек болып жатса, онда қажетті мәліметтердің бәрін де веб-тен тауып алуға болады. Көптеген оқулықтар (сапалары әртүрлі) мен спецификациялар бар. Мысалы, веб ортасынан `boost::regex` спецификациясын және Стандарттау комитеті (WG21 TR1) қабылдаған оның эквивалентін (баламасын) жеңіл тауып алуға болады.

`boost::regex` кітапханасы ECMAscript, POSIX, awk тілдерінің, grep және egrep утилиттерінің белгілеу жүйесін сүйемелдейді. Оған қоса, онда іздеу мүмкіндіктерінің толып жатқан түрлері бар. Бұл егер сізге басқа тілде сипатталған шаблонды салыстыру керек болып қалғанда, өте пайдалы әрекет болып саналады. Егер сізге мұнда сипатталған тақырыптардан тыс тіл құралдары қажет болып жатса, оларды өзіңіз іздеңіз. Бірақ есіңізде болсын, барынша көп қасиеттер санын пайдалану – бұл сапалы программалаудың белгісі болып табылмайды. Кез келген мүмкіндікте сіздің программаңызды пайдаланатын қолданушы жайлы да ойланыңыз (оның орнында бірнеше айдан соң, өзіңіз де болып қаларсыз). Қолданушы сіздің кодыңызды оқып түсінуге тырысады: кодты себепсіз күрделі етіп құрмау керек және оның түсініксіз жерлері болмағаны дұрыс.

23.8.1 Символдар және арнайы символдар

Регулярлық өрнектер сөз тіркестерінің символдарын салыстыру үшін қолдануға болатын шаблонды анықтайды. Келісім бойынша, шаблондағы символ, тіркестегі өзіне өзі сәйкес келеді. Мысалы, "`abc`" регулярлық өрнегі (шаблоны) мынадай "`Is there an abc here?`" сөз тіркесінің ішкі "`abc`" тіркесіне сәйкес келеді.

Регулярлық өрнектердің нақты қуаты шаблондағы ерекше мағынасы бар арнайы символдарда және символдар араласуында болады.

Арнайы символдар	
`.`	Кез келген жеке символ
`[`	Символдар класы
`{`	Санауыш
`(`	Топтың басы
`)`	Топтың соңы
`\`	Келесі символдың ерекше мағынасы бар
`*`	Ештеңе жоқ, бір немесе одан көп
`+`	Бір немесе одан көп
`?`	Міндетті емес (ештеңе жоқ немесе бір)
`|`	Балама ((альтернатива) – НЕМЕСЕ)
`^`	Жолдың басы; терістеу
`$`	Жол соңы

Мысалы, мынадай өрнек:

```
x.y
```

`x` әрпінен басталып, `y` әрпімен аяқталатын кез келген үш символдан тұратын тіркеске сәйкес келеді, мысалы, мынадай `xxy`, `x3y` және `xay` тіркестерге дұрыс, бірақ мыналардай `yxy`, `3xy` және `xy` тіркестерге сәйкес емес.

Мынадай өрнектер `{...}`, `*`, `+` және `?` постфикстік операторлар болып табылады. Мысалы, `\d+` өрнегі "бір немесе бірнеше ондық цифрлар" дегенді білдіреді.

Егер сіз шаблонда арнайы символдардың біреуін пайдаланғыңыз келсе, сіз оның алдына кері қиғаш сызық қойып, оны басқарушы символ етуіңіз керек; мысалы, шаблондағы `+` символы "бір немесе бірнеше символ" деген оператор болады, ал `\+` – ол тек `+` таңбасы болып табылады.

23.8.2 Символдар класы

Ең көп таралған символдар тізбегі (комбинациясы) қысылған түрде "арнайы символдар" болып бейнеленген.

Символдар класы үшін арнайы символдар		
`\d`	Ондық цифр	`[[:digit:]]`
`\l`	Төменгі регистрдегі символ	`[[:lower:]]`
`\s`	Ажырату символы (бос орын, табуляция белгісі, т.с.с.)	`[[:space:]]`
`\u`	Жоғарғы регистрдегі символ	`[[:upper:]]`
`\w`	(`a-z` немесе `A-Z`) әріптері, немесе (`0-9`) цифрлары, немесе астын сызу белгісі (`_`)	`[[:alnum:]]`
`\D`	`\d` емес	`[^[:digit:]]`
`\L`	`\l` емес	`[^[:lower:]]`
`\S`	`\s` емес	`[^[:space:]]`
`\U`	`\u` емес	`[^[:upper:]]`
`\W`	`\w` емес	`[^[:alnum:]]`

Жоғарғы регистрдегі символдар "төменгі регистрдегі арнайы символ нұсқасы емес" дегенді білдіреді. Мысалы, `\W` символы "жоғарғы регистр әрпі" дегенді емес, жай "әріп емес" дегенді білдіреді.

Үшінші бағана элементтері (мысалы, `[[:digit:]]`) ұзынырақ атауларды пайдаланатын баламалы (альтернативті) синтаксистік конструкцияларды көрсетеді.

`string` және `iostream` кітапханалары тәрізді `regex` кітапханасы `Unicode` сияқты символдардың үлкен топтарын өңдей алады. `string` және `iostream` кітапханаларындағы сияқты оқырмандар қажет болған кезінде өздері ақпарат іздеп табуы керек екенін естеріңізге салып отырмыз. Мәтіндерді `Unicode` ортасында өңдеу әрекеттерін талқылау бұл кітап аясынан тысқары жатыр.

23.8.3 Қайталаулар

Қайталанатын шаблондар постфикстік операторлармен беріледі.

Қайталау	
`{n}`	Дәлме-дәл exactly `n` рет
`{n,}`	`n` немесе көп есе
`{n,m}`	`n` еседен аз емес және `m` еседен көп емес
`*`	ештеңе жоқ, бір немесе бірнеше , яғни `{0,}`
`+`	бір немесе көп, яғни `{1,}`
`?`	міндетті емес (ештеңе жоқ немесе бір), яғни `{0,1}`

Мысалы, төмендегі өрнек:

```
Ax*
```

артында ешқандай символ жоқ немесе бірнеше `x` символдары қатар келген тіркестерді көрсетеді:

```
A
Axx
Axxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Егер біз `x` символы, кем дегенде бір рет кездесетін болсын десек, онда `*` емес, `+` операторын қолдануымыз керек. Мысалы, мынадай өрнек:

```
Ax+
```

артынан бір немесе бірнеше `x` символы қатарласа келетін `A` символына сәйкес келеді:

```
Ax
Axx
Axxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

бірақ тек

```
A
```

емес.

Жалпы жағдайда, міндетті емес символ сұрақ белгісінің көмегімен көрсетіледі. Мысалы, мынадай өрнек:

```
\d-?\d
```

ортасында міндетті емес дефисі бар екі цифрға сәйкес келеді:

```
1-2
12
```

бірақ мынадай емес:

```
1--2
```

Тіркестерге таңбалардың нақты кіру санын беру үшін немесе нақты диапазонның кіруін көрсету үшін жүйелі жақшалар қолданылады. Мысалы, келесі өрнек:

```
\w{2}-\d{4,5}
```

құрамында екі әріп және дефисі бар, олардың артынан төрт немесе бес цифр жалғасатын тіркестерге сәйкес келеді:

```
Ab-1234
XX-54321
22-54321
```

бірақ мынадай емес

```
Ab-123
?b-1234
```

Сонымен, цифрлар `\w` символдарымен беріледі екен.

23.8.4 Топтау

Кез келген бір регулярлық өрнек *жартылай шаблон* (sub-pattern – ішкі шаблон, шаблон бөлігі) екенін көрсету үшін оны жай жақшаларға алу керек. Мысалы:

```
(\d*:)
```

Бұл өрнек бір де бір цифры жоқ немесе соңында қос нүктесі бар бірнеше цифрдан тұратын жартылай шаблонды анықтайды. Мұнда топты күрделірек шаблонның бөлігі ретінде пайдалануға болады. Мысал қарастырайық.

```
(\d*:)?(\d+)
```

Бұл өрнек міндетті емес бос цифрлар тізбегінен тұрады, одан кейін қоснүкте және бір немесе бірнеше цифрлар жалғаса орналасады. Осындай шаблонды ықшам әрі дәл өрнектеу тәсілін қарапайым адамдар ойлап тапты!

23.8.5 Нұсқалар

"Немесе" (`|`) символы балама (альтернатива) мүмкіндікті береді. Мысал қарастырайық:

```
Subject: (FW:|Re:)?(.*)
```

Бұл өрнек электрондық поштаның хабарламасы тақырыбын анықтайды, оның құрамында болуы міндетті емес `FW:` немесе `Re:` символдары болады, олардан кейін ешнәрсе болмауы немесе бірнеше символ болуы мүмкін. Мысал қарастырайық:

```
Subject: FW: Hello, world!
Subject: Re:
Subject: Norwegian Blue
```

бірақ мынадай болмайды:

```
SUBJECT: Re: Parrots
Subject  FW: No subject!
```

бос балама беруге болмайды:

```
(|def) // error
```

бірақ біз бірнеше баламаны қатар көрсете аламыз:

```
(bs|Bs|bS|BS).
```

23.8.6 Символдар жиыны және диапазондар

Арнайы символдар ең кең тараған символдар кластарын белгілеу үшін қолданылады, олар: цифрлар (`\d`); әріптер, цифрлар және астын сызу таңбасы (`\w`); және т.б. (23.7.2 бөлімді қ.). Дегенмен көбінесе өз арнайы символыңды анықтау пайдалы болады. Оны істеу қиын емес. Мысал қарастырайық:

`[\w @]`	Сөз құраушы символ, бос орын немесе `@`
`[a-z]`	`a`-дан `z`-ке дейінгі төменгі регистр символдары
`[a-zA -Z]`	`a`-дан `z`-ке дейінгі жоғарғы немесе төменгі регистр символдары
`[Pp]`	Жоғарғы немесе төменгі регистрдегі `P` символы
`[\w\-]`	Сөз құраушы символ немесе дефис (жеке алынған `-` диапазонды береді)
`[asdfghjkl;']`	QWERTY пернетақтасының ортаңғы қатарының символдары
`[.]`	Нүкте
`[.[{(\\*+?^$]`	Регулярлық өрнектегі арнайы символ

Символдар класы спецификациясында дефис (`-`) диапазонды көрсету үшін қолданылады, мысалы, `[1-3] (1, 2` немесе `3)` және `[w-z] (w, x, y` немесе `z`). Осындай диапазондарды қолдану кезінде абай болыңыз: барлық тілдердегі әріптер бірдей емес, олардағы алфавит реттіліктер де әртүрлі болуы мүмкін. Егер сізге ағылшын тілінде қабылданған әріптер мен цифрлардың ішкі диапазонына кірмейтін диапазон қажет болса, онда құжаттамаға назар аударыңыз.

Символдар класы спецификациясында біз мына `\w` ("кез келген сөз құраушы символ" дегенді білдіреді) сияқты арнайы символдарды пайдаланатындығымызды айта кету керек. Бізге қалай кері қиғаш сызықты (`\`) символдар класына кіріктіруіміз керек? Әдеттегідей, оны басқару символына: `\\` айналдырамыз.

Егер символдар класы спецификациясындағы бірінші символ `^` болса, ол теріс-теуді `^` білдіреді. Мысалы:

`[^aeiouy]`	Ағылшынша емес дауысты әріп
`[^\d]`	Цифр емес
`[ ^aeiouy]`	Ағылшынша дауысты әріп немесе `^` символы

Соңғы регулярлық өрнектегі `^` символы тік жақшадан (`[`) кейінгі бірінші орында тұрған жоқ, демек, бұл терістеу операциясы емес, қарапайым символ. Регулярлық өрнектер өте қитұрқы бола алады.

`regex` кітапханасын жүзеге асыру, салыстыру үшін қолданылатын символдар класының атаулы жиынынан тұрады. Мысалы, егер сіз әріптік-цифрлық символдарды (яғни әріптер мен цифрлар: `a-z`, немесе `A-Z`, немесе `0-9`) салыстырғыңыз келсе, онда мұны `[[:alnum:]]` регулярлық өрнегі арқылы істеуге болады. Мұнда `alnum` сөзі символдар жиынының (әріптік-цифрлық символдар жиыны) аты болып табылады. Тік жақшаға алынған әріптік-цифрлық символдардың бос тіркесі үшін жасалған шаблон мынадай: `"[[:alnum:]]+"` бола алады.

Осы регулярлық өрнекті сөз тіркесі ішіне орналастыру үшін, біз қос тырнақшаны басқарушы символ етіп жіберуіміз керек:

```
string s = "\"[[:alnum:]]+\"";
```

Бұған қоса, **string** класының объектісін **regex** класының объектісі ішіне орналастыру үшін біз тек қос тырнақшаны ғана емес, кері қиғаш сызықты да басқарушы символ етіп, инициалдау үшін жай жақшаларды қолдануымыз керек, өйткені **regex** класының конструкторы тікелей түрде былай болады:

```
regex s("\\\"[[:alnum:]]+\\\"");
```

Регулярлық өрнектерді пайдалану көптеген белгілеулерді енгізуге мәжбүр етеді. Символдардың стандартты кластарын тізіп көрсетейік.

Символдар класы	
alnum	Кез келген әріптік-цифрлық символ
alpha	Кез келген әріптік-цифрлық символ
blank	Жолдарды бөлмейтін кез келген ажыратқыш символ
cntrl	Кез келген басқару символы
d	Кез келген ондық цифр
digit	Кез келген ондық цифр
graph	Кез келген графикалық символ
lower	Төменгі регистрдегі кез келген символ
print	Кез келген баспа символы
punct	Кез келген пунктуация символы
s	Кез келген ажыратқыш символ
space	Кез келген ажыратқыш символ
upper	Жоғарғы регистрдегі кез келген символ
w	Кез келген сөз құраушы символ (әріптік-цифрлық символ және астын сызу таңбасы)
xdigit	Кез келген оналтылық цифрлық символ

regex кітапханасының жүзеге асырылуында символдардың басқа кластары да болуы мүмкін, бірақ егер сіз мұндағы тізімде көрсетілмеген атаулы класты пайдалануға бет бұрсаңыз, онда оның программа ауысымдылығын әлсіретпейтініне көз жеткізіңіз.

23.8.7 Регулярлық өрнектердегі қателер

Егер біз регулярлық өрнекті дұрыс етіп бермесек, не болады? Мысал қарастырайық:

```
regex pat1("(|ghi)");  // балама операторы өтіп кеткен
regex pat2("[c-a]");   // диапазон емес
```

Біз шаблонды **regex** класының объектісіне меншіктегенімізде, ол тексеріледі. Егер регулярлық өрнектерді салыстыру механизмі ол дұрыс болмағандықтан немесе өте күрделі болғандықтан, жұмыс істей алмаса, **bad_expression** аласта-масы пайда болады (генерацияланады).

Регулярлық өрнектерді салыстыру механизмін зерттеуге мүмкіндік беретін шағын программа қарастырайық.

```
#include <boost/regex.hpp>
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
using namespace std;
using namespace boost;
// егер сіз boost кітапханасының іске асырылуын қолдансаңыз

// сырттан шаблон мен жолдар жиынын аламыз
// шаблонды тексереміз де,
// сол шаблон кіретін жолдарды іздейміз

int main()
{
    regex pattern;

    string pat;
    cout << "шаблон енгізіңіз: ";
    getline(cin,pat);      // шаблонды оқимыз

    try {
          pattern = pat;  // шаблонды тексеру
          cout << "pattern: " << pattern << '\n';
    }
    catch (bad_expression) {
        cout << pat << " дұрыс регулярлық өрнек емес\n";
        exit(1);
    }

    cout << "now enter lines:\n";
    string line;            // кіріс буфері
    int lineno = 0;
```

```
    while (getline(cin,line)) {
        ++lineno;
        smatch matches;

        if (regex_search(line, matches, pattern)) {
          cout << "жол " << lineno << ": " << line << '\n';
          for (int i = 0; i<matches.size(); ++i)
               cout << "\tmatches[" << i << "]: "
                    << matches[i] << '\n';
        }
        else
           cout << "сәйкес келмейді\n";
    }
}
```

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Осы программаны іске қосыңыз да, оны бірнеше шаблонды тексеру үшін қолданып көріңіз, мысалы, `abc`, `x.*x`, `(.*),\([^)]*\)`, and `\w+ \w+(Jr\.)?`.

23.9 Регулярлық өрнектерді салыстыру

Регулярлық өрнектер, негізінде, екі жағдайда пайдаланылады.

- Мәліметтер ағынында (кез келген ұзын) регулярлық өрнекке сәйкес сөз тіркесін *іздеу* – `regex_search()` функциясы бұл шаблонды ағындағы ішкі тіркес түрінде іздейді.
- Регулярлық өрнектті сөз тіркесімен (берілген көлемдегі) *салыстыру* – `regex_match( )` функциясы шаблон мен сөз тіркесінің толық сәйкестігін іздейді.

Мұның бір мысалы 23.6 бөліміндегі пошта индекстерін іздеу болып табылады. Келесі кестеден мәліметтер алуды қарастырайық.

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER IALT
0A	12	11	23
1A	7	8	15
1B	4	11	15
2A	10	13	23
3A	10	12	22
4A	7	7	14
4B	10	5	15

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER IALT
5A	19	8	27
6A	10	9	19
6B	9	10	19
7A	7	19	26
7	3	5	8
7I	7	3	10
8A	10	16	26
9A	12	15	27
0MO	3	2	5
0P1	1	1	2
0P2	0	5	5
10B	4	4	8
10CE	0	1	1
1MO	8	5	13
2CE	8	5	13
3DCE	3	3	6
4MO	4	1	5
6CE	3	4	7
8CE	4	4	8
9CE	4	9	13
REST	5	6	11
Alle klasser	234	202	386

Бұл өте қарапайым және онша күрделі емес кесте (2007 жылы Бьярне Страуструп орта мектебіндегі оқушылар саны) веб-парақтан алынды, онда ол осындай біз пайдаланған күйде тұрған болатын:

- Оның сандық өрістері бар.
- Кесте жолдарынан алынған мәтінді білетін адамдарға ғана түсінікті символдық өрістер бар. (Мұнда кесте мәліметтерін тек дат тілін білетін адамдар ғана түсінеді).
- Символдық тіркестерде бос орындар бар.
- Өрістер бір-бірінен ажыратқыш символдармен бөлінген, олардың рөлін мұнда табуляция таңбасы атқарып тұр.

Мұны өте қарапайым және онша күрделі емес кесте деп атадық, бірақ онда көрінбей тұрған бір кілтипан бар: негізінде, біз бос орындар мен табуляция таңбаларын ажырата алмаймыз; ол мәселені оқырмандар өздері шешуі тиіс.

Келесі есептерді шығару үшін регулярлық өрнектерді қалай пайдалануға болатынын көрсетейік.

- Кестенің дұрыс құрылғанына көз жеткізу керек (яғни әрбір қатардағы өрістер саны дұрыс алынған).
- Қосындылары дұрыс есептелгеніне сенімді болуымыз қажет (соңғы жолда бағаналардағы сандар қосындысы көрсетілген).

Егер біз мұны жасай алсақ, онда бәрін де жасай аламыз! Мысалы, біз қатарларының алғашқы символдары бірдей (мысалы, жылдары: бірінші класс нөмірі 1 болуы тиіс) болып келген жолдары біріктірілген немесе жылдар өткен сайын студенттер саны (10-11 жаттығуды қ.) өсе ме, кеми ме, соны тексеретін жаңа кесте жасай алар едік.

Мұндай кестені талдау үшін бізге тақырып үшін және қалған жолдар үшін қолданылатын екі шаблон керек болады:

```
regex header( "^[\\w ]+(      [\\w ]+)*$");
regex row( "^[\\w ]+(         \\d+)(  \\d+)(  \\d+)$");
```

Естеріңізде болса, біз ықшамдылығы мен пайдалылығы үшін, регулярлық өрнектерді мақтаған болатынбыз, бірақ оған жаңадан үйренушілерге игеру жеңілдігі кірмейді. Негізінде, регулярлық өрнектердің *тек хаттар үшін* (write-only language) деген қосалқы аты да бар. Тақырыптан бастайық. Онда ешқандай сандық мәліметтер болмайтындықтан, біз бірінші жолды алып тастауымызға да болар еді, бірақ тек қана тәжірибе алу үшін – оны құрылымдық талдаудан өткізейік. Онда төрт сөздік өрістер (табуляция таңбасымен бөлінген әріптік-цифрлық өрістер) бар. Бұл өрістерде бос орындар болуы мүмкін, сондықтан бұл символдарды беру үшін жай ғана `\w` басқару символын пайдаланумен шектеле алмаймыз. Оның орнына біз `[\w ]` өрнегін қолданамыз, яғни сөз құраушы символ (әріп, цифр немесе астын сызу белгісі) немесе бос орын. Бір немесе бірнеше сөз құраушы символ `[\w ]+` өрнегімен беріледі. Біз жол басында тұрған символды тапқымыз келіп отыр, сондықтан `^([\w ]+)` өрнегін жазамыз. "Құстұмсық" (`^`) жол басын білдіреді. Қалған өрістердің әрқайсысын табуляция белгісіне жалғасқан кейбір сөздермен (`[\w ]+`) көрсетуге болады. Жол соңына дейін бұның `( [\w ]+)*$` кез келген саны болуы мүмкін. Доллар белгісі (`$`) "жол соңын" білдіреді. Енді C++ тілінде тіркестік литерал жазамыз да, қосымша кері қиғаш сызықтар аламыз.

```
"^[\\w ]+(    [\\w ]+)*$"
```

Біз табуляция белгісін дәл анықтай алмаймыз, бірақ мұндағы жағдайда мәтін теру барысында ол өзі айқындалып белгілі болады.

Жаттығудың ең қызықты бөлігіне кірісейік: сандық мәліметтер алғымыз келетін жолдарға арналған шаблонға қарайық. Бірінші өрістің қайтадан `^[\w]+` шаблоны бар. Одан кейін әрқайсысының алдарында табуляция таңбасы бар `( \d+)` үш сандық өріс жалғасады, сондықтан келесі шаблон аламыз:

```
^[\w ]+(    \d+)(  \d+)(  \d+)$
```

Мұны тіркестік литералға енгізгеннен кейін ол мынадай болады:

```
"^[\\w ]+(  \\d+)(    \\d+)(  \\d+)$"
```

Енді біз талап етілген әрекеттердің бәрін де орындап шықтық. Алдымен кестенің дұрыс қалыптасқанын тексерейік:

```
int main()
{
    ifstream in("table.txt");              // кіріс файлы
    if (!in) error("no input file\n");

    string line;                           //  енгізу буфері
    int lineno = 0;

    regex header( "^[\\w ]+(  [\\w ]+)*$");  //тақырып жолы
    regex row( "^[\\w ]+(   \\d+)(  \\d+)(   \\d+)$");
    // мәліметтер жолы

    if (getline(in,line)) {      // тақырып жолын тексеру
            smatch matches;
            if (!regex_match(line, matches, header))
                    error("no header");
    }
    while (getline(in,line)) {   // мәліметтер жолын тексеру
            ++lineno;
            smatch matches;
            if (!regex_match(line, matches, row))
                    error("қате жол",to_string(lineno));
    }
}
```

Қысқа болуы үшін біз мұнда **#include** директиваларын келтірмедік. Әрбір жолдағы барлық символдарды тексереміз, сондықтан **regex_search** функциясын емес, **regex_match** функциясын шақырамыз. Бұлардың арасындағы айырмашылық **regex_match** функциясы енгізу ағынындағы әрбір символды шаблонмен салыстыруы керек, ал **regex_search** функциясы енгізу ағынын оның ішіндегі сәйкес ішкі тіркесті табу үшін тексереді. **regex_search** функциясын пайдалану керек болған кездерде, қателесіп **regex_match** функциясын пайдалану (және керісінше) ең қиын анықталатын қате болып табылады. Бірақ осы екі функция да өздерінің бірдей аргументтерін бірдей жағдайда пайдаланады.

Енді кестедегі мәліметтерді тексеруге көше аламыз. Біз мектепте оқитын ұлдар ("drenge") мен қыздардың ("piger") санын анықтаймыз. Әрбір жолдың соңғы өрісінде ("ELEVER IALT") оның алдыңғы екі өрісінің нақты қосындысы жазылғандығын тексеріп шығамыз. Соңғы жолда ("Alle klasser") бағаналар бойынша қосынды нәтижелер орналасады. Мұны тексеру үшін, мәтіндік өрісте аздап сәйкестік болатындай түрде және "Alle klasser" тіркесін анықтайтындай етіп, **row** өрнегін толықтырып өзгертеміз:

```
int main()
{
    ifstream in("table.txt");        //  кіріс файлы
    if (!in) error("кіріс файлы жоқ");

    string line;                     //  енгізу буфері
    int lineno = 0;

    regex header( "^[\\w ]+(    [\\w ]+)*$");
    regex row( "^([\\w ]+)(   \\d+)(   \\d+)(    \d+)$");

    if (getline(in,line)) {   // тақырып жолын тексереміз
        boost::smatch matches;
        if (!boost::regex_match(line, matches, header)) {
              error("no header");
        }
    }

    // бағаналар бойынша қосындылар:
    int boys = 0;
    int girls = 0;

    while (getline(in,line)) {
        ++lineno;
        smatch matches;
        if (!regex_match(line, matches, row))
            cerr << "bad line: " <<    lineno << '\n';

        if (in.eof()) cout << "at eof\n";

        // жолдарды тексереміз:
        int curr_boy = from_string<int>(matches[2]);
        int curr_girl = from_string<int>(matches[3]);
        int curr_total = from_string<int>(matches[4]);
        if (curr_boy+curr_girl != curr_total)
            error("bad row sum \n");

        if (matches[1]==" lle klasser") {   //соңғы жол
            if (curr_boy != boys)
            error("ұлдар саны дұрыс емес\n");
            if (curr_girl != girls)
            error("қыздар саны дұрыс емес\n");
            if (!(in>>ws).eof())
            error("қорытынды жолдан кейінгі символдар");
           return 0;       }
```

```
            // update totals:
            boys += curr_boy;
            girls += curr_girl;
        }

        error("қорытынды жол жоқ");
    }
```

Соңғы жол мағынасы бойынша қалғандарынан өзгеше болып отыр: онда қосындылар орналасады. Біз оны ("Alle klasser") белгісі бойынша анықтаймыз. Біз соңғы символдан кейін ажыратқыш белгі (мұны анықтау үшін `lexical_cast` класынан алынған әдіс пайдаланылады (23.2 бөлімді қ.) бола алмайтын символ тұрмауы тиіс деп шешкен болатынбыз, әйтпесе қате кеткені туралы хабарлама береміз.

Мәліметтер өрістерінен сандарды шығарып алу үшін біз 23.2 бөліміндегі `from_string` функциясын пайдаландық. Біз бұл өрістерде тек цифрлар ғана тұратынын тексеріп шыққанбыз, сондықтан `string` класының объектісін `int` типіндегі айнымалыға түрлендірудің дұрыстығын тексеру қажет емес.

23.10 Сілтемелер

Регулярлық өрнектер – көптеген программалау тілдерінде және әртүрлі форматтарда қол жеткізуге болатын кең таралған пайдалы құрал. Олар формальды тілдерге негізделген ықшам теориясымен және шекті автоматтарға негізделген тиімді технологиялар арқылы іске асырылуымен сүйемелденеді. Регулярлық өрнектерді сипаттау, олардың теориясы, шекті автоматтарды жүзеге асыру мен пайдалану бұл кітапта қарастыру аясынан тыс жатыр. Дегенмен, бұл тақырып компьютерлік ғылымдарда стандарт болып табылатындықтан, ал регулярлық өрнектер өте кең таралып кеткендіктен, қажет болып жатса, керекті ақпаратты тауып алу қиынға түспейді. Солардың бірсыпырасын көрсете кетейік:

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and J effrey D. Ullman. Compilers: *Principles, Techniques, and Tools, Second Edition* (көбінесе "The Dragon Book" деп аталатын). Addison-Wesley, 2007. ISBN 0321547985.

Austern, Matt, ed. "Draft Technical Report on C++ Library Extensions." ISO/ IEC DTR 19768, 2005. *www.open-std.org/jtc1/sc22/wg21/docs/papers/ 2005/n1836.pdf.*

Boost.org. A repository for libraries meant to work well with the C++ тілінің стандартты кітапханасымен үйлестірілген кітапханалар қоймасы. *www.boost.org.*

Cox, Russ. "Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, . . .)." *http://swtch.com/~rsc/regexp/regexp1.html.*

Maddoc,J . boost::regex documentation. *www.boost.org/libs/regex/doc/index.html.*

Schwartz, Randal L., Tom Phoenix, and Brian D. Foy. *Learning Perl, Fourth Edition.* O'Reilly, 2005. ISBN 0596101058.

ТАПСЫРМА

1. `regex` кітапханасы сіздің стандартты кітапханаңыздың бөлігі болып табылатындығын анықтаңыз. Көмек: `std::regex` және `tr1::regex` іздеу керек.
2. 23.7.7 бөліміндегі шағын программаны іске қосыңыз; ол үшін сіздің компьютеріңізге `boost::regex` кітапханасын орнатып алу керек (егер оны бұрын орнатпасаңыз), сонан соң `regex` кітапханасымен байланыс орнату үшін жобаның немесе командалық жолдың опцияларын баптап алып, `regex` тақырыптарын пайдалану қажет.
3. 23.7 бөліміндегі шаблондарды тексеру үшін 1-тапсырманың программасын пайдаланыңыз.

БАҚЫЛАУ СҰРАҚТАРЫ

1. "text" тіркесін біз қайдан табамыз?
2. Мәтінді талдау үшін стандартты кітапхананың қандай мүмкіндіктері жиі қолданылады?
3. `insert()` функциясы элементті қайда қояды – көрсетілген позицияға (немесе итераторға) дейін бе, әлде одан кейін бе?
4. `Unicode` дегенімз не?
5. Типті `string` класына және керісінше қалай түрлендіреміз?
6. Егер `s` – `string` класының объектісі болса, онда `cin>>s` нұсқауы мен `getline(cin,s)` функциясын шақырудың арасында қандай айырмашылық бар?
7. Стандартты ағындарды тізіп көрсетіңіз.
8. `map` ассоциативті жиымының кілті не үшін керек? Кілттер үшін пайдалы болып табылатын типтерге мысал келтіріңіз.
9. `map` класы контейнерлері бойынша қалай жылжуға болады?
10. `map` және `multimap` кластары арасындағы айырмашылық неде? `multimap` класында, `map` класында бар, қандай пайдалы операция жоқ, неге?
11. Бір бағытты итератор үшін қандай операциялар қажет етіледі?
12. Бос өріс пен жоқ өрістің айырмашылығы неде? Екі мысал келтіріңіз.
13. Регулярлық өрнектерді қалыптастыру кезінде басқару тізбегінің символы не үшін керек?
14. Регулярлық өрнектерді `regex` типіндегі айнымалыға қалай айналдыруға болады?
15. `\w+\s\d{4}` шаблонына қандай тіркестер сәйкес келеді? Үш мысал кел-

тіріңіз. Берілген шаблоны бар `regex` типіндегі айнымалыны инициалдау үшін қандай тіркестік литералды пайдалану керек?

16. Сөз тіркесі (программадағы) дұрыс регулярлық өрнек болатындығын қалай анықтауға болады?
17. `regex_search()` функциясы не істейді?
18. `regex_match()` функциясы не істейді?
19. Регулярлық өрнектегі нүкте символын қалай бейнелеуге болады?
20. Регулярлық өрнектегі "үштен кіші емес" түсінігін қалай бейнелеуге болады?
21. `7` символы `\w` тобына жата ма? Ал `_` (астын сызу) символы ше?
22. Жоғарғы регистрдегі символдар үшін қандай белгі қолданылады?
23. Өзіңіз таңдап алған символдар жиынын қалай беруге болады?
24. Бүтін сандық өрістен мәнді қалай шығарып алуға болады?
25. Регулярлық өрнек арқылы жылжымалы нүктелі санды қалай бейнелеуге болады?
26. Шаблонға сәйкес тіркестен жылжымалы нүктелі санды қалай шығарып алуға болады?
27. Жартылай *сәйкес келу* (sub-match) деген не? Оны қалай анықтауға болады?

ТЕРМИНДЕР

`multimap`	`smatch`	шаблон
`regex_match()`	регулярлық өрнек	шаблон бөлігі
`regex_search()`	сәйкес келу (match)	іздеу

ЖАТТЫҒУЛАР

1. Электрондық пошта хабарламалары бар файлмен жұмыс істейтін программаны іске қосыңыз; оны одан көлемдірек өз файлыңызды пайдаланып тесттен өткізіңіз. Осы файлда қате туралы хабарламалар шығаратын хабарламалар бар, мысалы, екі адрестік жолдары бар хабарламалар, адрестері және/немесе тақырыптары бірдей бірнеше хабарламалар, тағы бос хабарламалар. Одан басқа, программаны жалпы хабарлама болмайтын және программалық спецификацияға сәйкес келмейтін мысал арқылы тесттен өткізіңіз, мысалы, жолдары жоқ файл арқылы...
2. `multimap` класын қосып, оған хабарлама тақырыптарын енгізіңіз. Программа сөз тіркестерін пернетақтадан енгізіп, тақырыбы берілген тіркеспен бірдей болатын әрбір хабарламаны шығаратын болсын.

3. 23.4 бөліміндегі мысалды өзгертіп, электрондық пошта хабарламасының тақырыбы мен оны жіберушіні анықтау үшін регулярлық өрнектерді қолданыңыз.
4. Электрондық пошта хабарламалары бар файл (шынайы хабарламалары бар файл) тауып, қолданушының пернетақтадан енгізген хабарлама жіберушілер аттары арқылы тақырыптарды анықтай алатындай етіп, программаны өзгертіңіз.
5. Электрондық пошта хабарламалары бар көлемді файл (мыңдаған хабарламалары бар) тауып, оны **`multimap`** және **`unordered_multimap`** кластарының объектілеріне жазыңыз. Біздің қосымшада **`multimap`** класы объектісінің реттелген артықшылығы пайдаланылмайтындығына назар аударыңыз.
6. Мәтіндік файлдағы күн-ай мерзімін (датаны) анықтайтын программа жазыңыз. Онда кем дегенде, **`line-number: line`** форматындағы бір датасы бар әрбір жолды баспаға шығарыңыз. Қарапайым формат үшін регулярлық өрнектен бастаңыз, мысалы, 12/24/2000, программаны осы мысал арқылы тесттен өткізіңіз. Сонан соң жаңа форматтарды қосыңыз.
7. Файлдағы несиелік кәртішкелер нөмірлерін табатын программа (алдыңғыға ұқсайтын) жазыңыз. Несиелік кәртішкелер нөмірлерін жазу үшін қандай форматтар қолданылатынын анықтап талдаңыз да, программаңызда осыларды тексеруді іске асырыңыз.
8. 23.8.7 бөліміндегі программаны оның кірісіне шаблон мен файл атын енгізетін етіп түрлендіріңдер. Программа нәтижесі шаблонға сәйкес нөмірленген жолдар (**`line-number: line`**) болуы тиіс. Егер сәйкестік анықталмаса, ешнәрсе шығару қажет емес.
9. **`eof()`** (B.7.2 бөлім) функциясы арқылы кестедегі қай жолдың ең соңғы жол екенін анықтауға болады. Программаны қарапайым ету үшін, кестені талдайтын (23.9 бөлім) осы функцияны пайдаланыңыз. Құрған программаңызды кестеден кейін бос жолы бар файлдар және де жаңа жолға көшумен аяқталмайтын файлдар үшін тексеріп шығыңыз.
10. 23.9 бөліміндегі кестені тексеретін программаны бірінші цифрлары (жылды көрсетеді: бірінші класқа 1 саны сәйкес келеді) бірдей болып келген жолдарды біріктіретін жаңа кесте шығаратындай етіп өзгертіңіз.
11. 23.9 бөліміндегі кестені тексеретін программаны жылдар өткен сайын оқушылар саны өсуде ме, әлде кемуде ме, соны тексеретіндей етіп өзгертіңіз.
12. Күн-ай мерзімі бар жолдарды анықтайтын программаны (6-жаттығу) негізге ала отырып, барлық мерзімдерді анықтап тауып, соларды ISO форматына (жыл/ай/күн) түрлендіріңіз. Бұл программа ақпаратты кіріс файлынан оқып, сол файлға ұқсас нәтижелік (шығыс) файлға жазуы керек, мұнда тек мерзімдер (даталар) басқа форматта жазылып шығуы тиіс.

13. Нүкте (`.`) `'\n'` шаблонына сәйкес келе ме? Осы сұраққа жауап беретін программа жазыңыз.
14. 23.8.7 бөліміндегі программа тәрізді сырттан енгізілген шаблондарды салыстыру үшін тәжірибе (эксперимент) жасауға болатын программа жазу керек. Бірақ енді программа мәтіндерді жолдарға бөлетін шаблондармен тәжірибе жасау үшін, мәліметтерді файлдан оқып, оны компьютер жадына жазуы тиіс (жолдарға бөлу жаңа жолға көшу `'\n'` символы бойынша орындалады). Программаны бірнеше ондаған шаблондармен тексеріп, тесттен өткізіңіз.
15. Регулярлық өрнек арқылы бейнелеуге болмайтын шаблонды сипаттаңыз.
16. Тек сарапшылар (эксперттер) үшін: алдыңғы жаттығудағы шаблон регулярлық өрнек болып табылмайтынын дәлелдеңіз.

СОҢҒЫ СӨЗ

Компьютерлер мен есептеулер тек қана сандарға қатысты деп, ал есептеулер математика бөлігі деп санап, жаңылысып кетуге болады. Әрине, олай емес. Жай ғана компьютер экранына қараңызшы; ол мәтінмен және пикселдермен толтырылған. Мүмкін, сіздің компьютеріңіз әуендерді де ойнайтын болар. Әрбір қосымша программа үшін дұрыс құрал таңдай білу маңызды болып табылады. C++ тілі үшін бұл керекті кітапхананы дұрыс таңдай білу дегенді білдіреді. Мәтіндермен жұмыс істеу үшін негізгі құрал болып көбінесе регулярлық өрнектер есептеледі. Оның үстіне `map` ассоциативті контейнерлерлері мен стандартты алгоритмдерді де ұмытуға болмайды.

24

Сандар

Кез келген күрделі мәселенің айқын, қарапайым, оның үстіне қате шешімі болады.

- Г. Л. Менкен (H. L. Mencken)

Бұл тарауда тіл мен оның кітапханасы ұсынатын сандық есептеулерге арналған негізгі құралдарға шолу жасалған. Біз мұнда сандардың мөлшеріне, дәлдігіне және дөңгелектенуіне байланысты іргелі мәселелерді қарастырамыз. Тарауда C тілі стиліндегі көпөлшемді жиымдар мен N-өлшемді матрицаларға басты назар аударылған. Біз тесттен өткізу мен модельдеу және ойындарды программалау үшін жиі қажет болатын кездейсоқ сандарды алу (генерациялау) жолдарын да сипаттаймыз. Қорытындылай келе, комплексті сандармен жұмыс істеуге арналған кітапхананың негізгі функционалдық мүмкіндіктері қысқаша баяндалып, стандартты математикалық функциялар туралы да әңгіме қозғалады.

24.1 Кіріспе

Кейбір адамдар үшін, айталық, көптеген ғалым, инженер және статистиктер үшін байсалды сандық есептеулер негізгі жұмыс болып табылады. Көптеген адамдардың жұмысында сандық есептеулер белгілі рөл атқарады. Осы санатқа, кейде физиктермен жұмыс істейтін, компьютерлік ғылым мамандары жатады. Адамдардың көпшілігінде, бүтін сандармен және ондық нүктесі бар сандармен орындалатын қарапайым арифметикалық амалдар аясынан тысқары сандық есептеулер қажеттілігі сирек туындайды. Бұл тараудың мақсаты – тілдің қарапайым есептерді шығаруға қажетті мүмкіндіктерін сипаттау. Біз оқырмандарды сандық талдауға немесе ондық нүктелі сандармен операциялар орындау нақтылықтарына үйретуге тырыспаймыз, мұндай тақырыптар біздің кітап аясынан тысқары жатыр және олар нақты қосымшалармен тығыз байланыста болады. Мұнда біз келесі тақырыптарды қарастыруға талпынамыз:

- Тұрақты мөлшері бар құрамдас типтерге байланысты сұрақтар, мысалы, сандардың дәлдігі мен толып кетуі.
- C тілі стиліндегі және десандық есептеулерге барынша сәйкес келетін `Matrix` кітапханасынан алынған класс.
- Кездейсоқ сандарға кіріспе.

- Кітапханадағы стандартты математикалық функциялар.
- Комплекс сандар.

Мұнда C тілі стиліндегі көпөлшемді жиымдарға және матрицалармен (көпөлшемді жиымдармен) жұмыс істеуді жеңілдететін **N**-өлшемді **Matrix** матрицалар кітапханасына баса назар аударылады.

24.2 Санның мөлшері, дәлдігі және толып кетуі

Сіз құрамдас типтерді және қалыпты есептеу әдістерін пайдаланған кезіңізде, сандар копьютер жадының бекітілген мөлшері бар аймағында сақталады; басқаша айтқанда, бүтін сандық типтер (**int**, **long** және т.б.) солардың тек шамамен алынған мәндері, ал жылжымалы нүктелі сандар (**float**, **double** және т.б.) сол нақты сандардың жуық шамалары. Осыдан шығатыны, математикалық тұрғыдан алғанда, кейбір есептеулер онша дәл емес немесе дұрыс емес болып табылады. Мысал қарастырайық:

```
float x = 1.0/333;
float sum = 0;
for (int i=0; i<333; ++i)  sum+=x;
cout << setprecision(15) << sum << "\n";
```

Осы программаны орындағанда, біз бірді емес, келесі санды аламыз:

```
0.999999463558197
```

Біз осыған ұқсас нәтиже күткенбіз. Жылжымалы нүктелі сан биттердің бекітілген тұрақты санынан тұрады, сондықтан есептеу кезінде біз оны "бұзып" аламыз. Есептеу нәтижесі аппараттық жабдықтама беретін мәннен артық биттер санынан тұрады. Мсалы, **1/3** рационал санын дәл ондық сан түрінде бейнелеу мүмкін емес (бірақ ондық жіктеу арқылы оның көптеген цифрларын анықтауға болады). Осылайша **1/333** санын да дәл бейнелеу мүмкін емес, сондықтан **x** санының (**1/333** санының **float** типі арқылы машинадағы ең жақсы жуықтауы) **333** көшірмесін қосқан кезде, бірден аздап айырмашылығы бар сан аламыз. Жылжымалы нүктелі сандарды екпінді түрде пайдаланған кезде, оларды дөңгелектеу қатесі туындайды; бізге тек оның нәтижеге қандай деңгейде әсер ететінін бағалау ғана қалады.

Әрқашанда нәтиженің қандай дәлдікпен анықталғанын тексеріп отырыңыз. Есептеу кезінде сіз нәтиженің қандай болатынын шамалауыңыз керек, әйтпесе келеңсіз қатеге немесе есептеу қатесіне тап боласыз. Дөңгелектеу қатесі болатыны есте тұратын болсын, егер күмәндансаңыз, сарапшы кеңесіне жүгініңіз немесе сандық әдістер жайлы оқулықты қарап шығыңыз.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Мысалдағы 333 санын 10 санымен алмастырып, қайтадан есептеп көріңіз. Қандай нәтиже күтуге болатын еді? Қандай нәтиже алдыңыз? Біз ескертіп едік қой!

Мөлшері (ені) бекітілген бүтін сандардың әсері өте елеулі болуы да мүмкін. Жылжымалы нүктелі сандар, анықтама бойынша, нақты сандардың жуық мәндері болып табылады, сондықтан олар дәлдігін жоғалтуы мүмкін (яғни, ең төменгі жақта орналасқан биттерінен айрылады). Басқа жағынан алсақ, бүтін сандар көбінесе ұлғайып толып кетіп те жатады (яғни, ең жоғарғы жақтағы биттерінен айрылады). Осылардың нәтижесінде, жылжымалы нүктелі сандармен байланысты туындайтын қателер салмақтырақ болады (жұмысты жаңа бастағандар оларды байқамайды да), ал бүтін сандармен байланысты қателер бірден көзге түседі (оларды жұмысты жаңа бастағандар да байқамай қалмайды). Біз қателердің ертерек анықталғанын қалаймыз, ондайда оларды түзету де жеңіл болады.

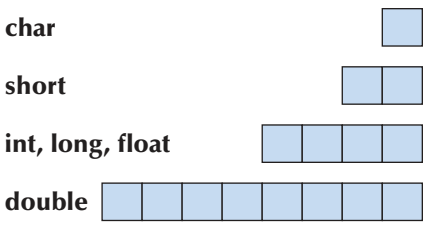
Бүтін сандармен орындалатын есеп қарастырайық:

```
short int y = 40000;
int i = 1000000;
cout << y << "      " << i*i << "\n";
```

Осы программаны орындап, келесі нәтиже аламыз:

```
-25536        - 727379968
```

Мұны күтуге де болатын еді. Мұнда біз разрядтардың толып кету әрекетін көріп отырмыз. Бүтін сандық типтер салыстырмалы түрде алғанда, аса үлкен емес сандарды ғана бейнелей алады. Бізге әрбір бүтін санды дәл бейнелеу үшін биттер жетіспейді, сондықтан есептеулерді тиімді түрде жүргізетін тәсілдер керек болады. Жоғарыдағы жағдайда, `short` типіндегі екі байттық сан 40 000 санын бейнелей алмайды, ал төрт байттық `int` типі 1 000 000 000 санын бейнелей алмайды. C++ тіліндегі құрамдас типтердің дәл мөлшерлері (көлемдері немесе ендері) аппараттық жабдықтама және компилятор жұмысына тәуелді болып келеді, `x` айнымалысының ені немесе `x` типінің байтпен берілген ұзындығын `sizeof(x)` операторы арқылы анықтауға болады. Анықтама бойынша, `sizeof(char)==1`. Мұны келесідей түрде бейнелеп көрсетуге болады:



Мұндай ұзындықтар, яғни типтер мөлшерлері Windows жүйесіне және Microsoft компаниясының компиляторына қатысты көрсетілген. C++ тілінде бүтін сандарды және жылжымалы нүктелі сандарды әртүрлі мөлшерлерді (ұзындықтарды) пайдалана отырып, бейнелеудің көптеген тәсілдері бар, бірақ маңызды себептер болмаса, **char**, **int** және **double** типтерін қолданған дұрыс болып саналады. Көптеген программаларда (әрине, бәрінде емес) бүтін сандардың және жылжымалы нүктелі сандардың қалған типтері біз қалағаннан гөрі әлдеқайда артық проблемалар туғызып жатады.

Бүтін санды жылжымалы нүктелі сан типіндегі айнымалыға меншіктеуге болады. Егер бүтін сан жылжымалы нүктелі сан типі көрсете алатын саннан артық болса, дәлдікті жоғалту орын алады. Мысал қарастырайық:

```
cout<<"sizes:"<<sizeof(int)<<' '<<sizeof(float)<<'\n';
int x = 2100000009;       //  large int
float f = x;
cout << x << ' ' << f << endl;
cout << setprecision(15) << x << ' ' << f << '\n';
```

Біздің копьютерде алынған нәтиже мынадай болды:

```
Sizes: 4 4
2100000009 2.1e+009
2100000009 2100000000
```

float және **int** типтері компьютер жадында бірдей мөлшерде орын алып тұрады (4 байттан). **float** типіндегі сан мантиссадан (көбінесе нөл мен бір аралығындағы сан) және оның дәрежелік көрсеткішінен (яғни, мантисса *10 дәреже көрсеткіші) тұрады, сондықтан ол **int** типіндегі ең үлкен санды дәл бейнелей алмайды. (Егер біз мұны жасауға талпынсақ, компьютер жадында дәреже көрсеткішін орналастырғаннан кейін мантиссаға қажетті орын бөліп бере алмас едік). Өзіміз күткеніміздей, **f** айнымалысы өз дәлдігінің мүмкіндігін барынша пайдаланып, **2100000009** санын көрсетеді. Дегенмен, соңғы **9** цифры өте үлкен қате береді, сол себепті мысал үшін осы санды таңдап алдық.

Басқа жағынан алғанда, біз бүтін сан типіндегі айнымалыға жылжымалы нүктелі санды меншіктесек, оның ені қысқартылады, яғни бөлшек бөлігі – ондық нүктеден кейінгі цифрлар – алынып тасталынады. Мысал қарастырайық:

```
float f = 2.8;
int x = f;
cout << x << ' ' << f << '\n';
```

x айнымалысының мәні **2**-ге тең болады. Сіздер ойлағандай, C++ тілінде "4/5 санын дөңгелектеу ережесін" қолдансақ та, ол **3**-ке тең болмайды. Өйткені C++

тілінде **float** типін **int** типіне түрлендіру дөңгелектеу емес, биттерді қысқарту арқылы орындалады.

Сонымен, есептеулерді орындау кезінде мүмкін болатын толып кету және қысқарту әрекеттерін болдырмауға тырысу керек. C++ тілі бұл мәселені сіз үшін шеше алмайды. Мысал қарастырайық:

```
void f(int i, double fpd)
{
    char c = i;
    // иә: char типі өте кіші бүтін сандарды бейнелейді
    short s = i;
    // қауіпті: int типіндегі айнымалы short
    // типіндегі айнымалыға берілген жады аймағына
    // сыймай қалуы мүмкін
    i = i+1;
    // егер i саны максимал сан болса, не болады?
    long lg = i*i;
    // қауіпті: long типті айнымалы нәтижені сыйғыза алмайды
    float fps = fpd;          // қауіпті: a large double
    // типіндегі айнымалы float типіне сыя алмайды
    i = fpd;
    // қысқарту: мысалы, 5.7 - > 5
    fps = i;
    // дәлдік жоғалуы мүмкін (өте үлкен бүтін сан үшін)
}

void g()
{
    char ch = 0;
    for (int i = 0; i<500; ++i)
        cout << int(ch++) << '\t';
}
```

Егер күмәндансаңыз, тәжірибе жасап көріңіз! Тайсалмау керек және де жай ғана құжаттаманы оқу да жеткіліксіз. Тәжірибе жасамай, сандық типтерге байланысты күрделі құжаттама мазмұнын түсінбей қалуыңыз мүмкін.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

g() функциясын орындаңыз. **f()** функциясын ол **c**, **s**, **i** және т.б. айнымалыларды баспаға шығаратындай етіп толықтырыңыз. Программаны әртүрлі мәндер үшін тесттен өткізіңіз.

Бүтін сандарды бейнелеу мен оларды түрлендіру келесі 25.5.3 бөлімінде қарастырылады. Қате кету мүмкіндігін азайту үшін мәліметтердің бірнеше ғана типтер санын қолданумен шектеліңіз. Мысалы, **float** типін пайдаланбай, тек **double** типін ғана қолдана отырып, біз **double** – **float** түрлендіруіне байланысты туындайтын қателердің пайда болу мүмкіндігін шектейміз. Мысалы, біз есептеулеріміз үшін тек **int**, **double** және **complex** типтерін, символдар үшін **char**, ал логикалық мәндер үшін **bool** типтерін ғана пайдалануды ұсынамыз. Қалған арифметикалық типтерді тек аса қажеттілік туғанда ғана пайдалануға тырысу қажет.

24.2.1 Сандық диапазондар шектері

C++ тілінің әрбір жүзеге асырылуында программалаушы диапазондар шектерін тексеру, сигналдық белгілер орнату, т.с.с. әрекеттерді орындай алуы үшін құрамдас типтердің қасиеттерін **<limits>**, **<limits.h>** және **<float.h>** тақырыптарында анықтайды. Мұндай мәнде Б9.1 бөлімінде көрсетілген. Олар төменгі деңгейдегі құралдар жасау үшін маңызды рөл атқарады. Басқа да қосымша программалар бар болғанмен, егер олар сізге керек болып жатса, онда сіз тікелей аппараттық жабдықтамамен жұмыс істейсіз. Мысалы, көбінесе тілді жүзеге асыру нақтылықтары жайлы сұрақтар туындайды, мысалы, "**int** типі қаншалықты үлкен болып табылады?" немесе "**char** типінің таңбасы бар ма?" Бұларға жүйелік құжаттамалардан белгілі бір анық жауап табу қиын болып жатады, ал стандартта тек минималды талаптар ғана көрсетілген. Дегенмен, бұл сұрақтарға жауап беретін программаларды жеңіл жазып шығуға болады.

```
cout << "int типіндегі байттар саны: " << sizeof(int) << '\n';
cout << "int типіндегі ең үлкен сан: " << INT_MAX << endl;
cout << "int типіндегі ең кіші сан: "
     << numeric_limits<int>::min() << '\n';

if (numeric_limits<char>::is_signed)
    cout << "char типінің таңбасы бар\n";
else
    cout << "char типінің таңбасы жоқ\n";

cout << "минималды мәні бар char: "
     << numeric_limits<char>::min() << '\n';
cout << "char типінің минимал мәні: "
     << int(numeric_limits<char>::min()) << '\n';
```

Егер сіз әртүрлі компьютерлерде жұмыс істеуге тиіс программа жазатын болсаңыз, онда бұл мәліметті сіздің программаңыз үшін қолжетімді қылып қою

қажеттілігі туындайды. Әйтпесе сізге программаны сүйемелдеуді күрделендіріп, жауаптарын оған "тігіп" қоюға тура келеді.

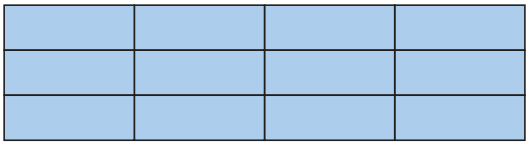
Осындай шектеулер программадағы толып кету әркеттерін де анықтауға пайдалы болып табылады.

24.3 Жиымдар

Жиым (**array**) – бұл әрбір элементін пайдалану оның индексі (позициясы) арқылы орындалатын тізбек. Бұл түсініктің синонимі болып вектор (vector) саналады. Бұл тарауда біз элементтері де жиым болып табылатын көпөлшемді жиымдарға көңіл бөлеміз. Әдетте көпөлшемді жиым матрица (**matrix**) деп аталады. Синонимдердің көп түрлілігі бұл түсініктің кең таралғанын және пайдалылығын білдіреді. Стандартты **vector** (Б.4 бөлімін қ.), **array** (20.9 бөлімін қ.) кластары және де құрамдас жиым (А.8.2 бөлімін қ.) бірөлшемді жиым болып табылады.

Бір және екіөлшемді жиымды былай көрсетуге болады:

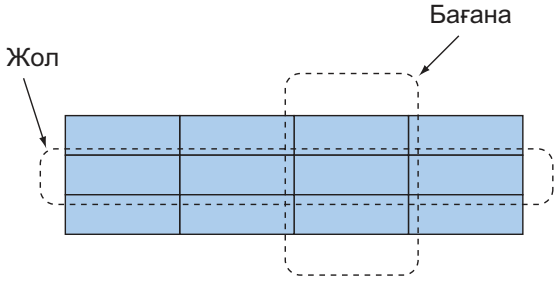
Вектор (мысалы, Matrix<Int>v(4)) бірөлшемді жиым немесе 1xN матрицасы болып аталады



3x4 матрицасы (мысалы, Matrix<Int,2>m(3,4)) екіөлшемді жиым болып аталады

Көптеген "сандарды сындыру" ("**number crunching**") деп аталатын есептеулерде жиымдар маңызды рөл атқарады. Ең қызықты ғылыми, техникалық, статистикалық және қаржылық есептеулер жиымдармен тығыз байланысты болып келеді.

Көбінесе жиым бағаналар мен жолдардан тұрады деп айтылады.



3x4 матрицасы
екіөлшемді жиым болып аталады
3 жол
4 бағана

Бағана – бұл бірдей алғашқы координаталары (x-координаталары) бар элементтер тізбегі. Жол – бұл бірдей екінші координаталары (y-координаталары) бар элементтер жиыны.

24.4 C тілі стиліндегі көпөлшемді жиымдар

C++ тілінде көпөлшемді жиым ретінде құрамдас жиымды пайдалануға болады. Мұнда көпөлшемді жиым жиымның жиымы болып саналады, яғни элементтері жиым болып келген жиым түрінде қарастырылады. Мысал қарастырайық:

```
int ai[4];                  // бірөлшемді жиым
double ad[3][4];            // екіөлшемді жиым
char ac[3][4][5];           // үшөлшемді жиым
ai[1] = 7;
ad[2][3] = 7.2;
ac[2][3][4] = 'c';
```

Бұл тәсіл бірөлшемді жиымның барлық артықшылықтары мен кемшіліктерін мұралайды.

- Артықшылықтары:
  - Аппараттық жабдықтамалар арқылы тікелей бейнелеу;
  - Тиімді төменгі деңгейдегі операциялар;
  - Тікелей тілдік сүйемелдеу.
- Мәселелері:
  - C тілі стилінде көпөлшемді жиымдар жиымдар жиымы болып табылады (төмендегіні қ.);
  - Бекітілген мөлшерлер (мысалы, компиляция кезінде бекітілгендер). Егер жиым мөлшерін программаның орындалу кезінде анықтағыңыз келсе, онда бос жады аймағын пайдалануыңыз керек;
  - Жиымдарды тиянақты түрде беруге болмайды. Жиым кез келген шағын мүмкіндікте өзінің бірінші элементіне нұсқауышқа айналып кетеді;
  - Диапазондарды тексеру жоқ. Әдеттегідей, жиым өз мөлшерін білмейді;
  - Жиымдармен орындалатын операциялар жоқ, тіпті меншіктеу де (көшіру де) жоқ.

Құрамдас жиымдар сандық есептеулерде кең қолданылады. Олар да қателер мен күрделіліктердің негізгі қайнар көзі болып табылады. Мұндай программаларды

құру мен түзету көптеген адамның басын ауыртады. Егер сіз құрамдас жиымдарды қолдануға мәжбүр болсаңыз, оқулықтарды оқып шығыңыз (мысалы, The *C++ Programming Language,* Appendix C, pp. 836–40). Өкінішке орай, C++ тілі көпөлшемді жиымдарды C тілінен мұралау жолымен алған, сондықтан ол осы кезге шейін көптеген программаларда қолданылып келеді.

Көптеген іргелі мәселелер көпөлшемді жиымдарды тікелей беруге болмайтынынан туындайды, сондықтан нұсқауыштармен жұмыс істеп, көпөлшемді жиымның позицияларын анықтауға байланысты тікелей есептеулер орындауға тура келеді. Мысал қарастырайық:

```
void f1(int a[3][5]);
// тек [3][5] матрицасында мағынасы бар

void f2(int [ ][5], int dim1);
// алғашқы өлшемі айнымалы бола алады

void f3(int [5][ ], int dim2);
// қате: 2-өлшем айнымалы бола алмайды

void f4(int[ ][ ], int dim1, int dim2);
// қате (тіпті жұмыс істемейді)

void f5(int* m,int dim1,int dim2)
// түсініксіз, бірақ жұмыс істейді
{
   for(int i=0; i<dim1; ++i)
      for(int j = 0; j<dim2; ++j) m[i*dim2+j] = 0;
}
```

Мұнда біз **m** жиымын, тіпті ол екіөлшемді болса да, **int*** нұсқауышы ретінде береміз. Екінші айнымалы параметр болуы тиіс болғандықтан, бізде компиляторға **m** жиымын (**dim1,dim2**) жиым түрінде екенін хабарлау мүмкіндігі жоқ, сондықтан біз нұсқауышты бірінші ұяға береміз.

m[i*dim2+j] өрнегі, негізінде, **m[i,j]** дегенді білдіреді, бірақ компилятор **m** айнымалысы екіөлшемді жиым екенін білмейтіндіктен, біз алдымен **m[i,j]** элементінің компьютер жадындағы позициясын есептеп шығаруымыз керек.

Бұл тәсіл өте күрделі және қателерге төтеп бере алмайды. Әрі өте баяу орындалады, өйткені элементтің позициясын тікелей түрде есептеп шығару оны ықшамдауды күрделендіріп жібереді. Мұндай жағдайдан қалай шығуды үйретудің орнына, біз құрамдас жиымдарға байланысты проблемаларды толығынан шеше алатын C++ кітапханасына көбірек назар аударатын боламыз.

24.5 `Matrix` кітапханасы

Сандық есептеулердегі жиымның (матрицаның) негізгі атқаратын қызметі қандай?

- "Менің кодым көптеген математика оқулықтарында баяндалған жиымдардың сипатталуына өте ұқсас болып көрінуі тиіс".
  - Және бұл векторларға, матрицаларға және тензорларға қатысты болып табылады.
- Программаны компиляциядан өткізу және орындалу кезеңдерінде тексеру.
  - Кез келген өлшемдегі жиымдар.
  - Кез келген өлшемдегі элементтер саны да кез келген жиымдар.
- Жиымдар толыққанды айнымалылар/объектілер болып табылады.
  - Оларды кез келген жаққа беруге болады.
- Жиымдармен орындалатын кәдімгі операциялар:
  - Индекстеу: `()`
  - Қию: `[]`
  - Меншіктеу: `=`
  - Қайта есептеу операциялары (`+=`, `-=`, `*=`, `%=` және т.б.).
  - Құрамдас векторлық операциялар (мысалы, `res[i]=a[i]*c+b[2]`).
  - Скалярлық көбейтулер (`res=a[i]*b[i]` элементтері қосындысы; тағы да `inner_product` атымен белгілі).
  - Негізінде, жиымдарды/векторларды дәстүрлі есептеу ісін программа мәтініне автоматты түрлендіруді қамтамасыз етеді, бұған қарсы жағдайда программаны өзіңіз жазуға тиіс боласыз (ол алдыңғыдан тиімсіз болмауы тиіс).
  - Жиымдарды, қажет болғанда, үлкейтуге болады (оларды жүзеге асыру кезінде "сиқырлы" сандар пайдаланылмайды).

`Matrix` кітапханасы тек осыны ғана жасайды. Егер сіз одан көбірек талап еткізіңіз келсе, онда жиымдарды өңдеуді, сиретілген жиымдармен жұмыс істеуді, компьютер жадын бөлуді басқару, т.с.с. жасайтын күрделі функцияларды өзіңіз жазып шығуыңыз керек немесе өз талаптарыңызға сәйкес келетін басқа кітапхананы пайдалануыңыз қажет. Бірақ осындай көптеген талаптарды `Matrix`

кітапханасының үстіне қондырмалар түрінде қойылған алгоритмдер мен мәліметтер құрылымы көмегімен орындауға болады. **Matrix** кітапханасы ISO C++ стандартының бөлігі болып табылады. Сіз оның сипаттамаларын **Matrix.h** тақырыбындағы сайттан таба аласыз. Ол өзінің мүмкіндіктерін **Numeric_lib** атаулар кеңістігінде анықтайды. "Вектор" және "жиым" сөздері C++ тілі кітапханаларында асыра жүктелгендіктен, біз **Matrix** сөзін таңдаған болатынбыз. **Matrix** кітапханасының жүзеге асырылуы, мұнда көрсетілмеген күрделі әдістерге негізделген.

24.5.1 Өлшемдер және қолжетімділік

Қарапайым мысал қарастырайық:

```
#include "Matrix.h"
using namespace Numeric_lib;

void f(int n1, int n2, int n3)
{
    Matrix<double,1> ad1(n1);
    // double типті элементтер; бірөлшемді
    Matrix<int,1> ai1(n1);
    // int типті элементтер; бірөлшемді;
    ad1(7) = 0;    // индекстеу ( ) – Fortran тілі стилінде
    ad1[7] = 8;       //  индекстеу [ ] – C тілі стилінде

    Matrix<double,2> ad2(n1,n2);     // екіөлшемді
    Matrix<double,3> ad3(n1,n2,n3); // үшөлшемді
    ad2(3,4) = 7.5;          // анық көпөлшемді индекстеу
    ad3(3,4,5) = 9.2;
}
```

Сонымен, **Matrix** айнымалысын (**Matrix** класының объектісі) анықтай отырып, сіз элементтер типін және жиым өлшемі санын көрсетуіңіз керек. Әрине, **Matrix** класы шаблондық болып саналады, ал элементтер типі мен өлшем саны шаблондық параметрлер түрінде болады. Нәтижесінде, **Matrix** класына екі шаблондық параметр (мысалы, **Matrix<double,2>**) беріп, типін (класын) аламыз. Тип арқылы аргументтерді (мысалы, **Matrix<double,2> ad2(n1,n2);**) көрсете отырып, объектілерді анықтауға болады; бұл аргументтер жиым өлшемін береді. Сонымен, **ad2** айнымалысы өлшемдері **n1** және **n2** болып келген екіөлшемді жиым болып табылады, оны **n1,n2** өлшемді матрица деп те атайды. **Matrix** класының екіөлшемді объектісінен жарияланған типтегі элементті алу үшін екі индексті көрсету керек. Құрамдас жиымдардағы және **vector** класы объектілеріндегі сияқты **Matrix** класының объектісіндегі элементтер

нөлден (**Fortran** тіліндегі секілді бірден емес) бастап индекстеледі; басқаша айтқанда, **Matrix** класының объектісіндегі элементтер **[0,max)** диапазонында нөмірленеді, мұндағы **max** – элементтер саны.

Бұл тікелей оқулықтан алынған. Егер сізде проблемалар туындаса, программалау пәнінен емес, математика саласынан керекті оқулықтармен танысып шығу керек. Мұның жалғыз кілтипаны біздің **Matrix** класының объектісіндегі өлшем санын көрсетпегенімізде болып отыр: келісім бойынша ол бірөлшемді жиым болып табылады. Біздің индекстеуді **[ ]** операторы (C/C++ тілдері стилінде) арқылы да және **()** операторы (Fortran тілі стилінде) арқылы да орындай алатынымызға назар аударыңыз.

Бұл бізге үлкен өлшемдегі жиымдармен жұмыс істеуге мүмкіндік береді. **[x]** индексі **Matrix** класының объектісіндегі жеке жолды ерекшелей отырып, әрқашанда жеке индексті білдіреді. Егер a айнымалысы **Matrix** класының n-өлшемді объектісі болса, онда **a[x]** – **Matrix** класының **(n1)**-өлшемді объектісі. **(x,y,z)** белгілеуі **Matrix** класы объектісінің сәйкес элементін белгілей отырып, бірнеше индекстерді пайдалануды білдіреді, индекстер саны өлшемдер санына тең болуы тиіс.

Егер қате жіберсек, не болатынын көрейік.

```
void f(int n1, int n2, int n3)
{
    Matrix<int,0> ai0;   // қате: 0-өлшемді матрица болмайды

    Matrix<double,1> ad1(5);
    Matrix<int,1> ai(5);
    Matrix<double,1> ad11(7);

    ad1(7) = 0;
    // Matrix_error аластамасы (7 диапазоннан тысқары)
    ad1 = ai;        // қате: элементтер типі әртүрлі
    ad1=ad11;  //Matrix_error аластамасы (өлшемдері әртүрлі)

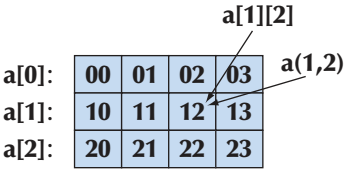
    Matrix<double,2> ad2(n1); //қате: 2-өлшем ұзындығы жоқ
    ad2(3) = 7.5;      // қате: индекстер саны дұрыс емес
    ad2(1,2,3) = 7.5;    // қате: индекстер саны дұрыс емес

    Matrix<double,3> ad3(n1,n2,n3);
    Matrix<double,3> ad33(n1,n2,n3);
    ad3=ad33; //OK: элементтер типі бірдей, өлшемдері бірдей
}
```

Өлшемдердің жарияланған саны мен оларды пайдаланудың арасындағы сәйкессіздік компиляциядан өткізу кезеңінде анықталады. Диапазон шегінен

тысқары шығып кету программаны орындау кезеңінде айқындалады да, сол сәтте `Matrix_error` аластамасы жасалады (генерацияланады).

Матрицаның бірінші өлшемі – жол, ал екіншісі – бағана, сондықтан индекс – екіөлшемді матрица (екіөлшемді жиым), ол (жол, бағана) түріндегі кесте тәрізді болады. Бұған қоса [жол][бағана] белгілеуін де пайдалануға болады, өйткені екіөлшемді матрицаны бірөлшемді индекс арқылы индекстеу бірөлшемді матрицаны – жолды туындатады. Бұл айтылғандарды келесідей түрде бейнелеуге болады.



`Matrix` класының бұл объектісі компьютер жадында жолдар бойынша орналасады.

00	01	02	03	10	11	12	13	20	21	22	23

`Matrix` класы өз өлшемін біледі, сондықтан оның элементтерін қарапайым аргумент ретінде беруге болады.

```
void init(Matrix<int,2>&a)
// әрбір элементті сипаттамалық мәнмен инициалдау
{
    for (int i=0; i<a.dim1(); ++i)
       for (int j = 0; j<a.dim2(); ++j)
            a(i,j) = 10*i+j;
}

void print(const Matrix<int,2>&a)
//  элементтерді жолдар бойынша шығару
{
    for (int i=0; i<a.dim1(); ++i)  {
       for (int j = 0; j<a.dim2(); ++j)
          cout << a(i,j) <<'\t';
       cout << '\n';
    }
}
```

Сонымен, `dim1()` – бірінші өлшемдегі элементтер саны, `dim2()` – екінші

өлшемдегі элементтер саны, т.с.с. Элементтер типі мен өлшемдер саны **Matrix** класының бөлігі болып табылады, сондықтан **Matrix** класының объектісін аргумент (бірақ шаблон жазуға болады) ретінде алатын функцияны жазу мүмкін емес.

```
void init(Matrix&a);
// қате: элементтер типі мен өлшем саны көрсетілмеген
```

Matrix кітапханасының матрицалық операциялары жоқ екеніне назар аударыңыз, мысалы, екі төрт өлшемді матрицаларды қосу немесе екіөлшемді матрицаны бірөлшемді матрицаға көбейту. Бұл операцияларды ықшам түрде жүзеге асыру осы кітапхана аясынан тысқары жатыр. **Matrix** кітапханасының үстіне сәйкестендірілген матрицалық кітапханаларды қондыруға болады (12-жаттығуды қ.).

24.5.2 Matrix класының бірөлшемді объектісі

Matrix класының қарапайым объектісі – бірөлшемді матрицамен не істеуге болады?

Мұндай объектіні жариялаудағы өлшемдер санын көрсетпеуге болады, өйткені келісім бойынша, бұл сан бірге тең.

```
Matrix<int,1> a1(10);
// a1 – бұл бүтін сандардың бірөлшемді матрицасы
Matrix<int> a(10);       // яғни Matrix<int,1> a(10);
```

Сонымен, **a** және **a1** объектілерінің типі бірдей – (**Matrix<int,1>**). **Matrix** класының әрбір объектісінен элементтердің жалпы саны мен белгілі бір өлшемдегі элементтер санын сұрауға болады. **Matrix** класының бірөлшемді объектісінде бұл параметрлер бірдей болып келеді.

```
a.size();    // Matrix класы объектісіндегі элементтер саны
a.dim1();    // бірінші өлшемдегі элементтер саны
```

Компьютер жадында матрица элементтерін орналастыру сұлбасын пайдалана отырып, олардың бірінші элементіне нұсқауыш жасау арқылы матрица элементтерін пайдалануға болады:

```
int* p = a.data(); // жиым нұсқауышы арқылы мәліметтер аламыз
```

Бұл, C тілі стилінде, **Matrix** класының объектілерін аргумент ретінде нұсқауыштарды қабылдайтын функцияларға беру кезінде пайдалы болып табылады. Матрицаларды индекстеуге болады:

```
a(i);
// i-элемент (Fortran тілі стилінде), диапазонды тексеру арқылы
a[i]; // i-элемент (C тілі стилінде), диапазонды тексеру арқылы
a(1,2);  // қате: a – Matrix класының бірөлшемді объектісі
```

Көптеген алгоритмдер **Matrix** класының объектісі бөлігін пайдаланады. Бұл бөлік қима деп аталады да, ол **slice()** функциясы арқылы құрылады.

(**Matrix** класының объектісі бөлігі немесе элементтер диапазоны). **Matrix** класында осы функцияның екі нұсқасы бар:

```
a.slice(i);
// a[i]-ден басталып, соңғысымен аяқталатын элементтер
a.slice(i,n);
// a[i]-ден басталып, a[i+n- 1]-мен аяқталатын n элементтер
```

Индекстер мен қималарды меншіктеу операторының сол жағында да, оң жағында да қолдануға болады. Олар көшірмелерін жасамай, **Matrix** класының объектісі элементтеріне сілтеме жасайды. Мысал қарастырайық:

```
a.slice(4,4) = a.slice(0,4);
// матрицаның бірінші бөлігін оның екінші бөлігіне меншіктеу
```

Мысалы, егер **a** объектісі басында былай болса,

```
{ 1 2 3 4 5 6 7 8 }
```

онда мынаны аламыз:

```
{ 1 2 3 4 1 2 3 4 }
```

Көбінесе қималар **Matrix** класы объектілерінің бастапқы және соңғы элементтері арқылы берілетініне назар аударыңыз; яғни **a.slice(0,j)** – бұл **[0:j)** диапазоны, ал **a.slice(j)** – **[j :a.size())**. Ашып айтсақ, жоғарыдағы мысалды басқаша түрде жеңіл жазып шығуға болады:

```
a.slice(4) = a.slice(0,4);
// матрицаның бірінші бөлігін оның екінші бөлігіне меншіктеу
```

Басқаша айтқанда, белгілеу – әркімнің өз ісі. Сіз **a.slice(i,n)** өрнегінде **i** және **n** индекстерін a матрицасының диапазонынан тысқары шығып кететін етіп көрсете аласыз. Бірақ алынған қимада тек **a** объектісіне кіретін элементтер ғана болады. Мысалы, **a.slice(a.size())** қимасы **[i:a.size())** диапазонын білдіреді, ал **a.slice(a.size())** және **a.slice(a.size(),2)** – бұлар

Matrix класының бос объектілері. Бұл көптеген алгоритмдерде пайдалы болып саналады. Біз бұл белгілеулерді математикалық мәтіндерден алдық. Әрине, **a.slice(i,0)** қимасы **Matrix** класының бос объектісі. Бізге мұны әдейілеп жазбау керек еді, дегенмен, **a.slice(i,n)** өрнегі **n** параметрі **0**-ге тең болған кезде, бос матрица болып шығып (бұл қате жасамауға мүмкіндік береді), қарапайым түрге айналатын алгоритмдер бар.

Барлық элементтерді көшіру әдеттегідей орындалады.

```
Matrix<int> a2 = a;  // көшіретін инициалдау
a = a2;              // көшіретін меншіктеу
```

Matrix класының әрбір объектісіне құрамдас операцияларды қолдануға болады:

```
a *= 7;  // қайта есептеу: әрбір i үшін a[i]*=7
         // (мыналардан басқасы: +=, -=, /=)
a = 7;   // әрбір i үшін a[i]=7
```

Бұл егер элемент типін соған сәйкес оператор сүйемелдейтін болса, онда әрбір меншіктеу операторына және әрбір күрделі меншіктеу операторына (**=, +=, -=, /=, *=, %=, ^=, &=, |=, >>=, <<=**) қатысты болып саналады. Оған қоса, **Matrix** класының әрбір объектісіне функцияларды қолдануға болады.

```
a.apply(f);   // a[i]=f(a[i]) әрбір a[i] элементі үшін
a.apply(f,7); // a[i]=f(a[i],7) әрбір a[i] элементі үшін
```

Күрделі меншіктеу операторлары және **apply()** функциясы өздерінің **Matrix** типіндегі аргументін толықтырады. Егер біз **Matrix** класының жаңа объектісін құрғымыз келсе, онда келесі нұсқауды орындай аламыз:

```
b = apply(abs,a);     // құрамында b(i)==abs(a(i)) бар
                  // Matrix класының жаңа объектісін құрамыз
```

Мұндағы **abs** функциясы – бұл абсолюттік мәнді есептейтін стандартты функция (24.8 бөлім). Негізінде, **apply(f,x)** функциясының **x.apply(f)** функциясымен байланысы **+** операторының **+=** операторымен байланысуымен бірдей болады. Мысал қарастырайық:

```
b = a*7;              // әрбір i үшін b[i] = a[i]*7
a *= 7;               // әрбір i үшін a[i] = a[i]*7
y = apply(f,x);       // әрбір i үшін y[i] = f(x[i])
x.apply(f);           // әрбір i үшін x[i] = f(x[i])
```

Нәтижесінде **a==b** және **x==y** болады.

Fortran тілінде, **apply** функциясының екінші нұсқасы жіберу функциясы ("broadcast" function) деп аталады. Бұл тілде көбінесе **apply(f,x)** орнына **f(x)** функциясын шақыру деп жазады. Осы мүмкіндік әрбір **f** функциясы үшін (Fortran тіліндегі сияқты тек жеке функциялар үшін ғана емес) қолжетімді болуы үшін, біз жіберу функциясына нақты бір ат беруіміз керек, сондықтан **apply** атын (қайтадан) пайдаландық.

Оған қоса, **a.apply(f,x)** болып жазылатын **apply** функция-мүшесінің нұсқасымен сәйкестікті қамтамасыз ету үшін біз былай жазамыз:

```
b = apply(f,a,x);      // әрбір i үшін b[i]=f(a[i],x)
```

Мысал қарастырайық:

```
double scale(double d, double s) { return d*s; }
b = apply(scale,a,7); //  әрбір i үшін b[i] = a[i]*7
```

apply() "автономдық" функциясы аргумент ретінде аргументтері бойынша нәтиже есептейтін функцияны қабылдайды, сонан соң сол нәтижені **Matrix** класының қорытынды объектісін инициалдау үшін қолданылады. Көбінесе бұл **Matrix** класының функция қолданылатын объектісін өзгертуге алып келмейді. Сонымен қатар, **apply()** функция-мүшесінің айырмашылығы бар, ол аргумент ретінде өз аргументтерін толықтырып өзгертетін функцияны қабылдайды; басқаша айтқанда, ол **Matrix** класының функция қолданылатын объектісі элементтерін өзгертеді. Мысал қарастырайық:

```
void scale_in_place(double& d, double s) { d *= s; }
b.apply(scale_in_place,7); // әрбір i үшін b[i] *= 7
```

Matrix класында дәстүрлі математикалық кітапханадан алынған көптеген пайдалы функциялар қарастырылған.

```
Matrix<int> a3 = scale_and_add(a,8,a2);
// біріктірілген қосу мен көбейту
int r = dot_product(a3,a);          // скалярлық көбейту
```

scale_and_add() операциясын көбінесе *біріктірілген көбейту мен қосу* (*fused multiply-add*) немесе жай ғана *fma* деп атайды; оның анықтамасы **Matrix** класының объектісіндегі әрбір **i** үшін мынадай болады:

```
result(i)=arg1(i)*arg2+arg3(i).
```

Скалярлық көбейту және **inner_product** атымен де белгілі, ол 21.5.3 бөлімінде сипатталған; оның анықтамасы **Matrix** класының объектісіндегі әрбір **i** үшін мынадай болады:

```
result+=arg1(i)*arg2(i)
```

мұнда объектіні жинақтау нөлден басталады.

Бір өлшемді жиымдар өте кең таралған; оларды құрамдас жиымдар түрінде, сонымен қоса, **Matrix** және **vector** кластары арқылы да бейнелеуге болады. **Matrix** класын ***=** сияқты матрицалық операцияларды орындау керек болған кезде немесе **Matrix** класының объектісі осы кластың өлшемі жоғары болып келетін басқа объектілерімен әрекеттесетін кезде қолдану қажет.

Бұл кітапхананың пайдалылығын оның математикалық операциялармен жақсы байланысқанымен және де оны қолдану кезінде матрицаның әрбір элементімен жұмыс істеу барысында цикл жазбауға болатындығымен түсіндіруге болады. Кез келген жағдайда, жұмыс нәтижесінде біз қысқалау код аламыз және қате жіберу мүмкіндігі де азаяды. **Matrix** класының объектілері, мысалы, көшіру, барлық элементтерге меншіктеу және барлық элементтермен операциялар орындау әрекеттері циклдерді қолданбауға (сондықтан олармен байланысты проблемалар жайлы ойланбасақ та болады) мүмкіндік береді.

Matrix класының мәліметтерді құрамдас жиымдардан **Matrix** класының объектісіне көшіруге арналған екі конструкторы бар. Мысал қарастырайық:

```
void some_function(double* p, int n)
{
     double val[] = { 1.2, 2.3, 3.4, 4.5 };
     Matrix<double> data(p,n);
     Matrix<double> constants(val);
     // . . .
}
```

Бұл көбінесе біз программаның басқа бөліктерінде жасалған, **Matrix** класының объектілерін пайдаланбайтын мәліметтерді кәдімгі жиымдар немесе векторлар түрінде алғанымызда, пайдалы болып саналады.

Компилятордың инициалданған жиымдағы элементтер санын өзі анықтай алатынына назар аударыңыз, сондықтан **constants** объектісін анықтау кезінде бұл санды көрсету міндетті емес, ол **4**-ке тең болады. Басқа жағынан, егер элементтер тек нұсқауышпен берілсе, онда компилятор олардың санын білмейді, сондықтан **data** объектісін анықтау кезінде біз **p** нұқсқауышын да және **n** элементтер санын да беруіміз керек.

24.5.3 Matrix класының екіөлшемді объектісі

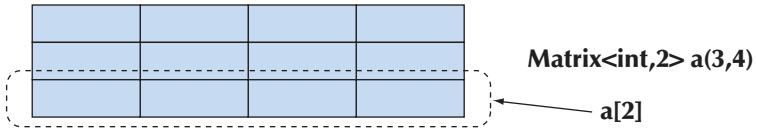
Matrix кітапханасының жалпы идеясы, жиымдар өлшемін тікелей көрсететін жағдайлардан басқа кездерде, әртүрлі өлшемдегі матрицалар көбінесе бір-бірімен өте ұқсас болып келеді. Сондықтан да **Matrix** класының бірөлшемді объектілері үшін айтылғандардың басым бөлігі көпөлшемді жиымдарға да қатысты болып келеді.

```
Matrix<int,2> a(3,4);
int s = a.size();   // элементтер саны
int d1 = a.dim1();  // жолдағы элементтер саны
int d2 = a.dim2();  // бағанадағы элементтер саны
int* p = a.data();  // C тілі стилінде нұсқауыш арқылы
                    // мәліметтер шығарамыз
```

Біз жалпы элементтер санын және әрбір өлшемдегі элементтер санын да сұрай аламыз. Одан басқа, компьютер жадында матрица түрінде орналасқан элементтерге нұсқауыш ала аламыз. Біз индекстерді пайдалана аламыз.

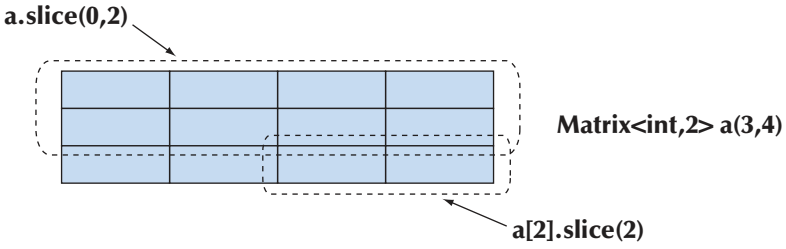
```
a(i,j);       // диапазонды тексеретін (i,j)-элемент
              // (Fortran тілі стилінде)
a[i];         // диапазонды тексеретін i-жол
              // (C тілі стилінде)
a[i][j];      // (i,j)-элемент (C тілі стилінде)
```

Екі өлшемді **Matrix** класының объектісінде **[i]** конструкциясы бойынша индекстеу i-жолды көрсететін бірөлшемді **Matrix** класының объектісін құрады. Бұл біздің жолдарды оларды **Matrix** класының объектілерін ала алатын, тіпті, **(a[i].data())** құрамдас жиымын да ала алатын операторлар мен функциялар аргументтер ретінде шығарып алатынымыздың білдіреді. Мұндағы **a(i,j)** индекстеу түрі **a[i][j]** индекстеуінен жылдам жұмыс істеуі мүмкін, бірақ ол компилятор мен оптимизаторға көп байланысты болып келеді.



Біз қималар алуымыз мүмкін.

```
a.slice(i);       // a[i]-ден бастап соңғы жолға дейін
a.slice(i,n); // a[i]-ден бастап, a[i+n- 1]-жолға дейін
```



Екі өлшемді **Matrix** класының екіөлшемді объектісінің қимасының өзі осы кластың екіөлшемді объектісі (жолдар саны бұдан аз болуы мүмкін) болып табылады. Екі өлшемді матрицалармен орындалатын таралған операциялар бірөлшемді жиымдармен орындалатын операциялармен бірдей болып келеді. Бұл операцияларға элементтердің қалай сақталатыны маңызды емес; олар барлық элементтерге компьютер жадында орналасу реті бойынша орындалады.

```
Matrix<int,2> a2 = a;       // көшіретін инициалдау
a = a2;                     // көшіретін меншіктеу
a *= 7;          // қайта есептеу (және +=, - =, /=, т.б.)
a.apply(f);
// әрбір a(i,j) элементі үшін a(i,j)=f(a(i,j))
a.apply(f,7);
// әрбір a(i,j) элементі үшін a(i,j)=f(a(i,j),7)
b=apply(f,a);
// b(i,j)==f(a(i,j)) өрнегімен жаңа матрица құрамыз
b=apply(f,a,7);
// b(i,j)==f(a(i,j),7) өрнегімен жаңа матрица құрамыз
```

Жолдарды алмастыру пайдалы әрекет болып отыр, сондықтан біз оны да қарастырып шығайық.

```
a.swap_rows(7,9);     // a[7] <-> a[9] жолдарын ауыстыру
```

Ал бағаналарды алмастыру **swap_columns()** әрекеті жоқ. Егер ол сізге қажет болып жатса, оны өзіңіз жазып шыға аласыз (11-жаттығуды қ.). Компьютер жадында матрицаны жолдар бойынша сақтауға орай бағаналар мен жолдар тең құқықты емес. Осы ассимметрия **[i]** операторының тек жолды қайтаратынына байланысты орын алады (ал бағаналар үшін мұндай операция қарастырылмаған). Сонымен **(i,j,k)** үштігіндегі бірінші **i** индексі жолды таңдайды. Бұл ассимметрияның математикалық түп-тамыры тереңде жатыр.

Екі өлшемді матрицамен орындалатын әрекеттер саны шексіз болуы ықтимал.

```
enum Piece { none,pawn,knight,queen,king,bishop,rook };
Matrix<Piece,2> board(8,8);    // шахмат тақтасы

const int white_start_row = 0;
const int black_start_row = 7;

Piece init_pos[] = {rook, knight, bishop, queen,
                              king, bishop, knight, rook};
Matrix<Piece> start_row(init_pos);
// init_pos арқылы элементтерді инициалдау
```

```
Matrix<Piece> clear_row(8);
// келісім бойынша мәні бар 8 элемент
```

`clear_row` объектісін инициалдау `none==0` шартын беру мүмкіндігін және осы элементтердің келісім бойынша нөлмен инициалданатынын пайдаланады. Біз басқа код жазуды қалар едік:

```
Matrix<Piece> start_row
   = {rook,knight,bishop,queen,king,bishop,knight,rook};
```

Бірақ бұл жұмыс істемейді (C++ (C++0x) тілінің жаңа нұсқасы практикаға енгізілгенге дейін), сондықтан жиымды инициалдау ісінде аздаған қулықтар жасап, оны `Matrix` класының объектілерін инициалдау үшін қолдануға тура келеді. Біз `start_row` және `clear_row` объектілерін мына түрде пайдалана аламыз:

```
board[white_start_row] = start_row;
// ақ фигураларды орналастыру
for(int i=1;i<7;++i) board[i]=clear_row;
//тақта ортасын босату
board[black_start_row] = start_row;
// қара фигураларды орналастыру
```

Мыналарға назар салыңыз: біз `[i]` өрнегін пайдаланып, жолды ашқанымызда, `lvalue` мәнін алдық (4.3 бөлімін қ.); басқаша айтқанда, біз нәтижені `board[i]` элементіне меншіктей аламыз.

24.5.4 `Matrix` класы объектілерін енгізу-шығару

`Matrix` кітапханасы `Matrix` класының бір және екіөлшемді объектілерін енгізу және шығару үшін өте қарапайым құралдар ұсынады:

```
Matrix<double> a(4);
cin >> a;
cout << a;
```

Кодтың осы фрагменті жүйелі жақшаларға алынып, бос орындармен бөлінген `double` типіндегі төрт санды оқиды; мысалы:

```
{ 1.2 3.4 5.6 7.8 }
```

Шығару өте қарапайым, сондықтан біз нені енгізсек, соны көреміз.

`Matrix` класының екіөлшемді объектілерін енгізу-шығару механизмі тік

жақшаға алынған **Matrix** класының бірөлшемді объектілері тізбегін оқиды және жазады. Мысал қарастырайық:

```
Matrix<int,2> m(2,2);
cin >> m;
cout << m;
```

Ол мынадай жазбаны оқиды:

```
{
{ 1 2 }
{ 3 4 }
}
```

Мәлімет шығару өте ұқсас болып келеді.

Matrix класының **<<** және **>>** операторлары қарапайым программалар жазуға мүмкіндік береді. Бұдан күрделі жағдайларда бізге оларды өз операторларымызбен ауыстыруға тура келеді. Осыған байланысты **Matrix** класының **<<** және **>>** операторлары **MatrixIO.h** тақырыбына (**Matrix.h** тақырыбына емес) орналастырылған, сондықтан өз программаңызда матрицаны пайдалану үшін, **MatrixIO.h** тақырыбын қосу міндетті емес.

24.5.5 Matrix класының үшөлшемді объектісі

Негізінде, **Matrix** класының үшөлшемді объектілері, бұдан жоғары өлшемдегі матрицалар сияқты, екіөлшемдіге ұқсас, тек олардың өлшемдері ғана артық болып келеді. Мысал қарастырайық:

```
Matrix<int,3> a(10,20,30);

a.size();           // элементтер саны
a.dim1();           // 1-өлшемдегі элементтер саны
a.dim2();           // 2-өлшемдегі элементтер саны
a.dim3();           // 3-өлшемдегі элементтер саны
int* p = a.data();
            // нұсқауыш бойынша мәліметтер алады (C тілі)
a(i,j,k);   // (i,j,k)-элемент (Fortran стилі),
            // диапазонды тексеру арқылы
a[i];       // i-жол (C стилі), диапазонды тексеру арқылы
a[i][j][k];             // (i,j,k)-элемент (C стилі)
a.slice(i);             // i-жолынан соңына дейін
a.slice(i,j);           // i-жолынан j-жолына дейін
```

```
Matrix<int,3> a2=a;                    // көшіретін инициалдау
a = a2;                                // көшіретін меншіктеу
a *= 7;                  // қайта есептеу (және +=, -=, /=, т.б.)
a.apply(f);
// әрбір a(i,j,k) элементі үшін a(i,j,k)=f(a(i,j,k))
a.apply(f,7);
// әрбір a(i,j,k) элементі үшін a(i,j,k)=f(a(i,j,k),7)
b=apply(f,a);
// b(i,j,k)==f(a(i,j,k)) өрнегімен жаңа матрица құру
b=apply(f,a,7);
// b(i,j,k)==f(a(i,j,k),7) өрнегімен жаңа матрица құру
a.swap_rows(7,9);  // a[7] <-> a[9] жолдарын алмастыру
```

Егер сіз **Matrix** класының екіөлшемді объектілерімен жұмыс істей білсеңіз, онда үшөлшемді объектілермен жұмыс істей бересіз. Мысалы, мұнда **a** – үшөлшемді матрица болсын, сондықтан **a[i]** – екіөлшемді (**i** индексі берілген диапазоннан тысқары шықпайтын шарт орындалған жағдайда); **a[i][j]** – бірөлшемді (**j** индексі берілген диапазоннан тысқары шықпайтын шарт орындалған жағдайда); **a[i][j][k]** – **int** типті элемент (**k** индексі берілген диапазоннан тысқары шықпайтын шарт орындалған жағдайда).

Біз әлемді үш өлшемде көретіндіктен, модельдеуде де көбінесе **Matrix** класының үшөлшемді объектілері пайдаланылады (мысалы, физикалық модельдеудегі координаталардың декарттық жүйесінде).

```
int grid_nx;   // тордың шешу қабілеті; алдында беріледі
int grid_ny;
int grid_nz;
Matrix<double,3> cube(grid_nx, grid_ny, grid_nz);
```

Егер төртінші өлшеу бірлігі ретінде уақытты қосатын болсақ, онда **Matrix** класының төрт өлшемді объектілері қажет етілетін төрт өлшемді кеңістік аламыз. Ары қарай да солай жалғаса береді.

24.6 Мысал: сызықтық теңдеулер жүйесін шешу

Егер сіз программаның сандық есеп үшін қандай математикалық есептеулер өрнектейтінін білсеңіз, онда оның мағынасы бар, ал егер білмесеңіз, кодтың ешқандай да мағынасы болмайды. Егер сіз сызықтық алгебра негіздерін білсеңіз, онда төмендегі мысал сізге қарапайым болып көрінер; ал егер олай болмаса, онда есептің оқулықтағы шешімі аздаған толықтырулармен программаға айналғанын көресіз.

Бұл мысал **Matrix** класының шынайы және маңызды пайдаланылуын көрсету үшін таңдалып алынған. Біз келесі түрдегі сызықтық теңдеулер жүйесін шығарамыз:

$$\begin{aligned} a_{1,1}x_1 + \cdots + a_{1,n}x_n &= b_1 \\ &\vdots \\ a_{n,1}x_1 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

мұндағы *x әріптері n* белгісіздерді білдіреді, ал *a* мен *b әріптері* – константалар. Қарапайымдылығы үшін белгісіздер мен константалар жылжымалы нүктелі сандар деп қабылдаймыз.

Біздің мақсатымыз – көрсетілген *n* теңдеулерді бір мезетте қанағаттандыратын белгісіздерді табу. Осы теңдеулерді матрица мен екі вектор арқылы ықшам түрде өрнектеуге болады:

$$\mathbf{Ax} = \mathbf{b}$$

мұндағы **A** — *nxn* коэффициенттердің квадраттық матрицасы:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \vdots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$$

мұндағы **x** пен **b** сәйкесінше белгісіздер векторлары мен константалар.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

A матрицасы мен **b** векторына байланысты бұл жүйенің бірде-бір шешімі болмауы мүмкін, бір ғана шешімі немесе шексіз көп шешімі болуы да мүмкін. Сызықтық жүйелерді шешу тәсілдерінің көптеген түрі бар. Біз Гаусстың аластамасы деп аталатын классикалық сұлбаны пайдаланамыз. Алдымен біз **A** матрицасы мен **b** векторын **A** матрицасы жоғарғы үшбұрыш болатындай қылып, яғни диагоналдан төменгі элементтердің барлығы да нөл болатын етіп түрлендіреміз. Басқаша айтқанда, жүйе мынадай болып шығады:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ 0 & \ddots & \vdots \\ 0 & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Алгоритм күрделі емес. *(i, j)* позициясындағы элемент нөлге тең болуы үшін *i* жолын бір константаға көбейтуіміз керек, нәтижесінде *(i, j)* позициясындағы элемент *j* бағанасындағы басқа бір элементке, мысалы, *a(k,j)* элементіне тең болуы тиіс. Бұдан кейін бір теңдеуден екінші теңдеуді азайтамыз да, *a(i,j)=0* мәнін аламыз. Мұнда і жолындағы барлық элементтер сәйкесінше өзгеріп шығады.

Егер диагоналдағы барлық элементтер нөлге тең болмаса, онда жүйенің бір ғана шешімі болады, оны біз кері орналастыру арқылы таба аламыз. Алдымен соңғы теңдеуді шығарамыз (ол оңай орындалады).

$$a_{n,n}\ x_n = b_n$$

Әрине, *x[n]* мәні *b[n]/a(n,n)* мәніне тең болады. Енді жүйенің n-жолын алып тастайық та, *x[n -1]* мәнін табайық. Осы әрекетті *x[1]* мәні табылғанша жалғастыра береміз.

Әрбір *n* мәні үшін *a(n,n)*-ге бөлу әрекетін орындаймыз, сондықтан диагоналдық мәндер нөлге тең болмауы тиіс. Егер бұл шарт орындалмаса, онда кері орналастыру әрекеті дұрыс нәтиже бермейді. Бұл жүйенің шешімі жоқ дегенді немесе шексіз көп шешімі бар дегенді білдіреді.

24.6.1 Гаусстың классикалық аластамасы

Енді осы алгоритмнің C++ тіліндегі кодтар түрінде қалай өрнектелетінін қарап шығайық. Біріншіден, пайдаланылғалы отырған матрицалардың екі типтері үшін де ыңғайлы атаулар енгізіп, белгілеулерді қарапайым түрге келтірейік.

```
typedef Numeric_lib::Matrix<double, 2> Matrix;
typedef Numeric_lib::Matrix<double, 1> Vector;
```

Бұдан кейін осы алгоритмнің өзін өрнектейік:

```
Vector classical_gaussian_elimination(Matrix A, Vector b)
    {
       classical_elimination(A, b);
       return back_substitution(A, b);
    }
```

Басқаша айтқанда, біз **A** кіріс матрицасы мен **b** векторының көшірмелерін жасап, жүйені шешетін функцияны шақырамыз да, сонан кейін кері орналаластыру арқылы нәтижені есептейміз. Осылай есеп шығару жолын бөліктерге бөліп орындау мен ыңғайлы белгілеулер жүйесі барлық оқулықтарға енгізілген. Программаны аяқтау үшін біз `classical_elimination()` және `back_substitution()` функцияларын жүзеге асыруымыз керек. Есептің осы шығарылу тәсілін оқулықтан да тауып алуға болады:

```
void classical_elimination(Matrix& A, Vector& b)
 {
    const Index n = A.dim1();

// диагональдан төмен орналасқан элементтерін нөлге теңеп,
// бірінші бағанадан соңғы бағанаға дейін қарастырып өтеміз:
    for (Index j = 0; j<n- 1; ++j) {
       const double pivot = A(j, j);
       if (pivot == 0) throw Elim_failure(j);

       // i-жолда диагоналдан төмен тұрған элементтерді
       // нөлге теңейміз:
       for (Index i = j+1; i<n; ++i) {
           const double mult = A(i, j) / pivot;
           A[i].slice(j) = scale_and_add(A[j].slice(j),
                                         -mult, A[i].slice(j));
           b(i) -= mult * b(j); // b векторын өзгертеміз
       }
    }
 }
```

Тірек элемент деп, біз осы сәтте өңдеп жатқан, диагоналда орналасқан элементті айтамыз. Ол нөлге тең болмауы тиіс, өйткені біз оған сандарды бөлеміз, егер ол нөл болса, онда аластама туындайды:

```
Vector back_substitution(const Matrix& A, const Vector& b)
 {
    const Index n = A.dim1();
    Vector x(n);

    for (Index i = n-1; i >= 0; -- i) {
      double s=b(i)-dot_product(A[i].slice(i+1),
                                x.slice(i+1));

      if (double m = A(i, i))
          x(i) = s / m;
      else
          throw Back_subst_failure(i);
    }

    return x;
 }
```

24.6.2 Жетекші элементті таңдау

Нөлге тең диагонал элементтері мәселесі туындамас үшін және алгоритмнің тұрақтылығын көтеру мақсатында диагоналда нөлдер мен өте кіші шамалар болмайтындай етіп жолдарды ауыстыруға болады. "Тұрақтылығын көтеру" деп айтқанда, біз сандардың дөңгелектеу қателеріне сезімталдығын төмендетуді ойлап отырмыз. Бірақ алгоритмді орындау барысында матрица элементтері өзгереді, сондықтан жолдарды алмастыру әрекетін тұрақты түрде орындап отыру қажет болады (басқаша айтқанда, біз матрицаны бір-ақ рет реттеп алып, сонан соң классикалық алгоритмді қолдана алмаймыз).

```
void elim_with_partial_pivot(Matrix& A, Vector& b)
{
   const Index n = A.dim1();

   for (Index j = 0; j < n; ++j) {
        Index pivot_row = j;

       // сәйкес келетін тірек элементті іздейміз:
       for (Index k = j + 1; k < n; ++k)
       if(abs(A(k,j)) > abs(A(pivot_row,j))) pivot_row = j;

      // ең жақсы тірек элемент табылса, жолдарды алмастыру:
          if (pivot_row  != j) {
             A.swap_rows(j, pivot_row);
             std::swap(b(j), b(pivot_row));
          }

      // аластау (алып тастау):
          for (Index i = j + 1; i < n; ++i) {
             const double pivot = A(j, j);
             if (pivot==0) error("can't solve: pivot==0");
             const double mult = A(i, j)/pivot;
             A[i].slice(j) = scale_and_add(A[j].slice(j),
                                           mult, A[i].
             slice(j));
             b(i) - = mult * b(j);
          }
   }
}
```

Циклдерді тікелей түрде жазбас үшін және кодты дәстүрлі көрініске келтіру мақсатында `swap_rows()` және `scale_and_multiply()` функцияларын пайдаланамыз.

24.6.3 Тесттен өткізу

Әрине, біз программаны тесттен өткізуіміз керек. Қуанышымызға орай, оны істеу онша қиын емес:

```
void solve_random_system(Index n)
{
    Matrix A = random_matrix(n);        // 24.7 бөлімді қ.
    Vector b = random_vector(n);

    cout << "A = " << A << endl;
    cout << "b = " << b << endl;

    try{
      Vector x = classical_gaussian_elimination(A, b);
      cout << "classical elim solution is x=" << x << endl;
      Vector v = A * x;
      cout << " A * x = " << v << endl;
    }
    catch(const exception& e) {
      cerr << e.what() << std::endl;
    }
}
```

Мұндағы `catch` бөліміне үш түрлі себеппен барамыз:

- Программада қате бар (бірақ оптимистік тұрғыдан қарап, бұл ешқашан да болмайды деп санаймыз).
- Кіріс мәліметтері алгоритмнің `classical_elimination` апатты жұмыс жағдайына алып келеді (мұндайда `elim with partial pivot` функциясын пайдалану керек).
- Сандарды дөңгелектеу қатесі.

Дегенмен, тест біз ойлағандай, шынайы болмай шықты, өйткені кездейсоқ матрицалар `classical_elimination` алгоритмімен ешқандай мәселе туындата қоймайды.

Біздің шешімді тексеру үшін экранға `b` векторына тең болатын `A*x` көбейтіндісін (немесе дөңгелектеу қатесін есепке алғанда соған өте жақын болатын) шығарамыз. Мүмкін болатын дөңгелектеу қателеріне байланысты біз тек келесі нұсқаумен ғана шектеле алмаймыз:

```
if (A*x!=b) error("substitution failed");
```

Ондық нүктелі сандар нақты сандардың тек жуық шамалары болғандықтан, жуық нәтиже ғана аламыз. Негізінде, мына == және != операторларды ондық нүктелі есептеу нәтижелеріне пайдаланбаған дұрыс: мұндай сандар тек жуық мәндер болып табылады.

Matrix кітапханасында матрицаны векторға көбейтетін операция жоқ, сондықтан бұл функцияны өзіміз жазып шығуымыз керек:

```
Vector operator*(const Matrix& m,   const Vector& u)
{
      const Index n = m.diml();
      Vector v(n);
      for(Index i=0;i< n;++i) v(i)= dot_product(m[i],u);
      return v;
}
```

Matrix класының объектісімен орындалатын қарапайым операция біз үшін қайтадан үлкен жұмыс көлемін атқарады. 24.5.3 бөлімінде көрсетілгендей, Matrix класының объектілерін шығару операциялары MatrixIO.h тақырыбында сипатталған. randommatrix() және random_vector() функциялары жай ғана кездейсоқ сандарды пайдаланады (24.7 бөлім). Оқырмандар бұл функцияларды жаттығу ретінде жазып шыға алады. index атауы Matrix кітапханасында қолданылатын индекс типінің синонимі болып табылады, ол typedef операторы арқылы анықталған (A.15 бөлімі). Бұл тип программаға using жариялауы арқылы қосылады.

```
using Mumeric_lib::Index;
```

24.7 Кездейсоқ сандар

Егер сіз біреуден кездейсоқ сан сұрасаңыз, олар 7 немесе 17 санын айтады, өйткені бұлар нағыз кездейсоқ сандар болып табылады. Адамдар ешқашанда нөл туралы айтпайды, өйткені ол өте бір дөңгелек сан секілді болып көрінеді де, оны ешкім кездейсоқ сан деп қабылдамайды, сол себепті ол сирек кездесетін кездейсоқ сан болып саналады. Математикалық тұрғыдан алғанда, бұл толық түсініксіздік: жеке алынған бірде-бір санды кездейсоқ деп қарауға болмайды. Біздің кездейсоқ сандар деп айтатынымыз – олар белгілі бір таралу заңдылығына бағынатын сандар тізбегі, олардың алдыңғысы арқылы келесісін болжап айтуға болмайды. Мұндай сандар программаларды тесттен өткізу кезінде (олар көптеген тест алуға мүмкіндік береді), ойындарда (бұл ойынның келесі қадамы алдыңғысына сәйкес келмейтініне кепілдік беретін тәсілдердің бірі) және модельдеу есептерінде (біз өз параметрлерінің өзгеру аралығында өзін кездейсоқ түрде ұстайтын бір болмысты модельдей аламыз) өте пайдалы болып саналады.

Қазіргі кезде кездейсоқ сандар практикалық құрал және математикалық проблема ретінде күрделіліктің ең жоғарғы деңгейіне жетті деуге болады, сондықтан да олар нақты программаларда кең қолданыла бастады. Біз мұнда қарапайым тесттен өткізу мен модельдеуді жүзеге асыру үшін кездейсоқ сандар теориясының негіздерін ғана қарастырып өтеміз. Стандартты кітапхананың **<cstdlib>** тақырыбында мынадай код бар:

```
int rand();   //  [0:RAND_MAX] диапазонынан сандар береді
RAND_MAX      //  мүмкін болатын ең үлкен кездейсоқ сан
rand() void Brand(unsigned int);
//  берілетін кездейсоқ сандардың бастапқы мәні
```

rand() функциясын қайталап шақыру әрекеттері **[0:rand_max]** диапазонында бірқалыпты тарала орналасқан int типті сандарды береді. Мұндай сандар тізбегі жалған кездейсоқ сандар деп аталады, себебі олар математикалық формула арқылы белгілі бір орыннан бастап қайталана бастайды (яғни, болжауға болатын және кездейсоқ емес). Мысалы, егер біз программада **rand()** функциясын көп рет шақырсақ, онда біз әрбір шақырған сайын тек бір ғана сандар тізбегін қайталап алып отырамыз. Бұл программаны тексеріп түзету кезінде өте пайдалы. Егер де біз әртүрлі тізбектер алғымыз келсе, онда **srand()** функциясын әртүрлі аргументтер мәндері арқылы шақыруымыз керек. **srand()** функциясының әрбір жаңа аргументі үшін **rand()** функциясы да мәндері әртүрлі тізбектер береді.

Мысалы, 24.6.3 бөлімде айтылған **random_vector()** функциясын қарастырайық. **random_vector(n)** функциясын шақыру **[0:n]** диапазонынан алынған кездейсоқ **n** элементі бар **Matrix<double,1>** класы объектісін туындатады:

```
Vector random_vector(Index n)
 {
     Vector v(n);

     for (Index i = 0; i < n; ++i)
            v(i) = 1.0 * n * rand() / RAND_MAX;

     return v;
 }
```

Мұндағы **1.0** санының қолданылғанына назар аударыңыз, ол барлық есептеулер жылжымалы нүктелі арифметикада орындалатынына кепілдік береді. Әйтпесе, онда әрбір бүтін санды **rand_max** санына бөлген сайын **0** мәнін алар едік.

Берілген диапазоннан бүтін сан алу, мысалы **[0:max)** аралығынан, қиынырақ болады. Көптеген адамдар бірден келесі шешімді ұсынады:

```
int val  = rand()%max;
```

Ұзақ уақыт бойы мұндай код тіпті қанағаттанарлықсыз деп саналып жүрді, өйткені ол жай ғана кездейсоқ санның кіші разрядтарын алып тастайтын, ал оларда көбінесе дәстүрлі кездейсоқ сандар генераторлары беретін сандардың қасиеті болмайтын. Бірақ қазіргі кезде көптеген операциялық жүйелерде бұл мәселе жоғары деңгейде шешілген, дегенмен де өз программаларыңыздың ауысымдылығын қамтамасыз ету үшін біз кездейсоқ сандарды есептеуді мынандай функцияларда жасырып қоюды ұсынамыз.

```
int rand_int(int max){return rand() %max; }

int rand_int(int min,int max) {
    return rand_int (max-min)+min;}
```

Сонымен, егер `rand()` функциясының жүзеге асырылуы қанағаттанарлықсыз болатын болса, біз `rand_int()` функциясының анықталуын жасырып қоя аламыз. Сандардың бірыңғай таралып орналасуы қажет етілмейтін өндірістік программалық жүйелерде және де қосымшаларда сапалы және жеңіл қол жеткізуге болатын кездейсоқ сандар кітапханалары пайдаланылады, мысалы `Boost::random`. Біздің кездейсоқ сандар генераторының сапасы жайлы білгіңіз келсе, 10-жаттығуды орындаңыз.

24.8 Стандартты математикалық функциялар

Стандартты кітапханада көптеген математикалық стандартты функциялар (`cos`, `sin`, `log` және т.б.) бар. Олардың жариялануын `<cmath>` тақырып файлынан табуға болады.

Стандартты математикалық функциялар	
`abs(x)`	Абсолюттік мән
`ceil(x)`	Ең кіші бүтін сан, `>= x` шартын қанағаттандырады
`floor(x)`	Ең үлкен бүтін сан, `<= x` шартын қанағаттандырады удовлетворяющее условию `<= x`
`sqrt(x)`	Квадрат түбір; `x` мәні теріс сан болмауы тиіс
`cos(x)`	Косинус
`sin(x)`	Синус
`tan(x)`	Тангенс
`acos(x)`	Арккосинус; нәтиже теріс сан емес
`asin (x)`	Арксинус; нөлге ең жақын нәтиже қайтарады
`atan(x)`	Арктангенс

Стандартты математикалық функциялар (*жалғасы*)	
`sinh(x)`	Гиперболалық синус
`cosh(x)`	Гиперболалық косинус
`tanh(x)`	Гиперболалық тангенс
`exp(x)`	Экспонента
`log(x)`	Натурал логарифм, негізі *e* тұрақтысына тең; `x` мәні оң сан болуы тиіс
`log10(x)`	Ондық логарифм

Стандартты математикалық функциялар аргументтері **`float`**, **`double`**, **`long double`** және **`complex`** типтерінде бола алады (24.9 бөлім). Бұл функциялар жылжымалы нүктелі сандармен есептеулер орындағанда өте пайдалы болып табылады. Толығырақ ақпарат қолжетімді құжаттамаларда бар, алдымен веб-парақтардағы құжаттарды пайдалану керек.

Егер стандартты математикалық функция дұрыс нәтиже бере алмайтын болса, ол **`errno`** жалаушасын орнатады. Мысал қарастырайық:

```
errno = 0;
double s2 = sqrt(- 1);
if(errno) cerr << "бір жерде бірдеңе дұрыс орындалмады";
if(errno == EDOM)
// аргумент мәнінің анықталу шегінен тысқары кету қатесі
     cerr << "sqrt()функциясы тегіс
                          аргументтер үшін анықталмаған";
pow(very_large,2);        // жақсы идея емес
if (errno==ERANGE)
// берілген диапазоннан шығып кету қатесі
     cerr << "pow(" << very_large << ",2)
                               double үшін өте үлкен сан";
```

Егер сіз салмақты математикалық есептеулер орындап жатсаңыз, онда нәтижені қайтарғаннан кейін оның бұрынғыша **`нөлге`** тең екеніне сенімді болу үшін, әрқашанда **`errno`** мәнін тексеріп шығуыңыз керек. Егер олай болмаса, бір жерде бір әрекеттердің дұрыс орындалмай кеткенінің белгісі болып саналады. Математикалық функциялардың **`errno`** жалаушасын қалай орнататынын білу үшін және оның неге тең болатынын анықтау үшін құжаттамаларды қарап шығу керек.

Мысалда көрсетілгендей, **`errno`** жалаушасының мәні нөл болмағаны кейбір әрекеттердің дұрыс орындалмағанын көрсетеді. Стандартты кітапханаға кірмейтін функциялар да қате тексеру кезінде **`errno`** жалаушасын көтеріп

кетеді, сондықтан не болғанын толығырақ білу үшін **errno** айнымалысының әртүрлі мәндерін анығырақ талдау керек. Бұл мысалда стандартты кітапханалық функцияны шақырғанша, **errno** айнымалысы **нөлге** тең болған, ал **errno** мәнін бірден функция орындалғаннан кейін тексеріп анықтау, мысалы, **edom** және **herange** константалары арқылы орындалады. **edom** константасы аргументтің функция анықталу аймағынан шығып кеткенін көрсетеді (domain error). **erange** константасы мәндердің берілген диапазоннан шығып кеткенін көрсетеді (range error).

errno айнымалысы арқылы қателерді өңдеу қарапайым ғана әрекет болып саналады. Оның түп-тамыры C тілі математикалық функцияларының бірінші нұсқасынан (1975 жылғы шығарылғаны) бастау алған болатын.

24.9. Комплекс сандар

Комплекс сандар ғылыми және инженерлік есептеулерде кеңінен қолданылады. Олар сізге керек болып жатса, онда олардың математикалық қасиеттері де сізге белгілі шығар деп ойлаймыз, сондықтан біз жай ғана комплекс сандардың C++ тілі стандартты кітапханасында қалай өрнектелетінін көрсетеміз. Комплекс сандардың және солармен байланысты математикалық функциялардың жариялануы **<complex>** тақырыбында орналасқан.

```
template<class Scalar> class complex {
//  комплекс сан – бұл екі скалярлық шама,
//  негізінде,  екі координата
Scalar re, im;
public:
    complex(const Scalar & r,const Scalar & i) im(r),im(i){ }
    complex (const Scalar & r) :re(r),im(Scalar ()) { }
    complex() :re(Scalar ()), im(Scalar ()) { }

    Scalar real() { return re; }  //  real part
    Scalar imag() { return im; }  //  imaginary part

    //    = += -= *= /= операторлары
};
```

complex стандартты кітапханасы скалярлық шамалардың **float**, **double** және **long double** сияқты типтерін сүйемелдейді. **<complex>** тақырыбында **complex** класының мүшелерінен және стандартты математикалық функциялардан (24.8 бөлім) басқа көптеген пайдалы операциялар бар.

Комплекс сандармен орындалатын операторлар	
`z1+z2`	Қосу
`z1-z2`	Азайту
`z1*z2`	Көбейту
`z1/z2`	Бөлу
`z1=z2`	Теңдікті тексеру
`z1!=z2`	Теңсіздікті тексеру
`norm(z)`	`abs(z)` квадраты
`conj(z)`	Араласқан сандар: егер `z {re, im}` мәніне тең болса, онда `conj(z) (re,-im)` мәніне тең болады
`polar(x,y)`	Полярлық координаттарға ауыстыру (*rho*, *theta*)
`real(z)`	Нақты бөлігі
`imag(z)`	Жалған бөлігі
`abs(z)`	Модуль, *rho*
`arg(z)`	Аргумент, *theta*
`out << z`	Комплекс санды шығару
`in >> z`	Комплекс санды енгізу

Ескерту: `complex` класында `<` және `%` операциялары жоқ.

`complex<T>` класы кез келген басқа бір құрамдас тип тәрізді пайдаланылады, мысалы, `double`. Мысал қарастырайық:

```
typedef complex<double> dcmplx;
//кейде complex<double> өрнегі тым көлемді болып табылады
void f(dcmplx z,vector<dcmplx>& vc)
{
    dcmplx z2 m pow(z,2);
    dcmplx z3 = z2*9.3+vc[3];
    dcmplx sum = accumulate(vc.begin(),vc.end(),dcmplx());
    // ...
}
```

`complex` класында `int` және `double` типтеріндегі сандар үшін барлық операциялар анықталмағандығын есте сақтаңыз. Мысал қарастырайық:

```
if (z2<z3)
// қате:  < операциясы комплекс сандар үшін анықталмаған
```

C++ тілінің стандартты кітапханасында комплекс сандарды бейнелеуді (сұлбасын) C және Fortran тілдеріндегі осыған сәйкес типтермен салыстыруға болатындығына назар аударыңыз.

24.10 Сілтемелер

Негізінде, бұл тарауда көтерілген сұрақтар, мысалы, дөңгелектеу қателері, матрицалармен орындалатын операциялар және комплекс сандар арифметикасы сияқты тақырыптардың жеке өздері бізді онша қызықтырмайды. Біз жай ғана C++ тілі ұсынатын кейбір мүмкіндіктерді математикалық есептеулерді орындайтын адамдарға арнап сипаттаймыз.

Егер сіз математиканы аздап ұмытқан болсаңыз, онда келесі ақпарат көздеріне назар аударыңыз:

http: //www-gap. dcs.st-and.ac.uk/~history веб-парағында орналасқан MacTutor History of Mathematics архиві.

- Математиканы жақсы көретіндердің барлығына немесе оны пайдаланғысы келетіндерге арналған өте жақсы веб-парақ.
- Математиканың гуманитарлық аспектісін көргісі келетіндерге арналған өте жақсы веб-парақ, мысалы, ірі математиктердің қайсысы Олимпиадалық ойындарды жеңген еді?
    - Атақты математиктер: биографиялары, табыстары.
    - Таңғаларлық жағдайлар (курьездер).
- Көрнекі қисық сызықтар.
- Белгілі есептер.
- Математикалық тақырыптар.
    - Алгебра.
    - Талдау (анализ).
    - Сандар теориясы.
    - Геометрия және топология.
    - Математикалық физика.
    - Математикалық астрономия.
        - Математика тарихы, көптеген т.б.

Freeman, T. L., and Chris Phillips. *Parallel Numerical Algorithms.* Prentice Hall, 1992.

Gullberg, Jan. *Mathematics – From the Birth of Numbers.* W. W. Norton, 1996. ISBN 039304002X. Математиканың негізі мен пайдасы туралы ең қызықты кітаптардың бірі, оны бір жағынан пайдалы мәлімет (мысалы, матрицалар жайлы) алу үшін, екінші жағынан құмарлана оқып шығу үшін алуға болады.

Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition.* Addison-Wesley, 1998. ISBN: 0202496842.

Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions.* SIAM, 1998. ISBN 0898714141.

Wood, Alistair. *Introduction to Numerical Analysis.* Addison-Wesley, 1999. ISBN 02019429IX.

ТАПСЫРМА

1. Экранға мынадай типтердің **char**, **short**, **int**, **long**, **float**, **double**, **int*** және **double*** мөлшерін (енін) шығарыңыз (**<limits>** тақырыбын емес, **sizeof** операторын пайдаланыңыз).
2. **sizeof** операторын пайдалана отырып, экранға мынадай объектілердің **Matrix <int> a(10),Matrix<int> b(10), Matrix<double> c(10), Matrix<int,2> d(10,10), Matrix<int,3> e(10,10,10)** мөлшерін, яғни енін шығарыңыз.
3. 2-жаттығуда тізіп көрсетілген объектілердің әрқайсысындағы элементтер санын баспаға шығарыңыз.
4. **cin** ағымынан **int** типіндегі сандарды енгізетін программа жазыңыз да, осы **int** сандарының әрқайсысына **sqrt()** функциясын қолдану нәтижесін анықтаңыз. Егер **sqrt(x)** функциясын кейбір **x** мәндеріне қолдануға болмайтын болса, экранға "санның квадрат түбірі жоқ" (яғни, **sqrt()** функциясы қайтаратын мәндерді тексеріңіз) деген мәлімет шығарыңыз.
5. Енгізу ағымынан жылжымалы нүктелі он санды оқып, оларды **Matrix<double>** типіндегі объектіге жазыңыз. **Matrix** класында **push_back()** функциясы жоқ, сондықтан сақ болыңыз және **double** типіндегі сандардың қате санын енгізу әрекетіне жауап беретін мүмкіндікті қарастырыңыз. Осы **Matrix** класының объектісін экранға шығарыңыз.
6. **[0,n)*[0,m)** көбейту кестесін есептеңіз де, оны **Matrix** класының екіөлшемді объектісі ретінде бейнелеңіз. **cin** ағымынан **n** және **m** сандарын енгізіңіз де, алынған кестені экранға мұқият түрде шығарыңыз (**m** өте шағын сан болып есептеледі, сондықтан жұмыс нәтижесі экранның бір жолына толығынан сыяды деп саналады).
7. **cin** ағымынан **complex<double>** класының он объектісін (иә, **cin** класы **complex** типі үшін **>>** операторын сүйемелдейді) енгізіңіз де, оны **Matrix** класының объектісіне орналастырыңыз. Сонан соң он комплекс матрицаның қосындысын есептеп, экранға шығарыңыз.
8. **int** типіндегі алты санды **Matrix<int,2> m(2,3)** класының объектісіне жазып шығыңыз да, оларды экранға шығарыңыз.

БАҚЫЛАУ СҰРАҚТАРЫ

1. Сандық есептеулерді кім орындайды?
2. Дәлдік дегеніміз не?
3. Толып кету деген не?
4. `double` мен `int` типтерінің қалыптағы мөлшерлері (ендері) қандай?
5. Толып кетуді қалай анықтауға болады?
6. Сандардың өзгеру шектерін қалай анықтауға болады, мысалы, `int` типіндегі ең үлкен сан үшін?
7. Жиым, жол және бағана деген не?
8. C тілі стиліндегі көпөлшемді жиым деген не?
9. Матрицалық есептеулерді орындау үшін программалау тілінің қандай қасиеттері (мысалы, кітапхана болуы керек) болуы тиіс?
10. Матрицаның өлшемі деген не?
11. Матрицаның неше өлшемі болуы тиіс?
12. Қима деген не?
13. Қайта жіберу (пересылка) деген не? Мысал келтіріңіз.
14. Fortran және C тілдері стиліндегі индекстеудің қандай айырмашылықтары бар?
15. Матрицаның әрбір элементіне кез келген операцияны қалай қолдануға болады? Мысал келтіріңіз.
16. Біріктірілген көбейту және қосу (fused operation) деген не?
17. *Скалярлық көбейту* ұғымына анықтама беріңіз.
18. Сызықтық алгебра деген не?
19. Гаусс аластау әдісін сипаттаңыз.
20. Тірек элемент (сызықтық алгебрада және нақты өмірде) элемент деген не?
21. Санды кездейсоқ ететін не?
22. Бірқалыпты таралу деген не?
23. Стандартты математикалық функцияларды қайдан табуға болады? Олар аргументтердің қандай типтері үшін анықталған?
24. Комплекс санның жалған бөлігі деген не?
25. -1 санының квадрат түбірі неге тең болады?

ТЕРМИНДЕР

C тілі	жалған бөлігі	қима
`errno`	жиым	нақты сан
Fortran тілі	жол	мөлшер
`Matrix`	индекстеу	өлшем
`sizeof`	кездейсоқ сан	скалярлық көбейту
бағана	комплекс сан	элементтер бойынша операциялар
бірқалыпты таралу	көпөлшемді	
біріктірілген қосу және көбейту	қайта есептеу	

ЖАТТЫҒУЛАР

1. `f` функциясының аргументтері `a.apply(f)` және `apply(f,a)` өрнектерінде әртүрлі болып табылады. Әрбір нұсқа үшін `double()` функциясын жазыңыз да, оларды жиымның мынадай `{1 2 3 4 5}` элементтерін екі еселеу үшін қолданыңыз. `a.apply(f)` және `apply (f,a)` өрнектерінде де қатар қолдануға болатын жеке `double()` функциясын анықтаңыз. Барлық функцияларды да `apply(f,a)` функциясының аргументі ретінде қолдану үшін осылай жазудың неге ыңғайлы болатынын түсіндіріңіз.
2. 1-жаттығуды функцияны емес, объект-функцияны пайдалана отырып, қайта орындаңыз.
3. Тек сарапшылар үшін (кітапта сипатталған құралдар арқылы бұл есепті шығаруға болмайды). Аргумент ретінде `void 9T&),T (const T&)` функциясын қабылдайтын `apply(f,a)` функциясын жазыңыз және де соларға парапар (эквивалентті) объект-функция жазыңыз. Көмекші мәлімет: `Boost::bind`
4. Гаусс аластама тәсілінің программасын орындап шығыңыз, яғни оны аяқтап, компиляциядан өткізіп қарапайым мысалда тесттен өткізіңіз.
5. Гаусс аластама тәсілінің программасын мынадай `A=={ {0 1}{ 1 0 }` және `b=={ 5 6 }` жүйелерге қолданыңыз да, программаның апатты жағдаймен аяқталатынына көз жеткізіңіз. Сонан соң `elim_with_ partial_pivot()` функциясын шақыруға тырысыңыз.
6. Гаусс аластама тәсілінің программасында `dot_product()` және `scale_ and _add()` векторлық операцияларын циклмен алмастырыңыз. Бұл программаны тесттен өткізіңіз де, түсініктеме беріңіз.

7. Гаусс аластама тәсілінің программасын `Matrix` кітапханасы көмегінсіз жазып шығыңыз. Басқаша айтқанда, `Matrix` класын емес, құрамдас жиымды немесе `vector` класын пайдаланыңыз.
8. Аластама тәсілін Гаусс тәсілімен көрсетіңіз.
9. `Matrix` класының мүшесі болып табылмайтын `apply()` функциясын сол функцияда қолданылған типтегі объектілері бар `Matrix` класының объектісін қайтаратындай етіп қайта жазып шығыңыз. Басқаша айтқанда, `apply(f,a)` функциясы `Matrix <R>` класының объектісін қайтаруы тиіс, мұндағы `R` – `f` функциясы қайтаратын мән типі. Ескерту: бұл шешімде кітапта айтылмаған шаблондар туралы ақпарат талап етіледі.
10. `rand()` функциясы қандай дәрежеде кездейсоқ болып саналады? Енгізу ағымынан `n` және `d` сияқты екі бүтін санды қабылдап алып, `randint(n)` функциясын `d` рет шақырып, нәтижесін жазатын программа жазыңыз. Экранға әрбір `[0:n)` диапазонына түскен сандар санын шығарып, олардың саны қаншалықты тұрақты болатынын бағалауға тырысыңыз. Мәндердің ауқымдылығы тәжірибелер саны көп болмаған кездерде ғана туындайтынына сенімді болу үшін, программаны онша үлкен емес шағын `n` және `d` мәндерімен орындап шығыңыз.
11. 11.24.5.3 бөліміндегі `swaprows()` функциясына ұқсас `swap_columns()` функциясын жазып шығыңыз. Әрине, ол үшін `Matrix` кітапханасы кодын оқып үйрену керек. Өз программаңыздың тиімділігі жайлы қам жемеңіз: негізінде, `swap_columns()` функциясының жылдамдығы `swap_rows()` функциясының жылдамдығынан жоғары бола алмайды.
12. Келесі операторларды жүзеге асырыңыз:

```
Matrix<double> operator*
(Matrix<double,2>&,Matrix<double>&);
```

және

```
Matrix<double,N> operator+
(Matrix<double,N>&,Matrix<double,N>&)
```

Қажет болса, бұлардың оқулықтардағы математикалық анықтауларын қарап шығыңыз.

СОҢҒЫ СӨЗ

Егер сіз математиканы жақсы көрмесеңіз, онда, мүмкін, бұл тарау сізге ұнамаған болар, сіз өзіңізге жоғарыда баяндалған ақпарат қажет болмайтын қосымшалар аймағын таңдайтын шығарсыз. Басқа жағынан алсақ, егер сіз математиканы жақсы көрсеңіз, онда біз келтірген кодтардағы математикалық түсініктерді өрнектеу дәлдігін бағалайтын шығарсыз деп үміттенеміз.

25

Құрамдас жүйелерді программалау

" 'Қауіпті' сөзі әлдекімнің өмірден өтуінің мүмкін екенін білдіреді".

- Қауіпсіздік мекемесінің қызметкері

Бұл тарауда біз құрамдас жүйелерді программалау мәселелерін қарастырамыз, басқаша айтқанда, экраны мен пернетақтасы бар дәстүрлі компьютер үшін емес, алдымен құрылғылар үшін программа жазуға байланысты тақырыптарды талқылаймыз. Басты назарды осындай құрылғыларды программалаудың қағидалары мен әдістеріне, аппараттық жабдықтамамен тікелей жұмыс істеу үшін қажетті кодтау стандарттарының тілдік мүмкіндіктеріне аударған жөн. Айтылған тақырыптарға ресурстарды және компьютер жадын басқару, нұсқауыштар мен жиымдарды пайдалану, сонымен қатар биттерді өңдей отырып жұмыс істеу (манипуляциялау) істері жатады. Мұндағы басты акцент мәліметтерді қауіпсіз түрде пайдалануға, сонымен қоса төменгі деңгейлі құралдарды баламалы тәсілмен қолдануға беріледі. Біз құрылғылардың арнайы архитектурасын сипаттауға немесе аппараттық жабдықтама жадына тікелей қол жеткізу әдістерін сипаттауға талпынбаймыз, ол үшін арнайы әдебиеттер бар. Көрсетілім ретінде біз кодтау – қайта кодтау алгоритмінің жүзеге асырылуын таңдадық.

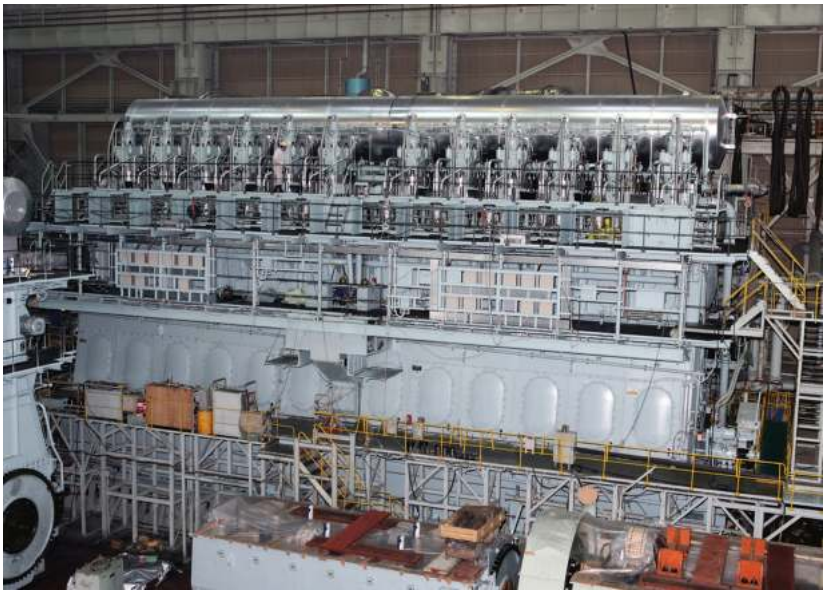
25.1 Құрамдас жүйелер

Қазіргі компьютерлердің көбісі компьютер сияқты болып көрінбейді. Олар басқа бір ірі жүйелердің немесе құрылғылардың жай ғана бөлігі болып табылады. Мысалдар қарастырайық.

- *Автомобильдер.* Заманауи автомобильге ондаған компьютерді: жанармай беруді басқаратын, қозғалтқыштың жұмысын қадағалайтын, радионы баптайтын, тежеуішті бақылайтын, шинадағы қысымды тексеріп отыратын, алдыңғы әйнек тазалағышты басқаратын және т.с.с. кірістіріп қоюға болады.
- *Телефондар.* Ұялы телефон ең аз дегеннің өзінде екі компьютерден тұрады; олардың біреуі әдетте, сигналдарды өңдеуге арналады.
- *Ұшақтар.* Заманауи ұшақ жұмыстың барлығын да басқаратын: жолаушылардың көңілін көтеру жүйесінен бастап, қанаттарының көтерілу күшін оңтайландыратын жапқыштарға дейінгі компьютерлермен жабдықталған.

- *Фотоаппараттар*. Бес процессоры бар фотоаппараттар бар, олардағы әрбір объективтің өзі жеке процессормен жабдықталған.
- *Несие кәртішкелері* (кәртішкелердің барлығының да микропроцессорлары бар).
- *Медицина құрал-жабдықтарының мониторлары мен контроллерлері* (мысалы, компьютерлік томографияға арналған сканер).
- *Жүккөтергіштер* (лифттер).
- *Қалта компьютерлері*.
- *Ас-үй жабдықтары* (мысалы, жылдамдатып ас пісіретін және нан жабатын пештер).
- *Телефон коммутаторлары* (көбінесе, мыңдаған арнайы компьютерлерден тұрады).
- *Сорғыштар контроллерлері* (мысалы, су немесе мұнай сорғыштар).
- Дәнекерлеу роботтары, олар адамдар жұмыс істей алмайтын қауіпті жерлер мен оңайлықпен қол жеткізуге болмайтын орындарда жұмыс істейді.
- *Жел қозғағыштар*. Олардың кейбіреуі мегаваттаған электр энергиясын өндіреді және 70 метрге дейінгі биіктікте орналасады.
- *Бөгеттердегі шлюздер контроллерлері*.
- *Конвейерлердегі сапа мониторлары*.
- *Штрих кодтарды оқитын құрылғылар*.
- *Автожинақтауыш роботтар*.
- *Центрифуга контроллерлері* (көптеген медициналық сараптама үдерістерінде пайдаланылады).
- *Дискіқозғалтқыштар контроллерлері.*

Бұл компьютерлер аса ірі жүйелердің бір бөлігі болып табылады, олар әдетте компьютерлерге ұқсамайды және біз ешқашанда оларды компьютер деп ойламаймыз. Мысалы, біз көшеден өтіп бара жатқан автомобильді көргенде: "Қараңыз, үлестірмелі компьютерлік жүйе кетіп барады!" деп айтпаймыз. Бірақ автомобиль де үлестірмелі компьютерлік жүйенің бір түрі болып табылады, оның әрекеттері механикалық, электрондық және электр жүйелерінің жұмысымен өте тығыз байланысқан, біз оны оқшауланған компьютер деп санай алмаймыз. Осы жүйенің жұмысына қойылатын шектеулер (уақытша және кеңістіктегі) мен оның программаларының дұрыстығы түсінігін ол құрамына кіретін ірі жүйеден бөліп алу мүмкін емес. Құрамдас компьютер көбінесе, физикалық құрылғыны басқарады және компьютердің дұрыс жұмыс істеу тәртібі сол физикалық құрылғының дұрыс жұмыс атқару белгісі ретінде анықталады. Кеме жүргізетін ірі дизельдік қозғалтқышты қарастырайық.



Адам тұрып басқаратын бесінші цилиндрдің қақпағына назар аударыңыз. Бұл – үлкен кемені жүргізетін ірі қозғалтқыш. Егер осындай қозғалтқыш істен шығатын болса, біз ол туралы таңғы жаңалықтардан естиміз. Осындай қозғалтқыштың әрбір цилиндрінің қақпағында үш компьютерден тұратын цилиндрді басқару жүйесі бар. Әрбір цилиндрді басқару жүйесі жалпы қозғалтқышты басқару жүйесімен (тағы үш компьютер) екі тәуелсіз желі арқылы байланысқан. Бұған қоса, қозғалтқышты басқару жүйесі сондағы арнайы графикалық интерфейс арқылы қозғалтқышқа механиктер команда бере алатындай түрде басқару орталығымен қосылған. Осы жүйені теңіздегі қозғалысты басқаратын орталықтан радиосигналдар көмегімен (спутниктер арқылы) қашықтан бақылауға болады. Компьютерлерді пайдаланудың басқа мысалдары 1-тарауда келтірілген.

Сонымен, программалаушының көзқарасы бойынша осындай компьютерлерде орындалатын программалардың қандай ерекшеліктері бар? Сұрақты жалпы түрде қояйық: "қарапайым" программалардағы бізді мазаламайтын қандай мәселелер құрамдас жүйелерде алдыңғы сапқа шығады?

- *Көбінесе сыни тұрғыдан алғанда маңызды болып сенімділік есептеледі.* Одан бас тарту ауыр жағдайларға: үлкен шығындарға (миллиардтаған долларлар) және мүмкін біреудің өлім-жітіміне әкелуі мүмкін (апатқа ұшыраған кемеге мінген адамдар немесе теңіз суларына төгіліп кеткен жанармайлардың кесірінен қырылған жан-жануарлар).
- *Көбінесе ресурстар (компьютер жады, процессордың циклдері, қуаты) шектелген болады.* Қозғалтқышты басқаратын компьютер үшін, бәлкім, бұл үлкен мәселе емес, бірақ ұялы телефондар үшін, сенсорлар, қалта компьютерлері, ғарыш (космос) зондындағы компьютерлер және

басқалар үшін бұл маңызды болып табылады. Әлемде 2 Гц жиілігі бар екі процессорлық портативті компьютер сирек емес, ұшақтың немесе ғарыш зондының жұмысында басты рөл атқаратын процессорының жиілігі 60 Гц және жады көлемі 256 Кбайт компьютерлер және тіпті, жиілігі 1 Гц-тен төмен және оперативті жады көлемі бірнеше жүздеген сөздермен ғана өлшенетін кішкентай құрылғылар орындауы мүмкін. Сыртқы әсерлерге (вибрацияларға, соққыларға, электр энергиясын тұрақты түрде бермеуге, ыстыққа, суыққа, ылғалға, оны таптауға және т.б.) төзімді компьютерлер әдетте студенттердің ноутбугына қарағанда, өте баяу жұмыс істейді.

- *Көбінесе нақты уақыт кезеңінде әсер ету маңызды болып табылады.* Егер отынның инжекторы инъекционды циклге түспесе, онда 100 мың ат күші бар аса күрделі жүйемен жұмыс істеуде қиындық туындауы мүмкін; егер инжектор бірнеше циклді өткізіп жіберетін болса, яғни бір секундтай уақытта дұрыс жұмыс істемесе, онда диаметрі 10 метр және салмағы 130 тонна болатын пропеллермен қызықты нәрселер орын алуы мүмкін. Біз осылардың болмауын өте қалар едік.
- *Көбінесе жүйе көп жылдар бойы тоқтаусыз жұмыс істеуі қажет.* Бұл жүйелер, мысалы, орбитаны айналып жүрген байланыс жолсерігі сияқты қымбат болуы мүмкін немесе оларды жөндеу ештеңеге тұрмайтындай тым арзан болуы да мүмкін (мысалы, MP3-плеерлер, несие кәртішкелері немесе автомобиль қозғалтқышының инжекторлары). АҚШ-та телефон коммутаторларының сенімділік критерийі ретінде жиырма жыл ішінде жиырма минут қана жұмыс істемей тұрып қалуы алынады (тіпті, оның программасын өзгерткіңіз келгенде де, оны қайта шашып құрастырудың қажеті жоқ).
- *Көбінесе жөндеу жұмысын (ремонт) жүргізу мүмкін емес немесе ол өте сирек болады.* Сіз кемелерді олардың компьютерін немесе басқа да жүйелерін жөндеу үшін екі жыл сайын тұраққа (гавань) әкеліп, компьютерлік мамандардың қажетті уақытта қажетті орындарда болуын жабдықтамасыз ете аласыз. Бірақ көбінесе жоспарланбаған жөндеу жұмысын орындау мүмкін емес (егер кеме Тынық мұхитының ортасында болғанда, дауыл (шторм) тұрса, онда программадағы қателер келеңсіз рөл атқаруы мүмкін). Сіз Марсты айналып орбитада жүрген ғарыштық зондты жөндеу үшін біреуді жібере алмайсыз.

Кейбір жүйелер жоғарыда көрсетілген көптеген шектеулерге тап болады, ал кейбіреулерінде – оның тек біреуі ғана кездеседі. Бұл – нақты бір қолданбалы аймақтағы сарапшылардың жұмысы. Біздің мақсатымыз – сізден осы сұрақтарды талдайтын сарапшы жасап шығару емес, ондай іс ақылсыздық немесе жауапсыздық болар еді. Біздің мақсатымыз – сіздерді солардың шешіміне байланысты негізгі мәселелермен және концепциялармен таныстырып, осындай жүйелерді құру

кезінде өзіңізге қажетті дағдылардың қиындығын бағалау болып табылады. Мүмкін, сіз одан да тереңірек білім алғыңыз келетін шығар. Құрамдас жүйелерді құратын және іске асыратын адамдар техникалық өркениетті дамыту ісінде маңызды рөл атқарады. Бұл – кәсіби мамандардың үлкен жетістіктерге жететін аймағы.

Осылар C++ тіліндегі программалаушыларға және де жаңадан үйреніп жүргендерге қатысты ма? Иә және тағы да иә. Құрамдас жүйелер қарапайым дербес компьютерлерге қарағанда анағұрлым көп. Программалау жұмысының басым бөлігі осы құрамдас жүйелерді программалау арқылы орындалады. Оның үстіне, осы тараудың басында көрсетілген құрамдас жүйелердің мысалдары, менің C++ тілінде программалаудағы жеке тәжірибелерім негізінде құрастырылған болатын.

25.2 Негізгі түсініктер

Құрамдас жүйелердің ішкі бөлігі болып табылатын компьютерлердегі программалаудың басым бөлігінің қарапайым программалаудан ешқандай айырмасы жоқ, сондықтан оларға осы кітапта тұжырымдалған идеялардың көбін қолдануға болады. Бірақ мұндағы акцент басқада: біз программалау тілінің құралдарын шығарылатын есептердің шектеулерін ескеретіндей етіп бейімдеуіміз қажет және де көбінесе ең төменгі деңгейдегі аппараттық жабдықтамамен жұмыс жасауымыз керек.

- *Дұрыстығы.* Бұл түсінік әдеттегіден де өте маңызды болады. Дұрыстық – бұл жай ғана абстрактілі түсінік емес. Құрамдас жүйенің контексінде программа жай ғана дұрыс жауап берген кезде ғана дұрыс деп саналмайды, ол оны берілген тәртіппен, көрсетілген уақытта және тек қолда бар ресурстар жиынтығын пайдаланып жасалған кезде ғана дұрыс болып саналады. Негізінде, *дұрыстығы* түсінігінің бүге-шігесі әрбір нақты жағдайларға байланысты тиянақты түрде тұжырымдалады, бірақ та осындай спецификацияны көбінесе бірсыпыра тәжірибелерден кейін ғана құруға болады. Дегенмен, маңызды тәжірибелерді жүйе толық құрылғаннан кейін ғана (программа орындалатын компьютермен бірге) жүргізуге болады. Құрамдас жүйенің дұрыстығы түсінігінің толық тұжырымдамасы өте қиын да және сонымен қатар аса маңызды да болуы мүмкін. "Өте қиын" сөзі "бөлінген уақытта және берілген ресурстармен орындау мүмкін емес" дегенді білдіруі мүмкін; біз қолда бар құралдар мен әдістер көмегімен мүмкіндігінше жасауға болатынның барлығын да орындауға міндеттіміз. Қуанышқа орай, мұндағы спецификациялардың, модельдеу әдістері мен тесттен өткізу мүмкіндіктерінің және де берілген аймақтағы басқа да технологиялар санының өте көп болуы мүмкін. "Аса маңызды" сөзі жұмыс барысындағы "ақаулық зиян келтіруге немесе бүлінуге әкеледі" дегенді білдіруі мүмкін.

- *Ақауларға тұрақтылығы.* Біз программа қанағаттандыратын шарттар жиынын мұқият көрсетуіміз керек. Мысалы, студенттік программаны өткізу кезінде, егер мұғалім қоректену сымын токтан ажыратып тастаса, сіз мұны дұрыс емес теріс жағдай деп есептей аласыз. Электр тогының өшіп қалуы дербес компьютерлердегі қолданбалы программалардың жауап беретін шарттар тізіміне кірмейді. Бірақ құрамдас жүйелердегі энергияның өшіп қалуы әдеттегі жағдай ретінде қарастырылады, мұнда сіздің программаңыз оны есепке қажет. Мысалы, жүйенің өмір сүруге керекті маңызды бөліктері екі электр көзінен қатар қоректене алады, резервтегі батареялары болады, т.с.с. Кейбір қосымша программалар үшін мынадай түрде "Мен аппараттық жабдықтама тоқтаусыз (ақаусыз) жұмыс жасайды деп ойладым" деп айтылған сөз ақталу болып саналмайды. Жиі өзгертіліп жататын шарттар жағдайында аппараттық жабдықтаманың ұзақ уақыт бойы тоқтаусыз жұмыс істеуі мүмкін емес нәрсе. Мысалы, кейбір телефон коммутаторларына және аэроғарыш аппараттарына арналған программаларда ерте ме, кеш пе әйтеуір бір кездерде компьютер жадындағы мәліметтердің бірсыпыра бөліктері қайта жасалып, оның мазмұны өзгертілетініне (мысалы, нөлді бірге ауыстыру сияқты) алдын ала "шешім" жасалып қойылған. Оған қоса, компьютер өзінің бірлікті ұнататынын көрсетіп, оны нөлге алмастыруға "қарсылық білдіруі" де мүмкін. Егер сіздің компьютеріңіздің жады жеткілікті көлемде аса үлкен болса және сіз оны ұзақ уақыт бойы үздіксіз пайдаланған болсаңыз, онда ақырында одан да қате шығуы мүмкін. Егер компьютердің жады жер шары атмосферасы шегінен тысқары жерде радиациялық сәулелендіруге душар болатын болса, онда ол одан едәуір ерте істен шығады. Біз жүйемен (құрамдас немесе басқаша) жұмыс жасау барысында, жабдықтар істен шыққан жағдайда онымен не істеу керек екендігін шешіп алуымыз қажет. Әдетте, (үнсіз) келісім бойынша, аппараттық жабдықтама тоқтамай жұмыс жасайды деп саналады. Егер де біз жоғары талап қойылатын жүйелермен жұмыс істейтін болсақ, онда мұндай жағдайларды алдын ала айқындап алған жөн.

- *Тұрып қалудың болмауы.* Әдетте, құрамдас жүйелер программалық жабдықтаманы ауыстырмай немесе тәжірибелі оператор кіріспегенше ұзақ жұмыс істеуі мүмкін. "Ұзақ уақыт" деген сөз күндерді, айларды, жылдарды немесе аппаратық жабдықтаманың жұмыс істеген барлық уақытын білдіреді. Бұл жағдай құрамдас жүйелерге тән қасиет, бірақ "қарапайым қосымшалардың" барлығына бірдей қатысты емес, сонымен қатар, кітапта келтірілген мысалдар мен жаттығулардың да барлығына қолданылмайды. "Мәңгі жұмыс істеуі керек" деген талап алдыңғы жаққа қателерді өңдеу мен ресурстарды басқару ісін қояды. "Ресурс" дегеніміз не? Ресурс – машина құрамында шектелген көлемде болатын кез келген бір зат; программа ресурсты тікелей әрекет орындау арқылы алуы мүмкін (бос жады көлемін бөлу) және оны пайдаланып болған соң, жүйеге (жадыны босату) тікелей немесе жанамалы (тікелей емес) түрде қайтару керек. Ресурстардың мыса-

лы болып жады аймағы, файлдардың дескрипторлары, желілік байланыстар (сокеттер) және бұғаттау істері саналады. Үздіксіз (ұзақ уақыт бойы) жұмыс істейтін жүйенің бөлігі болып табылатын программа, оған тұрақты қажет болатын өз ресурстарынан басқасын пайдаланып болған соң, босатып отыруы қажет. Мысалы, көптеген операциялық жүйелерде, файлды күн сайын жабуды ұмытып кетіп отыратын программа бір айдан артық жұмыс істей алмайды. Күн сайын 100 байт жады көлемін босатпайтын программа, жылына 32 Кбайт жады көлемін тауысады, бұл бірнеше айдан кейін шағын бір құрылғының жұмыс істемеуіне алып келуі мүмкін. Ресурстардың осындай түрде "азаюының" ең жаманы – мұндай программалар көптеген айлар бойы жақсы жұмыс жасайды да, кейіннен күтпеген жерден тоқтап қалады. Сонымен, егер де программа бәрі бір тоқтайтын болса, онда мұндай жағдайдың ертерек ұшырасып, біздің оны жөндеп түзетуге уақытымыздың жеткілікті болғанын қалар едік. Сонымен, программа қолданушыға жеткенше, ақаулардың ертерек анықталғаны дұрыс болар еді.

- *Нақты уақыт шектеулері.* Құрамдас жүйені, егер ол әрқашанда берілген уақыт мерзіміне дейін жауап беретіндей түрде болса, оны *нақты уақыт кезеңінің қатаң шарттары бар* (hard real time) жүйелерге жатқызуға болады. Егер ол берілген уақыт мерзіміне дейін болатын жағдайлардың басым бөлігінде ғана жауап беруге міндетті болса, ал ол кейде тіпті өзіне берілген уақытты өткізіп алатындай түрде болатын болса, онда мұндай жүйені нақты уақыт кезеңінің жұмсақ шарттары бар жүйелерге жатқызуға болады. *Нақты уақыт кезеңінің жұмсақ шарттары бар* жүйелер мысалы ретінде автомобиль терезелерінің контроллерлері мен стереожүйенің күшейткіштерін қарастыруға болады. Қарапайым адам терезе сүйкеуіштерінің бір-екі миллисекундқа кешігіп қалғандығын бәрібір байқамай қалады, тек тәжірибелі тыңдарман ғана дыбыс ұзақтығының миллисекундтық өзгерісін сезіп қалуы мүмкін. Нақты уақыт кезеңінің қатаң шарттары бар жүйелер мысалы болып, поршень қозғалысын есепке ала отырып нақты берілген уақыт мезетінде бензин шашатын жанармай инжекторы саналады. Егер де мұнда бір миллисекундқа кешігу болатын болса, онда қозғалтқыш қуаты азайып, ол бұзыла бастайды; ақырында қозғалтқыш істен шығып, жол оқиғасына немесе апатқа алып келуі мүмкін.

- *Алдын ала болжау.* Бұл – құрамдас жүйелердегі өзекті түсінік. Әрине, бұл терминнің көптеген интуитивті түсініктері болуы мүмкін, бірақ мұнда – құрамдас жүйелерді программалау контексінде – біз оның тек техникалық мәнін қолданамыз: егер бір әрекет осы компьютерде әрқашанда бір уақытта ғана және де осындай операциялардың бәрі де бір сәтте орындалатын болса, онда операцияға алдын ала *болжау* (predictable) жасалған болып саналады. Мысалы, егер `x` және `y` – бүтінсандық айнымалылар болса, онда `x+y` нұсқауы әрқашанда бір бекітілген уақытта орындалады. Ал егер `xx` және `yy` – басқа екі бүтінсандық айнымалылар болса, онда `xx+yy` нұсқауы да дәл сондай

уақытта орындалады. Көбінесе, машина архитектурасына (мысалы, кэштеу ерекшеліктеріне және конвейерлік өңдеуге сәйкес ауытқулар) байланысты операциялардың орындалу жылдамдықтарына қарай туындайтын аздаған өзгерістерді есепке алмай-ақ, жай ғана берілген уақыттың жоғарғы шегін алу керек. Алдын ала болжауға болмайтын операцияларды (осы сөздің тікелей мағынасында) нақты уақыт кезеңдерінің қатаң шарты бар жүйелерде пайдалануға болмайды және оларды нақты уақыт кезеңінің басқа жүйелерінде де аса жоғарғы сақтықпен пайдалану керек. Болжауға болмайтын операцияның классикалық мысалы болып, тізімнің элементтері саны белгісіз болып және оны жоғарыдан бағалау мүмкін болмаған кездегі тізім бойынша сызықтық іздеу ісі (мысалы, `find()` функциясын орындау) саналады. Мұндай іздеуді, егер біз тізім элементтерінің санын немесе оның ең үлкен мәнін сенімді түрде айта алатын болсақ, нақты уақыт кезеңдерінің қатаң шарты бар жүйелерде қолдануға болады. Басқаша айтқанда, керекті жауап бекітілген уақыт аралығында келетініне кепілдеме беру үшін, біз уақыт қорын толық пайдалануға алып келетін командалардың кез келген тізбегін орындауға кететін уақытты – мүмкін, кодты талдау құралдары арқылы болар – есептеп шығуымыз керек.

- *Параллелизм.* Құрамдас жүйелерге көбінесе сыртқы ортада болып жатқан оқиғалар әсер етеді. Бұл осы программада бірнеше оқиғалардың бір мезетте қатар орындалып жатуының мүмкін екендігін көрсетеді, өйткені олар сол сәтте қатарласа жүріп жатқан нақты ортадағы оқиғаларға сәйкес келеді. Бірнеше әрекеттерді бір сәтте қатарластыра орындайтын программа *параллельдік* (concurrent parallel) программа деп аталады. Өкінішке орай, осындай өте қызықты, қиын әрі маңызды тақырып біздің кітап шеңберінде қарастырылатын мәселелерден мазмұнына байланысты тыс қалып отырғанын айтпасқа болмайды.

25.2.1 Болжау

C++ тілі алдын ала болжау тұрғысынан қарағанда, өте жақсы тіл, бірақ идеалды емес. Іс жүзінде C++ тілінің барлық құралдарын (виртуалдық функцияларды шақыруды қоскқанда), төмендегілерді қоспағанда, алдын ала болжауға болады.

- Бос жады аймағын `new` және `delete` операторлары арқылы бөлу (25.3 бөлімін қ.).
- Аластамалар (19.5 бөлім).
- `dynamic_cast` операторы (А.5.7 бөлім).

Осы құралдарды нақты уақыт кезеңдерінің қатаң шарттары бар қосымшаларда пайдаланбаған дұрыс. `new` және `delete` операторларымен байланысты

проблемалар тиянақты түрде 25.3 бөлімінде сипатталған; олар айқындалған сипатта берілген. Стандартты кітапханадағы `string` класы мен стандартты контейнерлер (`vector`, `map` және т.б.) жанамалы түрде бос жады аймағын пайдаланады, сондықтан оларды да алдын ала болжауға болатынына назар аударыңыз. `dynamic_cast` операторының проблемасы оның параллель түрде жүзеге асырылу қиындықтарына байланысты туындайды, бірақ ол іргелі қиындық емес.

Аластамалармен болатын проблемалар, нақты `throw` бөліміне қарап тұрып, программалаушы соған сәйкес `catch` бөлімін іздеу қанша уақыт алатынын және сондай бөлімнің бар екенін үлкенірек программа фрагментін талдамай тұрып айта алмайтынына келіп тіреледі. Құрамдас жүйелерге арналған программаларда дәл осындай `catch` бөлімі болған болса, жақсы болар еді, өйткені біз программалаушының C++ тіліндегі түзетіп жөндеу құралдарын пайдаланатынына көзіміз жетпейді.

Негізінде аластамалармен байланысты проблемаларды, нақты `throw` бөлімі үшін қандай `catch` бөлімі шақырылатынын және оған басқару қаншалықты ұзақ уақытта берілетінін анықтайтын механизммен шешуге болады. Дегенмен, қазіргі кезде бұл мәселе әлі зерттеліп жатыр, сондықтан егер сізге алдын ала болжау керек болса, онда сіз қайтарылатын кодтарға және де басқа ескірген әрі шаршататын, бірақ болжауға болатын әдістерге негізделген қателерді өңдеуге тиістісіз.

25.2.2 Қағидалар

Құрамдас жүйелер үшін программалар құру кезінде жоғары жұмыс өнімділігі мен кодтың сенімділігін арттыру мақсатында программалаушы тек қана төменгі деңгейдегі тіл құралдарын қолданатын болады деген қауіп бар. Бұл стратегия шағын код фрагменттерін жазу кезінде өзін-өзі ақтайды. Бірақ ол мұндай жүйе жасауға керекті код дұрыстығын тексеруді қиындатып, уақыт пен ақша шығынын арттырып, барлық жобаны аяқ алып жүре алмас лайсаңға да айналдырып жіберуі мүмкін.

Әр кездегі сияқты, біздің мақсат – есебімізге байланысты қойылған шектеулерді есепке ала отырып, барынша жоғары деңгейде жұмыс істеу. Өзіңізді мақталып жүрген төменгі деңгейдегі ассемблерлік кодқа дейін түсіріп жібермеңіз! Әрқашанда программада өз идеяларыңызды мүмкіндігінше тікелей түрде (берілген шектеулерді сақтай отырып) өрнектеуге тырысыңыз. Айқын әрі түсінікті түрде, сүйемелдеуге де жеңіл код жазуға ұмтылыңыз. Сізді мәжбүрлемесе, оңтайланған тиімді код жазуға талпынбаңыз.

Құрамдас жүйелер үшін тиімділіктің (уақыт немесе жады көлемі бойынша) үлкен мәні бар, бірақ әрбір шағын код бөлігін барынша қысқартуға да тырыспаған дұрыс. Оның үстіне, көптеген құрамдас жүйелерден, бірінші кезекте, программаның дұрыс құрылып, жылдам жұмыс істеуі талап етіледі; сіздің программаңыз жылдам жұмыс істеп тұрғанда, компьютерлік жүйе сіздің келесі әрекетіңізді күтіп,

тосып тұрады. Тұрақты түрде барынша тиімді етіп бірнеше жол код жазуға ұмтылу көп уақыт алады, қате де көбірек кетеді және программаны оңтайландыруды (оптимизациялауды) да қиындатады, өйткені алгоритмдер мен мәліметтер құрылымын түсіну мен толықтыру да қиындай түседі. Мысалы, төменгі деңгейдегі оңтайландыру кезінде көбінесе компьютер жадын пайдалануды оңтайландыру мүмкін болмайды, өйткені бірсыпыра жерлерде программа бөліктерінің аздаған қосымша айырмашылықтарына байланысты басқа бөліктері пайдалана алмайтын бірдей кодтар тізбегі пайда болады.

Өзінің өте тиімді программалар жазуымен аты шыққан Джон Бентли (John Bentley), оңтайландыру тәсілдерінің екі заңын тұжырымдады:

- Бірінші заңы: "Мұндайды жасама!"
- Екінші заңы (тек сарапшылар үшін): "Мұндайды әзірге жасама!"

Программаны оңтайландыруға кіріспес бұрын сіз жүйенің қалай жұмыс істейтінін біліп алуыңыз керек. Тек сонда ғана сіз оңтайландыру ісі дұрыс әрі сенімді жұмыс істейтіндігіне (немесе кейін солай болатынына) сенімді бола аласыз. Алгоритмдер мен мәліметтер құрылымына көбірек көңіл бөліңіз. Жүйенің алғашқы нұсқасы іске қосылысымен оның көрсеткіштерін мұқият өлшеп шығыңыз да, ойдағыдай етіп баптаңыз. Қуанышқа орай, күтпеген жерден жақсылықтар да болып жатады: дұрыс код кейде әжептеуір жылдам жұмыс істейді және компьютер жадының көп көлемін де алмайды. Дегенмен, бұлай бола береді деп ойлай бермеңіз; бәрін де өлшеп отырыңыз. Келеңсіз тосын жағдайлар да жиі кездесіп жатады.

25.2.3 Жүйенің ақаудан кейінгі жұмыс қабілетін сақтау

Сізге кенеттен ақау шығып, тұрып қалмайтындай жүйе жасау керек болсын делік. "Тұрып қалмайтын" деген сөзге біз "ай бойы адам араласуынсыз жұмыс істеу" деген ұғым береміз. Біз қандай ақаулықтарды болдырмауымыз керек? Әрине, біз кенеттен күн сөніп қалады немесе жүйені піл басып кетеді деп ойламаймыз. Бірақ біз жалпы ненің қандай күйде болатынын алдын ала дөп басып айта алмаймыз. Дегенмен, нақты бір шынайы жүйе үшін жүзеге асуы мүмкін болатын қателер жайлы болжамдар жасайтындай түрде болуымыз керек. Жиі болып жататын жағдайларға мысалдар келтірейік.

- Ақау шығуы немесе энергия көзінің өшіп қалуы;
- Ажыратып қосқыштың дірілдеуі;
- Жүйеге ауыр зат түсіп, процессордың істен шығып қалуы;
- Жүйелік блоктың жоғарыдан құлап кетуі (соққыдан диск істен шығып қалуы ықтимал);

- Жады ұяшықтарында жазылған мәліметтің ойламағандай түрде өзгеруін туғызатын радиоактивті сәулелендірудің орын алуы.

Бәрінен де қиыны жоғалып кетіп жататын қателерді табу. *Жоғалып кететін қате* (transient error) деп біз кейде орын алып, бірақ программа орындалған сайын қайталанбайтын оқиғаны айтамыз. Мысалы, процессор температура 54 °C шамасынан асқан кезде ғана дұрыс істемеуі мүмкін. Мұндай оқиға мүмкін емес сияқты болып көрінеді, бірақ бірде жүйені зауыт цехының еденіне қойып ұмытып кеткен кезде, оның болғаны бар, бірақ ондай жағдай зертханада ешқашанда қайталанған емес.

Зертхана қабырғасында болмайтын қателерді анықтау өте қиын болады. Мысалы, реактивті қозғалтқыштар зертханасының инженерлері Марста жұмыс істеп жүрген (оған сигнал 20 минуттан соң барады) құрылғыдағы қозғалтқыштың программалық және аппараттық жабдықтамасы жұмысының қатесін анықтап, оны талдап жөндеу үшін қандай күш жұмсалатындығын көзіңізге елестете де алмайсыз.

Қателерге тұрақты жүйелерді жасау мен оларды жүзеге асыруда пәндік аймақты жетік білу ісі, яғни жүйенің өзін, оның айналасын және қолданылу ерекшеліктерін жақсы игеру өте маңызды рөл атқарады. Мұнда біз тек жалпыға ортақ мәселелерді ғана аздап қарастырып шығамыз. Бұлардың әрқайсысының мыңдаған ғылыми мақалалардың және ондаған жылдар бойы жүргізілген зерттеулердің жемісі екенін айта кетейік.

- *Ресурстардың азаюын болдырмау.* Азаюды болдырмаңыз. Өз программаңыздың қандай ресурстарды пайдаланатындығын анық білуге тырысыңыз да, соларды үнемдеуге (идеалда) ұмтылыңыз. Кез келген ресурстың азаюы ең соңында сіздің жүйеңіздің немесе ішкі жүйеңіздің тоқтап қалуына, яғни ақаулығына келіп тіреледі. Ең маңызды ресурстарға уақыт пен компьютер жады жатады. Көбінесе, программа басқа да ресурстарды, мысалы, бұғаттауларды, байланыс арналарын және файлдарды пайдаланады.
- *Қос-қостан жасау.* Егер жүйенің жұмыс жасауы үшін белгілі бір құрылғының дұрыс істеуі (мысалы, компьютер, шығару құрылғысы, дөңгелек) өте маңызды болып табылатын болса, онда жобалаушылар алдында іргелі таңдау – осы маңызды ресурсты екеу етіп жасау мәселесі туындайды. Біз керекті аппараттық жабдықтама тұрып қалып, ақау шығып жататындығымен келісуіміз керек немесе тағы бір қосымша (резервтегі) құрылғы қосып, оны осы программалық жабдықтаманың басқаруына беруіміз керек. Мысалы, кемелердің дизельдік қозғалтқыштарындағы жанармай инжекторлары контроллерінің керекті кезде іске қосылатын, желі арқылы байланысқан параллель (резервтелген) үш компьютері болады. Резервте тұрған қосымша құрылғылар негізгі түпнұсқалық құрылғымен бірдей болуы шарт емес (мысалы, ғарыш зондының қуатты негізгі антеннасы және одан әлсіз қосымша антеннасы бар) екенін атап өтейік.

Қосымша құрылғыларды жүйенің күнделікті жұмысында оның жұмыс өнімділігін арттыру үшін де пайдалануға болатынын айта кетейік.

- *Өзін-өзі тексеру.* Программаның (немесе аппараттық жабдықтаманың) дұрыс жұмыс істемей тұрғанын білу керек. Осы тұрғыдан алғанда, аппараттық жабдықтамалардың өздерін өздері бақылайтын (мысалы, есте сақтау құрылғылары), кішігірім қателерді жөндеп түзететін және үлкен қателіктер жайлы мәлімет беретін компоненттері өте пайдалы болып табылады. Программалық жабдықтама мәліметтер құрылымының біртұтастығы мен инварианттарды (9.4.3 бөлімді қ.) тексере алады және ішкі "санитарлық бақылау" (тексеру операторлары) жүргізе алады. Өкінішке орай, өзін-өзі тексеру ісі өздігінен онша сенімді түрде жұмыс істей алмайды, сондықтан қате туралы берілген мәліметтің өзі қате шығарып жүрмеуін қадағалау қажет. Қателерді тексеру құралдарын толық тексеріп шығудың өзі – өте күрделі жұмыс.

- *Дұрыс істемейтін программадан жылдам шығу жолы.* Жүйені модульдерден құрастыру керек. Қателерді өңдеу негізіне модульдік қағида жатуы тиіс: әрбір модульдің өз жеке есебі болуы керек. Егер модуль өз тапсырмасын орындай алмайтын болса, ол туралы басқа модульге хабар беріле алады. Модуль ішіндегі қателерді өңдеу қарапайым түрде болуы тиіс (бұл оның дұрыс және тиімді жұмыс істеу ықтималдығын арттырады), ал күрделі қателерді өңдеумен басқа бір модуль айналысуы тиіс. Сенімділігі жоғары жүйелер модульдерден және көптеген деңгейлерден тұрады. Әрбір деңгейде пайда болған күрделі қателер жайлы мәліметтер келесі деңгейге беріледі де, ақырында олар адамға да жетуі мүмкін. Күрделі қате жайлы мәлімет алған модуль (оны басқа ешқандай да модуль түзете алмаса) соған сәйкес әрекеттер атқарып, қате шыққан модульді қайта жүктеуі мүмкін немесе одан күрделілігі шағындау (бірақ сенімдірек) резервтегі басқа модульді іске қоса алады. Нақты бір жүйедегі модульді ерекшелеп бөліп алу ісі – жобалау кезінде істелетін жұмыс, бірақ негізінде модуль болып класс, кітапхана, программа немесе компьютердегі барлық программалар есептеле береді.

- *Ішкі жүйелер мониторингі,* бұлар туындаған проблемалар жайлы жүйелердің өздері хабарлай алмайтын жағдайларда орындалады. Көп деңгейлі жүйелерде төменірек деңгейде орналасқан жүйелерді жоғарғы деңгейдегі жүйелер қадағалап отырады. Тоқтап қалмауы (ақау болмауы) тиіс көптеген жүйелердің (мысалы, кеме қозғалтқыштары немесе ғарыш станцияларының контроллерлері) үш-үштен резервтегі ішкі қосымша көшірмелері болады. Мұндай үш есе үлкейту тек екі резервтік көшірмелердің бар екенін ғана білдіріп қоймайды, олардағы қай жүйенің істен шыққаны жайлы шешім "біреуге қарсы екеудің" дауыс беруі арқылы шешіліп отырады. Үш есе үлкейтіп көп деңгейлі жүйелерді ұйымдастыру істері олар тым күрделі (мысалы, жүйенің немесе ішкі жүйенің ең жоғарғы

деңгейі ешқашан тоқтап қалмауы тиіс болғанда) болып кеткен кездерде өте пайдалы болып саналады.

Біз жүйені өзіміз қалай қаласақ, солай етіп жобалай аламыз және оны қалай істей алсақ, сондай етіп жүзеге асыра аламыз, бірақ ол бәрі бір дұрыс жұмыс жасамайтындай қалыпта қалып қоюы мүмкін. Қолданушыларға беру алдында оны жүйелі түрде және мұқият тесттен өткізіп алу қажет (бұл туралы толығырақ 26-тарауда айтылады).

25.3 Компьютер жадын басқару

Компьютерің ең негізгі екі ресурсы болып уақыт (нұсқауларды орындауға кететін) пен жады (мәліметтер мен кодты сақтайтын) саналады. C++ тілінде мәліметтерді сақтау үшін компьютер жадын бөлудің үш тәсілі бар (17.4 және A.4.2 бөлімдерін қ.).

- *Статикалық жады.* Байланыс редакторы бөледі және ол программа орындалып біткенше пайдаланылады.
- *Стектік (автоматты) жады.* Функцияны шақырған кезде бөлінеді де, функциядан қайтып, басқару программаға берілгенде ол босатылады.
- *Динамикалық жады (үйінді).* `new` операторымен бөлініп беріледі де, кейіннен қайта пайдалану үшін `delete` операторы арқылы босатылады.

Осылардың әрқайсысын құрамдас жүйелерді программалау тұрғысынан қарастырып шығайық. Анықтап айтсақ, болжам жасау (25.2.1 бөлімін қ.) маңызды рөл атқаратын есептер тұрғысынан, мысалы, нақты уақыт кезеңінің қатаң шарттары бар жүйелерді және қауіпсіздікті қамтамасыз етуге ерекше талап қоятын жүйелерді программалаудағы компьютер жадын басқару мәселелерін оқып үйренеміз.

Статикалық жады құрамдас жүйелерді программалауда ерекше проблемалар туғызбайды: мұндағы компьютер жады программа іске қосылғанша және жүйе қанат жайғанша толығымен бөлініп беріледі.

Стектік жады проблемалар туғызуы ықтимал, өйткені оның көлемі жеткіліксіз болып қалуы мүмкін, бірақ бұл мәселеден құтылу қиын емес. Жүйені жасаушылар программаның орындалу барысында стек өзінің мүмкін болатын шегінен ешқашанда асып кетпеуін қадағалап отыру керек. Көбінесе бұл функцияларды қабаттастыра шақыру саны шектеулі болуы тиіс екендігін көрсетеді; басқаша айтқанда, біздегі функциялардың бірін-бірі шақыратын тізбегінің (мысалы, f_1 шақырады f_2 шақырады ... f_n шақырады) ешқашанда өте ұзын болып кетпейтінін көрсететін мүмкіндік болуы керек. Кейбір жүйелерде бұл әрекет рекурсивтік шақыруларға тыйым салуға алып келеді. Ал кейбір жүйелерде мұндай тыйым салуларды қолдану бірсыпыра рекурсивтік функцияларға қатысты дұрыс шешім

болып табылады, бірақ оларды әмбебап тәсіл деп санауға болмайды. Мысалы, мен `factorial(10)` нұсқауын орындау `factorial` функциясын оннан артық шақырмайтынын білемін. Бірақ құрамдас жүйені жасайтын программалаушы күмәнді немесе кездейсоқтық жағдай болып қалмасын деп көбінесе `factorial` функциясының (15.5 бөлімін қ.) итеративті нұсқасын таңдап алып та жатады.

Компьютер жадын динамикалық түрде бөлуге әдетте тыйым салынады немесе оған қатаң шектеу қойылады; басқаша айтқанда, `new` операторына тыйым салынады немесе оны пайдалануға программаны іске қосу периоды санымен шектеу қойылады, ал `delete is` операторына тыйым салынады. Осындай шектеулердің негізгі себептерін көрсетейік.

- *Болжау.* Мәліметтерді бос жады аймағында орналастыруға алдын ала болжам жасауға болмайды; басқаша айтқанда, бұл операцияның тұрақты уақыт аралығында орындалатынына кепілдік берілмейді. Көбінесе, бұл былай болмайды: new операторының жасалған көптеген нұсқаларында жаңа объектіні орналастыруға қажетті уақыт бірсыпыра объектілерді орналастырып болып, өшіргеннен кейін өте тез өсіп кетіп те жатады.

- *Фрагменттерге бөліну*. Компьютердің бос жады аймағы фрагменттерге бөлініп кетуі мүмкін; басқаша айтқанда, объектілерді орналастырып, кейіннен оларды өшірген соң қалған жады көлемі қолданылмайтын жады аймағы болып табылатын көптеген "тесіктер" санынан құралуы мүмкін; олар пайдасыз болып табылады, өйткені әрбір "тесік" программада қолданылатын бір объект толық сыятындай емес, өте шағын көлемді алып тұрады. Сонымен, пайдаланылатын бос жады көлемі бастапқы жады көлемі мен объектілер орналасқан көлемнің айырмасынан әлдеқайда аз болуы мүмкін.

Келесі бөлімде біз мұндай келеңсіз жағдай қалай қалыптасатынын көрсетеміз. Осыдан барып, біз нақты уақыт кезеңдерінің қатаң шарттары бар жүйелерде немесе қауіпсіздікті қамтамасыз етуге ерекше талап қоятын жүйелерде `new` және `delete` операторларын пайдаланатын программалау тәсілдерін қолданбауға тырысуымыз керек болады. Келесі бөлімде біз стектер мен пулдарды пайдалана отырып, бос жады аймағына байланысты проблемалардан қалай құтылуға болатынын көрсетеміз.

25.3.1 Бос жады аймағымен болатын проблемалар

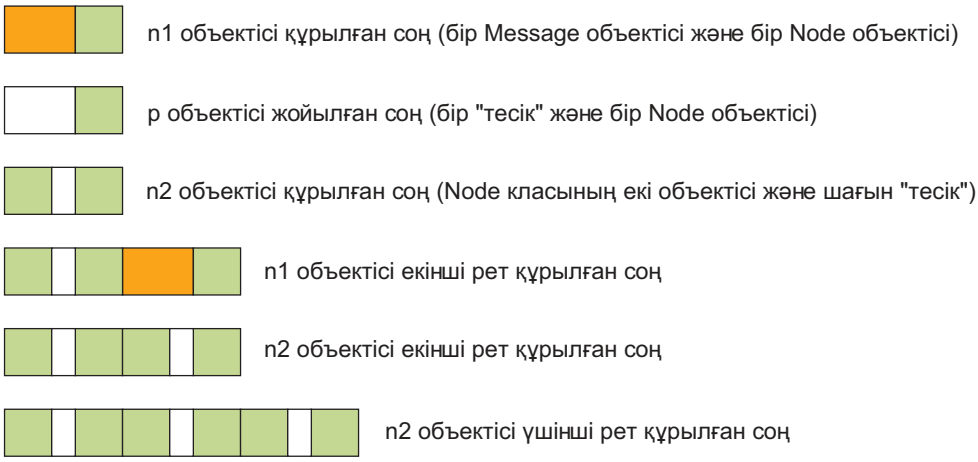
`new` операторымен байланысты проблема қайдан шығады? Негізінде бұл проблема `new` және `delete` операторларын қатар қолданудан туындайды. Келесі объектілерді орналастыру мен өшіру әрекеттері тізбектерінің нәтижесін қарастырайық.

```
Message* get_input(Device&);
// бос жады аймағында Message класы объектісін құрамыз

while(/* . . . */) {
        Message* p = get_input(dev);
        // . . .
        Node* n1 = new Node(arg1,arg2);
        // . . .
        delete p;
        Node* n2 = new Node (arg3,arg4);
        // . . .
}
```

Осы циклді орындаған сайын біз **Node** класының екі объектісін жасаймыз, оларды жасау барысында **Message** класының объектісі жасалады және өшіріледі. Мұндай код фрагменті басқа бір құрылғыдан келіп түсетін мәліметтерді енгізу үшін қолданылатын мәліметтер құрылымдарында жиі пайдаланылады. Осы кодқа қарап тұрып, циклді орындаған сайын біз жады аймағының **2*sizeof (Node)** байтын жұмсайтынымызды (плюс бос жады аймағын пайдалану) айтып тұжырым жасай аламыз. Өкінішке орай, біз пайдаланатын жады аймағы жоғарыдағы **2*sizeof(Node)** байттармен шектелетініне ешқандай кепілдік беруге болмайды. Негізінде мұның осылай болатынына сену де қиын.

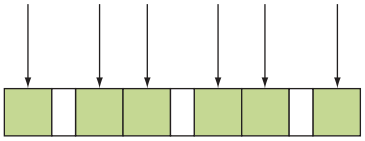
Компьютер жадын басқарудың қарапайым (және осындай болуы да ықтимал) механизмін көзге елестетейік. Және де **Message** класының объектісі **Node** класының объектісіне қарағанда, аздап көлемдірек делік. Мұндай жағдайды келесідей түрде көрсетуге болады: сары түспен **Message** класының объектісі алып тұрған жады аймағын белгілейік те, ал жасыл түспен – **Node** класының объектісі алып тұрған аймақты және ақ түспен – "тесіктерді" (яғни пайдаланылмайтын жады аймағы) белгілейік.



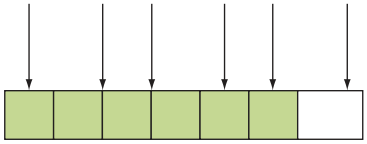
Сонымен, циклді орындаған сайын, біз пайдаланылмайтын жады аймағын ("тесікті") бос қалдырып отырамыз. Мұндай жады көлемі бірнеше байт қана болуы мүмкін, бірақ егер біз оны пайдалана алмасақ, онда ол жады көлемінің азаюына парапар болады, ал тіпті аз ғана азаюдың өзі жалғаса берсе, ерте ме кеш пе әйтеуір бір күндері ұзақ уақыт істейтін жүйелерді де істен шығарады. Бос жады аймағын осындай бір де бір объектіні орналастыруға болмайтын көптеген "тесіктерге" *бөлу жадының фрагменттерге бөлінуі* (memory fragmentation) деп аталады. Ақырында, бос жады аймағын басқару механизмі программа пайдаланатын объектілерді орналастыруға болатын барлық "тесіктерді" қамтиды да, ешбір объект сыймайтын, сондықтан пайдаланылмайтын өте кішкене бір "тесікті" ғана бос қалдырады. Бұл ұзақ жұмыс істейтін, **new** және **delete** операторларын кеңінен қолданатын барлық программаға ортақ күрделі проблема; жады аймағының осылай фрагменттерге бөлінуі жиі кездесіп жатады. Ол new операторын орындауға керекті уақытты тым үлкейтіп жібереді, өйткені ол объектілерді орналастыруға сәйкес келетін орынды іздеп табуды жүзеге асыруы тиіс. Әрине, құрамдас жүйелер үшін мұндай тәртіп орнатуға болмайды. Бұл әрекет асығыс құрылған құрамдас емес жүйелердің өзінде де күрделі проблемалар туғызуы мүмкін.

Неге бұл проблеманы тіл де, жүйе де шеше алмайды? Ал компьютер жадында ешқандай да "тесік" жасамайтындай программа жазуға болмас па екен? Алдымен жады аймағындағы қажетсіз кішкентай "тесіктер" проблемасының сұранып тұрған ең тікелей шешімін қарастырайық: **Node** класы объектілерінің барлығын да бос жады аймағы объектілерді түгел орналастыра алатындай түрде біртұтас үздіксіз көлемді қамтитындай етіп жылжытуға тырысайық .

Өкінішке орай, жүйе мұны істей алмайды. Оның себебі, C++ тіліндегі код компьютер жадында орналасқан объектілерге ғана тікелей сілтеме жасайды. Мысалы, **n1** және **n2** нұсқауыштары жады ұяшықтарының нақты адрестерін есте сақтайды. Егер біз олар сілтеме жасап тұрған объектілерді жылжытсақ, онда мұнан кейін адрестер дұрыс болмай қалады. Біз құрылған объектілерге арналған нұсқауыштарды (бір жерде) сақтадық делік. Сонда біз өз мәліметтер құрылымымыздың сәйкес бөлігін келесідей түрде бейнелей аламыз.



Енді біз жады аймағын тығыздап, пайдаланылмайтын жады көлемі үздіксіз фрагмент болатындай етіп объектілерді жылжытамыз.



Өкінішке орай, объектілерді жылжытып, бірақ оларға сілтеме жасап тұрған нұсқауыштарды жаңаламай, біз шатаса бастадық. Неге біз объектілерді жылжытқан соң, нұсқауыштарды өзгертпедік? Біз мұндай программаны мәліметтер құрылымының нақтылықтарын (бүге-шігесін) біліп отырып қана жаза алар едік. Негізінде жүйе (яғни, C++ тілін динамикалық сүйемелдеу жүйесі) нұсқауыштардың қайта сақталатынын білмейді; басқаша айтқанда, бізде объект бар болғанмен, "Дәл қазір осы объектіге қандай нұсқауыштар сілтеме жасап тұр?" – деген сұраққа жауап бере алмаймыз. Бірақ тіпті, бұл проблеманы жеңіл шешуге болғанмен, мұндай тәсіл (*қоқысты жинауды тығыздап жинау* деген атпен белгілі (compacting garbage collection)) әрқашанда өзін ақтай бермейді. Мысалы, ол жақсы жұмыс істеуі үшін, әдетте, жүйеге нұсқауыштарды қадағалап, объектілерді жылжытуға керекті жады көлемінен екі есе артық бос жады көлемі талап етіледі. Осындай бос жады көлемі құрамдас жүйеде болмай қалуы мүмкін. Оның үстіне, қоқысты тығыздап жинаудың тиімді механизмінен де алдын ала болжам жасауды талап ету қиын.

Әрине, біздегі мәліметтер құрылымдары үшін "Нұсқауыштар қайда орналасқан?" деген сұраққа жауап беріп, оларды тығыздап қысуға да болады, бірақ жалпы блок басында жады аймағын фрагменттерге бөлуден сақтау қиын емес. Қарастырылатын мысалда біз **Node** класының екі объектісін де **Message** класының объектілеріне дейін орналастыруымызға болар еді.

```
while( . . . ) {
     Node* n1 = new Node;
     Node* n2 = new Node;
     Message* p = get_input(dev);
     // ... ақпаратты түйіндерде сақтаймыз ...
     delete p;
     // ...
}
```

Бірақ жалпы жағдайда фрагменттерге бөлуді болдырмау үшін кодтарды қайта құру ісі қарапайым есеп емес. Оны сенімді түрде шешу қиын жұмыс. Көбінесе бұл жақсы программалар жазудың басқа бір ережелеріне қарсы қайшылықтарға әкеліп жатады. Осының салдарынан біз бос жады аймағын пайдалануды тек блок басында ғана фрагменттерге бөлуді болдырмайтын тәсілдермен шектеуді ұсынамыз. Көбінесе проблеманы шешуден гөрі оны болдырмау жеңіл болып табылады.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Компьютер жадындағы "тесіктердің" қалай пайда болатынын көру үшін немесе олар жалпы пайда бола ма, жоқ па соны анықтау үшін, жоғарыда келтірілген программаны орындаңыз да, жасалған объектілердің адрестері мен көлемдерін баспаға шығарыңыз Егер уақытыңыз болса, фрагменттерге бөлінудің қалай жүзеге асатынын жақсылап түсіну үшін, жоғарыдағы суреттердегі сияқты жады сұлбасын бейнелеп салып шығыңыз.

25.3.2 Әмбебап бос жады аймағының баламасы

Сонымен, біз жады аймағының фрагменттерге бөлінуін болдырмауға тырысуымыз керек. Ол үшін не істеу керек? Біріншіден, **new** операторы өзінен өзі фрагменттерге бөлуді жүзеге асырмайды; "тесіктер" пайда болуы үшін **delete** операторы керек. Сол себепті, алдымен **delete** операторына тыйым салу қажет. Сонда компьютер жадында орналасқан объект сақталып қала береді.

Егер **delete** операторына тыйым салынса, онда **new** операторын болжауға болады; басқаша айтқанда, барлық **new** операторлары бірдей уақытта орындала ма екен? Иә, бұл ереже тілдің барлық нұсқаларында орындалады, бірақ оған стандартпен кепілдік берілмейді. Әдетте, компьютер іске қосылған соң немесе ол қайта жүктелгеннен кейін құрамдас жүйені дайындық қалпына келтіретін бірсыпыра жүктеу командалары орындалады. Жүктеу орындалып жатқан кезде біз жады аймағын өз қалауымызша, тіпті оны пайдаланғанша да, бөлуімізге болады. Сонымен, біз **new** операторын жүктеу кезеңінде орындай аламыз. Оған балама (немесе толықтыру) ретінде ауқымды, яғни глобалды жады аймағын (статикалық жады) кейіннен қолдану үшін резервке қоя аламыз. Программалар құрылымының ерекшеліктеріне байланысты ауқымды мәліметтерді қолданбаған дұрыс, бірақ кейде жады көлемін алдын ала бөлу үшін осы механизмді пайдаланған артық болмайды. Бұл механизмнің айқындалған жұмыс істеу ережелері осы жүйенің программалау стандарттары арқылы орнатылады (25.6 бөлімді қ.).

Болжам жасай отырып жады бөлу ісінде өте пайдалы болып саналатын екі мәліметтер құрылымы бар.

- Стектер. *Стек (stack)* – бұл кез келген мәліметтер санын (ең жоғарғы көлемнен аспайтын) орналастыруға болатын мәліметтер құрылымы, мұнда ең соңынан енгізілген мәліметті ғана өшіруге болады; яғни стек өз көлемін тек төбесі арқылы ғана үлкейте немесе кішірейте алады. Ол жады аймағын фрагменттерге бөлмейді, өйткені оның екі ұяшығының арасында "тесік" болуы мүмкін емес.
- Пулдар. *Пул (pool)* – бұл мөлшерлері бірдей объектілер жиынтығы (коллекциясы). Біз объектілерді пулға енгізіп, орналастыра аламыз және одан өшіруге де болады, бірақ оған оның ішкі көлеміне сыятын объектілер санын ғана енгізе аламыз. Жады аймағының фрагменттерге бөлінуі мұнда болмайды, өйткені объектілердің мөлшерлері бірдей болып келеді.

Стектер мен пулдарға объектілер енгізу (орналастыру) және өшіру операцияларын алдын ала болжауға болады және олар жылдам орындалады.

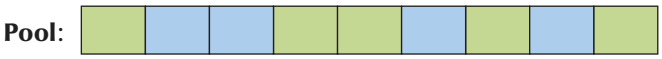
Сонымен, нақты уақыт кезеңінің қатаң шарттары бар жүйелер мен қауіпсіздікке ерекше талаптар қоятын жүйелерде, қажет болып жатса, стектер мен пулдарды пайдалануға болады. Оның үстіне, тәуелсіз тұлғалар немесе мекемелер (олардың спецификациялары сіздің талаптарыңызды қанағаттандыратындай болып жатса) жасап шығарған, іске асырған және тесттен өткізген стектер мен пулдарды пайдаланған дұрыс болып табылады.

C++ тілінің стандартты контейнерлерін (**vector**, **map** және т.б.) және **string** стандартты класын құрамдас жүйелерде тікелей пайдалануға болмайды, өйткені олар жанамалы түрде болса да, **new** операторын қолданатынына назар аударыңыз. Алдын ала болжам жасауды қамтамасыз ету үшін, стандартты контейнерлерге ұқсас контейнерлер жасап алуыңызға (сатып алуыңызға немесе уақытша ала тұруыңызға) болады, бірақ сіздегі C++ тілінің қалыпты контейнерлері құрамдас жүйелерде пайдалануға арналмаған.

Әдетте, құрамдас жүйелер сенімділікке өте үлкен талаптар қояды, сондықтан шешім қабылдау кезінде сіз бір деңгейге төмен түсіп, төменгі деңгей құралдарын қолданып жүрмеңіз, яғни сіз біздің программалау стилімізді ұстаған соң, одан ешқашанда ауытқымауыңыз керек, осыны атап өткен артық болмайды. Нұсқауыштармен, тікелей түрлендіру әрекеттерімен және де осыған ұқсас басқа да амалдармен толтырылған программаның дұрыс жұмыс істеуі өте сирек болатын құбылыс екенін естен шығармаңыз.

25.3.3 Пул мысалы

Пул – бұл ішінен берілген типтегі объектілерді шығарып алуға және оларды өшіруге де болатын мәліметтер құрылымы. Пулды құру кезінде оның ішінде сақтауға болатын объектілердің максималды саны тағайындалады. Бұрыннан орналастырылған объектіні көрсету үшін жасыл түсті және объектіні орналастыруға болатын орынды көрсету үшін көк түсті пайдаланып, біз пулды келесідей түрде бейнелей аламыз.



Pool класын былай анықтауға болады:

```
template<class T, int N>class Pool {
// T типіндегі  N объектіден тұратын пул
public:
      Pool();
      // T типіндегі  N объектіден тұратын пул құрамыз
      T* get();
      // пулдан T типіндегі объектінін аламыз;
      // егер бос объектілер болмаса, 0 қайтарамыз
      void free(T*);
      // get() функциясы арқылы пулдан алынған
      // T типіндегі объектіні қайтарамыз
      int available() const;
      // T типіндегі бос объектілер саны
```

```
    private:
        // T[N] үшін орын және пулдан қандай объектілер алынғанын,
        // қандайлары алынбағандығын
        // (мысалы, бос объектілер тізімі)
        // анықтауға мүмкіндік беретін мәліметтер
    };
```

Pool класының әрбір объектісі элементтерінің типімен және объектілерінің ең үлкен (максимал) санымен сипатталады. Оны шамамен мынадай:

```
Pool<Small_buffer,10> sb_pool;
Pool<Status_indicator,200> indicator_pool;

Small_buffer* p = sb_pool.get();
// . . .
sb_pool.free(p);
```

түрде пайдалануға болады.

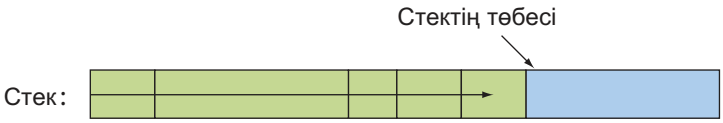
Пул ешқашанда бітпейді, – деп кепілдік беру программалаушының міндеті. "Кепілдік беру" деген сөздің дәл мағынасы қосымшаға байланысты болады. Кейбір жүйелерде программалаушы арнайы код жазуы керек, мысалы, пулда объектілер болмай қалғанда, ешқашанда шақырылмайтын **get()** функциясын жазуға болады. Басқа жүйелерде программалаушы **get()** функциясы жұмысының нәтижесін тексеріп, егер ол нөлге тең болса, кейбір түзетулер жасауына болады. Екінші тәсілдің қалыпты мысалы болып телефондық жүйе есептеледі, ол бір мезетте түскен қоңыраулардың 100 мыңнан астамын қатар өңдей алады. Әрбір қоңырау үшін белгілі бір ресурс, мысалы, нөмір теретін буфер, бөлініп беріледі. Егер жүйе нөмір тергіштердің барлық санын пайдаланып қойған болса (мысалы, **dial_buffer_pool.get()** функциясы **0** қайтарады), онда ол жаңадан нөмір теріп қосуға (және жады аймағын босату үшін бірнеше қосылған байланыстарды ажыратуы да мүмкін) тыйым салатын болады. Мұндай жағдайда кезекті абонент байланысқа шығуды кейінірек жүзеге асыратын болады.

Әрине, бұл **Pool** шаблондық класы пулдар жайлы жалпы идеяның тек бір нұсқасы ғана болып табылады. Мысалы, егер жады аймағын пайдалануға қойылатын шек мұндай қатаң болмаса, элементтерінің сандары конструктормен анықталатын пулдарды анықтай аламыз, тіпті бастапқы көрсетілгеннен көбірек объектілер қажет болып жатса, элементтерінің саны кейіннен өзгертілетін пулдарды да анықтауға болады.

25.3.4 Стек мысалы

Стек – бұл ішінен жады көлемінің порцияларын шығарып алуға және ең соңғы болып орналасқан порцияны босатып алуға болатын мәліметтер құрылымы.

Объектіні орналастыру үшін жасыл түсті және объектіні орналастыруға болатын орынды белгілеу үшін көк түсті пайдалана отырып, біз пулды келесідей түрде бейнелеп көрсете аламыз:



Суретте көрсетілгендей бұл стек оңға қарай "өседі". Объектілер стегін пул сияқты төмендегідей түрде:

```
template<class T, int N> class Stack {
// T типіндегі объектілер стегі
        // . . .
};
```

анықтауға болады.

Бірақ жүйелердің басым бөлігінде әртүрлі мөлшердегі, яғни көлемдегі объектілер үшін жады бөлу керек болады. Стекте мұны істеуге болады, ал пулда – болмайды, сондықтан біз келесі мысалда бір стекті анықтауды көрсетеміз, оның ішінен әртүрлі көлемдегі объектілерге арналған "шикі" жады аймағын алуға болады.

```
template<int N>class Stack {    // N байттан тұратын стек
public:
    Stack();                    // N байттан тұратын стек құрады
    void* get(int n);           // стектен n байт бөледі;
                                // бос жады болмаса 0 қайтарады
    void free();                // get() функциясы қайтарған
                                // ең соңғы мәнді қайтарады
    intavailable()const;//қолжеткізугеболатынбайттарсаны

private:
    // char[N] үшін жады аймағы және стектен
    // қандай объектілер алынғанын,
    // қандайлары алынбағандығын (мысалы, төбеге нұсқауыш)
    // анықтауға мүмкіндік беретін мәліметтер
};
```

`get()` функциясы қажет етілетін байттар санына сілтеме жасайтын `void*` нұсқауышын қайтаратын болғандықтан, біз осы жады аймағын біздің объектілерге керекті типке айналдыруымыз керек. Бұл стекті, мысалы, мынадай түрде:

```
Stack<50*1024> my_free_store;
// 50K жады көлемі стек ретінде қолданылады

void* pv1 = my_free_store.get(1024);
int* buffer = static_cast<int*>(pv1);

void* pv2 = my_free_store.get(sizeof(Connection));
Connection* pconn =
          new(pv2)Connection(incoming,outgoing,buffer);
```

пайдалануға болады.

`static_cast` операторын пайдалану 17.8 бөлімде сипатталған. `new(pv2)` конструкциясы орналастыру синтаксисі деп аталады. Ол келесіні білдіреді: "`pv2` нұсқауышы сілтеме жасап тұрған жады ұяшығында объекті құруы керек". Өздігінен бұл конструкция жады аймағына ешнәрсе орналастырмайды. `Connection` класында аргументтер тізімі (`incoming`,`outgoing`,`buffer`) көрсетілген конструктор бар деп есептеледі. Егер бұл шарт орындалмаса, онда программа компиляциядан өткізілмейді.

Әрине, біздің `Stack` шаблондық класы стектер туралы жалпы идеяның бір нұсқасы ғана болып табылады. Мысалы, егер жады аймағын пайдалануға қойылатын шектеулер онша қатаң болмаса, онда біз қолжетімді байттар санын конструктор арқылы беруге болатын стекті анықтауымызға болады.

25.4 Адрестер, нұсқауыштар және жиымдар

Болжам жасау құрамдас жүйелердің кейбірінен талап етіледі, ал сенімділік барлық жүйелерде де болуы тиіс. Осы талап бізді қателерге төтеп бере алмайтын (құрамдас жүйелерді программалау мәтіндеріне тәуелді) кейбір тілдік конструкциялар мен программалау тәсілдерінен бас тартуға мәжбүрлейді. C++ тіліндегі проблемалардың негізгі бөлігі нұсқауыштарды дұрыс пайдаланбаудан туындайды.

Екі проблеманы атап өтейік:

- Тікелей (тексерілмейтін және қауіпті) түрлендірулер;
- Жиым элементтеріне нұсқауыштар жасау.

Бірінші проблеманы типтерді тікелей түрлендіруді (келтіруді) қолдануды қатаң түрде шектеу арқылы шешуге болады. Нұсқауыштар мен жиымдарға байланысты проблемалардың себептері тереңдеу, олар түсінуді талап етеді де, (қарапайым) кластар арқылы немесе кітапханалық құралдар көмегімен (мысалы, `array` класы; 20.9 бөлімді қ.) шешіледі. Сондықтан бұл бөлімде біз екінші мәселені шешуге тоқталамыз.

25.4.1 Тексерілмейтін түрлендірулер

Физикалық ресурстар (мысалы, сыртқы құрылғылардағы контроллерлер регистрлері) мен солардың төменгі деңгейлі жүйедегі негізгі басқару құрылғыларының нақты адрестері бар. Біз сол адрестерді өз программаларымызда көрсетіп және сол мәліметтерге белгілі бір тип меншіктеуіміз керек. Мысал қарастырайық.

```
Device_driver* p = reinterpret_cast<Device_driver*>(0xffb8);
```

Осы түрлендірулер 17.8 бөлімде де сипатталған. Бұл программалау түрі анықтамалықтарды тұрақты түрде пайдалануды талап етеді. Аппараттық жабдықтама ресурсы – регистр адресі (бүтін сан түрінде өрнектелген, көбінесе он алтылық сан түрінде) мен аппараттық жабдықтаманы басқаратын программалық жабдықтамадағы нұсқауыштар арасында жұқалау сәйкестік бар. Сіз оның дұрыстығын компилятор көмегінсіз қамтамасыз етуіңіз керек (өйткені бұл проблема программалау тіліне қатысты емес). Әдетте, қарапайым (келеңсіз, толық тексерілмейтін) болып келетін, `int` типін нұсқауышқа түрлендіретін `reinterpret_cast` операторы қосымша мен онша қарапайым болып табылмайтын аппараттық ресурстар арасындағы байланысу тізбегінің негізгі бөлігі (звеносы) болып саналады.

Егер тікелей түрлендірулер (`reinterpret_cast`, `static_cast` және т.с.с.; A.5.7 бөлімін қ.) міндетті емес болып табылатын болса, оларды пайдаланбауға тырысыңыз. Мұндай түрлендірулер (келтірулер) негізінен C және C++ (C тілі стилінде) тілінде жұмыс істейтін программалаушылар ойлағандай емес, өте сирек қажет болады.

25.4.2 Проблема: дисфункционалдық интерфейс

18.5.1 бөлімінде көрсетілгендей, жиым көбінесе функцияға, элементке нұсқауыш ретінде беріледі (бірінші элементке нұсқауыш ретінде өте жиі). Осының нәтижесінде ол көлемін "жоғалтады", сондықтан оны қабылдайтын функция нұсқауыш сілтеп тұрған элементтер санын тікелей анықтай алмайды. Осы әрекет оңай табыла қоймайтын және күрделі түрде түзетілетін көптеген қателердің туындауына себепші болады. Мұнда біз жиымдар мен нұсқауыштарға байланысты пайда болатын проблемаларды қарастырамыз және оған балама тәсілдерді көрсетеміз. Ең жаман интерфейс мысалынан бастаймыз (өкінішке орай, жиі кездеседі) да, оны жақсартуға талпынып көреміз.

```
void poor(Shape* p, int sz)   // интерфейстің нашар жобасы
{
        for (int i = 0; i<sz; ++i) p[i].draw();
}
```

```
void f(Shape* q, vector<Circle>& s0)      // өте нашар код
{
      Polygon s1[10];
      Shape s2[10];
      //  initialize
      Shape* p1 = new Rectangle(Point(0,0),Point(10,20));
      poor(&s0[0],s0.size()); // #1 (жиымды вектордан беру)
      poor(s1,10);            // #2
      poor(s2,20);            // #3
      poor(p1,1);             // #4
      delete p1;
      p1 = 0;
      poor(p1,1);             // #5
      poor(q,max);            // #6
}
```

`poor()` функциясы дұрыс жасалмаған интерфейс мысалы болып табылады: ол өзін шақыратын модульге қатеге себепші болатын көптеген мүмкіндіктер береді және оны іске асыру барысында олардан қорғануға ешқандай жол да бермейді.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Ары қарай оқуды жалғастырмас бұрын сіз `f()` функциясынан неше қате табатыныңызды анықтауға тырысып көріңіз. Анығын айтар болсақ, `poor()` функциясын шақырудың қайсысы программа жұмысын тоқтатып тастайды, яғни қателік алып барады?

Бір қарағанда, бұл шақыру өте дұрыс сияқты болып көрінеді, бірақ бұл кодты жөндеп түзету ісі программалаушыларға ұйқысыз түндер әкеледі және сапасы жағынан да ол инженерлерге көптеген қиындықтар туғызады.

1. Типі дұрыс емес, мысалы, `poor(&s0[0],s0.size())` элемент беріледі. Оның үстіне, `s0` векторы бос болуы мүмкін, ондайда `&s0[0]` өрнегі дұрыс болып саналмайды.
2. "Магиялық константа" пайдаланылады (мұндағы жағдайда, ол дұрыс): `poor(s1, 10)`. Элемент типі тағы да дұрыс емес.
3. "Магиялық константа" пайдаланылады (мұндағы жағдайда, ол дұрыс емес): `poor(s2, 20)`.
4. `poor(p1,1)` өрнегінің бірінші шақырылуы дұрыс (оны оңай түсінуге болады).
5. `poor(p1,1)` өрнегінің екінші шақырылуында нөлдік нұсқауышты беру.

6. `poor (q,max)` өрнегінің шақырылуы, мүмкін, дұрыс шығар. Тек код фрагментіне қарап тұрып, оны айту қиын. `q` нұсқауышының `max` элементі бар жиымға сілтеме жасап тұрғанын анықтау үшін, біз `q` нұсқауышы мен `max` айнымалысының анықталуын және олардың осы шақыру кезіндегі мәндерін табуымыз керек.

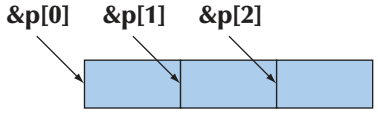
Осы көрсетілген нұсқалардың әрқайсысындағы қателер қарапайым болатын. Біз бұл жерде алгоритмдермен немесе мәліметтер құрылымымен байланысты жасырын қателер тапқан жоқпыз. Мұндағы проблема жиымды нұсқауыш бойынша беруді қарастыратын `poor()` функциясының интерфейсінде болып отыр, ол өз кезегінде, көптеген қателердің шығу көзіне айналып кетеді. Оның үстіне, `pi` және `s0` сияқты түсініксіз атаулар кодты талдауды қиындатып жібереді. Дегенмен, мнемоникалық түрде болғанмен, бірақ дұрыс қойылмаған атаулар одан да күрделі проблемалар туғызуы мүмкін.

Теориялық тұрғыдан қарағанда, компилятор осы қателердің кейбірін таба алады (мысалы, `poor(p1,1)` функциясының екінші шақырылуы, мұндағы, `p1==0`), бірақ практикада біз мұндай апатты жағдайды айналып өттік, себебі компилятор `Shape` абстракты класының объектілерін құруды болдырмады. Бірақ бұл қате `poor()` функциясының нашар интерфейсімен байланысты емес, сондықтан біз босаңсымауымыз керек. Кейіннен абстрактылы болып табылмайтын `Shape` класы нұсқасын пайдаланатын боламыз, сонымен бәрібір біз интерфейс проблемасынан құтыла алмадық.

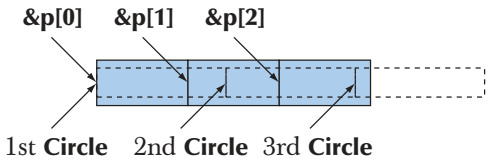
Біз `poor (&s0 [0], s0. size ())` функциясын шақыру қате болып табылады деген шешімге қалай келдік. `&s0[0]` адресі `Circle` класы объектілері жиымының бірінші элементіне қатысты болып келеді; ол `Circle*` нұсқауышының мәні болып табылады. Біз `Shape*` типіндегі аргумент аламыз деп күтеміз де, `Shape` (мұндағы жағдайда `Circle*`) класынан туынды боп келетін класс объектісіне нұсқауыш береміз. Бұл дұрыс болып саналады: объектіге бағытталған программалауды және ортақ интерфейс арқылы (мұндағы жағдайда `Shape` класы көмегімен) әртүрлі типтегі объектілерге қол жеткізуді қамтамасыз ету үшін бізге мұндай түрлендіру керек болады (14.2 бөлімді қ.). Бірақ `poor()` функциясы `Shape*` айнымалысын жай ғана нұсқауыш ретінде қолданып қана қоймайды; ол оны жиым ретінде қолданып, элементтерін индекстейді.

```
for (int i = 0; i<sz; ++i) p[i].draw();
```

Басқаша айтқанда, ол `&p[0]`, `&p[1]`, `&p[2]` және т.с.с. ұяшықтардан бастап, элементтерді іздейді.



Жады ұяшықтарының адрестері терминдерінде бұл нұсқауыштар бір-бірінен `sizeof(Shape)` қашықтығында орналасады (17.3.1 бөлімін қ.). Өкінішке орай, `poor()` функциясын шақыратын модуль үшін `sizeof(Circle)` мәні `sizeof(Shape)` мәнінен артық болады, сондықтан жады аймағын бөлу сұлбасын төмендегідей түрде бейнелей аламыз:



Басқаша айтқанда, `poor()` функциясы `draw()` функциясын `Circle` класы объектісінің ортасына сілтеме жасайтын нұсқауышымен бірге шақырады! Бұл бірден апатты жағдайға душар етеді.

`poor(s1,10)` функциясын шақыру қауіпті сипатта болады. Ол "магиялық константаны" пайдаланады, сондықтан программаны сүйемелдеу кезінде проблемалар пайда бола ма деген күмән бірден туындайды, бірақ бұл тереңірек проблема. `Circle` класы объектілерін пайдалану кезінде біздің байқағанымыз – `Polygon` класы объектілері жиымын пайдалану бірден проблема туғызбауының жалғыз себебі – `Polygon` класы `Shape` базалық класына класс мүшелерін қоспайды (`Circle` класынан айырмасы; 13.8 және 13.12 бөлімдерін қ.), яғни `sizeof(Shape)==sizeof(Polygon)` шарты орындалады және жалпы айтқанда, `Polygon` класының жады аймағын бөлу сұлбасы `Shape` класының жады бөлу тәсілімен бірдей болып келеді. Басқаша айтқанда, мұнда бізге жақсы болды, өйткені `Polygon` класының анықталуын аздап өзгертудің өзі программаның бар жұмысын жоққа шығарады. Сонымен, мұнда `poor(s1,10)` функциясын шақыру жұмыс істеп тұр, бірақ одан қате шықса, ол кейіннен жарылатын мина сияқты өте қауіпті болады. Бұл кодты сапалы деп айтуға болмайды.

Біздің бұдан шығарар қорытындымыз бір әмбебап ереже жасауға негіздеме бола алады, осы ережеге сәйкес "`D` класы – бұл `B` класының бір түрі" деген тұжырымдамадан "`Container<D>` класы – бұл `Container<B>` класының бір түрі " деген тұжырым жасауға болмайды екен (19.3.3 бөлімді қ.). Мысал қарастырайық.

```
class   Circle : public Shape { /* . . . */ };

void fv(vector<Shape>&);
void f(Shape &);
void g(vector<Circle>& vd, Circle & d)
{
     f(d);
     // OK: Circle класын Shape класына
     // тікелей емес түрде түрлендіру
     f(vd);     // қате: vector<Circle> класынан
     // vector<Shape> класына түрлендіру жоқ
}
```

Мұндағы `poor()` функциясының интерфейсі тым нашар, бірақ бұл кодты құрамдас жүйелер тұрғысынан қарастырса қалай болар еді; басқаша айтқанда, қауіпсіздік немесе жұмыс өнімділігі маңызды болып табылатын қосымшаларда осындай проблемалар жайлы тынышсызданууға болар ма екен? Кәдімгі жүйелерді программалауда біз осы кодты қауіпті деп санап оларға: "Бұлай істемеңіз" деп айта аламыз ба? Қазіргі кездегі көптеген құрамдас жүйелер графикалық қолданушы интерфейсіне негізделген, ал олар әрқашанда да объектіге бағытталған программалау қағидаларына сәйкес ұйымдастырылады. Осындай мысалдарға iPod құрылғысының қолдану интерфейсі, кейбір мобильдік телефондардың интерфейстері және де ұшуды басқару жүйелеріндегі операторлар дисплейі жатады. Оның үстіне, осыларға ұқсас құрылғылар контроллерлері (мысалы, көптеген электромоторлар) кластардың классикалық иерархияларын құрайды. Басқаша сөзбен айтқанда, кодтың осы түрі және, жекелеп айтсақ, функцияның осылай жарияланған түрі – ерекше қауіп туғызады. Бізге ақпарат тасымалдауда мұндай ірі проблемалар туғызбайтын бұдан гөрі қауіпсіздеу тәсіл керек.

Сонымен, біз функцияларға нұсқауыштар мен жиым көлемін көрсетіп, құрамдас жиымдарды бергіміз келмейді. Мұны немен ауыстыруға болады? Бәрінен де жеңілі – контейнерге сілтеме жасау, мысалы, `vector` класының объектісіне сілтеме жасауға болады. Келесі көрсетілген функция интерфейсіне байланысты туындаған проблемадан

```
void poor(Shape* p, int sz);
```

мынадай функцияны пайдалана отырып:

```
void general(vector<Shape>&);
```

құтылуға болады.

Егер сіз `std::vector` класының объектілері (немесе солардың эквиваленттері) қолданыла алатын жүйені программалайтын болсаңыз, онда интерфейстерде тізбекті түрде `vector` класын (немесе соның эквивалентін) пайдаланыңыз да, ешқашанда нұсқауыш пен жиым элементтері саны көрсетілген құрамдас жиымдарды функцияға жіберуші болмаңыз.

Егер сіз `vector` класын немесе оның эквиваленттерін пайдаланумен шектеле алмайтын болсаңыз, онда класты (`Array_ref`) қолдану сұранып тұрғанның өзінде сіз қарапайым шешім болмайтын аймаққа кіріп кетесіз.

25.4.3 Шешім: интерфейстік класс

Өкінішке орай, көптеген құрамдас жүйелерде біз `std::vector` класын қолдана алмаймыз, өйткені ол бос жады аймағын пайдаланады. Біз бұл проблеманы `vector` класының ерекше бір іске асырылған нұсқасын ұсынып немесе (одан қарапайымырақ) `vector` класына ұқсайтын, бірақ жады аймағын басқаратын механизмі жоқ контейнерді пайдалану жолымен шеше аламыз. Мұндай интерфейстік класты сипаттамай тұрып, оның бізге қажетті қасиеттерін көрсетейік.

- Ол компьютер жадындағы объектілерге сілтеме жасайды (ол объектілерді иеленбейді, оларды орналастырмайды, өшірмейді, т.с.с.).
- Ол өз көлемін (мөлшерін) біледі (сондықтан мүмкін болатын диапазоннан тысқары кеткенін тексере алады).
- Ол өз элементтерінің типін дәлме-дәл біледі (сондықтан типке байланысты қателер туғыза алмайды).
- Оны жұп (нұсқауыш, санауыш) түрінде тасымалдау (көшіру) қиын емес.
- Оны жанамалы түрде нұсқауышқа түрлендіруге болмайды.
- Ол толық диапазоннан оның ішкі бөлігін жеңіл ерекшелей (белгілей) алады.
- Оны құрамдас жиым ретінде жеңіл пайдалануға болады.

"Құрамдас жиым ретінде жеңіл пайдалануға болады" деген қасиетті тек шамамен ғана қамтамасыз етуге болады. Егерде біз мұны дәлме-дәл істей алатын болсақ, онда болдырмауға тырысып отырған қателерімізбен келісуге тура келер еді.

Осындай класс мысалын қарастырайық.

```
template<class T>
class Array_ref {
public:
    Array_ref(T* pp, int s) :p(pp), sz(s) { }

    T& operator[ ](int n) { return p[n]; }
    const T& operator[ ](int n) const { return p[n]; }

    bool assign(Array_ref a)
    {
            if (a.sz!=a) return false;
            for (int i=0; i<sz; ++i) { p[i]=a.p[i]; }
            return true;
    }
```

```
        void reset(Array_ref a) { reset(a.p,a.sz); }
        void reset(T* pp, int s) { p=pp; sz=s; }

        int size() const { return sz; }

        // келісім бойынша көшіру операциялары:
        // Array_ref класы ешқандай да ресуртарға иелік етпейді
        // Array_ref класының сілтемелік семантикасы бар
    private:
        T* p;
        int sz;
    };
```

`Array_ref` класы минималды мөлшерге жақын.

- Онда `push_back()` (оған динамикалық жады керек) және `at()` (оған аластамалар керек) функциялары жоқ.
- `Array_ref` класы сілтеме формасында болады, сондықтан көшіру операциясы жай ғана жұптарды (`p`, `sz`) көшіреді.
- Әртүрлі жиымдарды инициалдай отырып, барлығы да бір типте болатын, бірақ көлемдері әртүрлі `Array_ref` класы объектілерін ала аламыз.
- `reset ()` функциясы арқылы жұпты (`p`,`size`) жаңарта отырып, бұрыннан бар `Array_ref` класының көлемін өзгерте аламыз (көптеген алгоритмдер ішкі диапазонды көрсетуді талап етеді).
- `Array_ref` класында итераторлар интерфейсі жоқ (бірақ қажет болғанда, бұл кемшілікті оңай жоюға болады). Негізінде `Array_ref` класының тұжырымдамасы, яғни концепциясы екі итератормен берілген диапазонды өте еске түсіреді.

`Array_ref` класы өз элементтеріне иелік етпейді және жадыны да басқармайды, ол жай ғана элементтер тізбегіне қол жеткізіп, соларды функцияға беретін механизм болып табылады. Басқаша айтқанда, ол стандартты кітапханадағы `array` класынан басқаша болады (20.9 бөлімді қ.).

`Array_ref` класы объектілерін жасауды жеңілдету үшін бірнеше көмекші функциялар жазамыз.

```
template<class T> Array_ref<T> make_ref(T* pp, int s)
{
    return (pp) ? Array_ref<T>(pp,s) : Array_ref<T>(0,0);
}
```

Егер біз **Array_ref** класы объектілерін нұсқауышпен инициалдасақ, онда оның көлемін тікелей көрсетуіміз керек. Бұл әрине, оның кемшілігі, өйткені көлемін көрсете отырып, жеңіл қателесіп кетуге болады. Оның үстіне, ол нұсқауышты пайдаланатын мүмкіндік ашады, ал ол нұсқауыш туынды класс жиымын базалық класс нұсқауышына жанамалы тәсілмен түрлендіру нәтижесі болып табылады, мысалы, **Polygon[10]** нұсқауышын **Shape*** нұсқауышына түрлендіру сияқты (25.4.2 бөлімінде сипатталған келеңсіз проблема), бірақ кейде біз программалаушыға сенім артуымыз керек.

Біз нөлдік нұсқауыштар мен бос векторларға қатысты сақ болуымыз қажет, өйткені бұлар көбінесе проблемалар туындатып жатады.

```
template<class T> Array_ref<T> make_ref(vector<T>& v)
{
    return (v.size()) ? Array_ref<T>(&v[0],v.size()) :
                                        Array_ref<T>(0,0);
}
```

Мұндағы идея элементтер векторын беруге келіп тіреледі. Біз vector класын таңдап алдық, бірақ ол көбінесе **Array_ref** класы пайдалы болып табылатын жүйелерге сәйкес келе бермейді. Мұның себебі оның осында пайдалануға болатын контейнерлерге (мысалы, пулдарға негізделген контейнерлерге; 25.3.3 бөлімді қ.) ортақ өзекті қасиеттері бар.

Қорытындылай келе компилятор жиым көлемін білетін жағдайлардағы құрамдас жиымдарды өңдеуді қарастырайық.

```
template <class T, int s> Array_ref<T> make_ref(T (&pp)[s])
{
    return Array_ref<T>(pp,s);
}
```

Мұндағы **T(&pp) [s]** қызықты өрнегі **pp** аргументін **T** типіндегі **s** элементтен тұратын жиымға сілтеме жасай отырып жариялайды. Бұл бізге **Array_ref** класы объектісін элементтерінің саны есте сақталған жиыммен инициалдауға мүмкіндік береді. Біз бос жиымды жариялай алмаймыз, сондықтан оның элементтері бар ма, жоқ па, ол жағын тексеруге міндетті емеспіз.

```
Polygon  ar[0];   //   қате: элементтері жоқ
```

Array_ref класының осы нұсқасын пайдалана отырып, біз мысалымызды қайта жазып шыға аламыз.

```
void better(Array_ref<Shape> a)
{
    for (int i = 0; i<a.size(); ++i) a[i].draw();
}

void f(Shape* q, vector<        ircle>& s0)
{
    Polygon s1[10];
    Shape s2[20];
    //  инициалдау
    Shape* p1 = new Rectangle(Point(0,0),Point(10,20));
    better(make_ref(s0));
    // қате: Array_ref<Shape> талап етіледі
    better(make_ref(s1));
    // қате: Array_ref<Shape> талап етіледі
    better(make_ref(s2));
    // OK (түрлендіру талап етілмейді)
    better(make_ref(p1,1));
    // OK: бір элемент
    delete p1;
    p1 = 0;
    better(make_ref(p1,1));
    // OK: элементтер жоқ
    better(make_ref(q,max));
    // OK (егер max айнымалысы дұрыс берілсе)
}
```

Біз программаның бұрынғыдан жақсы болғанын көріп тұрмыз.

- Код қарапайымырақ болды. Программалаушыға енді объектілер көлемі жайлы ойланбаса да болады, бірақ олар керек болып жатса, оны программаның әр жерінде емес, арнайы бір белгіленген орнында ғана (**Array_ref** класының объектілерін жасау кезінде) беру қажет.
- **Circle[]** – **Shape[]** және **Polygon[] – Shape[]** түрлендірулеріндегі типтер мәселелері шешілген.
- **s1** және **s2** объектілерінің элементтері санының қатесі туралы проблема жанамалы (тікелей емес) түрде шешілген.
- **max** айнымалысының (нұсқауыштарды пайдалануға қажет элементтердің басқа санауыштарымен бірге) әлеуеттік проблемасы тікелей көрінетін болды – бұл біз көлемді тікелей көрсете алатын жалғыз орын.
- Нөлдік нұсқауыштар мен бос векторларды пайдалану жанамалы және жүйелі түрде жүзеге асырылмайтын болды.

25.4.4 Мұралау және контейнерлер

Егер біз `Circle` класы объектілерінің коллекциясын (топтамасын) `Shape` класы объектілерінің топтамасы ретінде өңдегіміз келсе, яғни егер `better()` функциясы нақты түрде (бізге бұрыннан таныс `draw_all()` функциясының бір нұсқасы болып табылатын; 19.3.2 және 22.1.3 бөлімдерін қ.) полиморфизмді жүзеге асыруы үшін не істеуіміз керек? Негізінде, біз мұны істей алмаймыз. Типтер жүйесінде `vector<Circle>` типін `vector<Shape>` типі ретінде қарастыруға болмайтынына нақты негіздемелер бар екені 19.3.3 және 25.4.2 бөлімдерінде көрсетілген. Сол себепке байланысты ол `Array_ref<circle>` типін `Array_ref<Shape>` типі ретінде қабылдай алмайды. Егер ол есіңізде болмаса, онда 19.3.3 бөлімді тағы бір оқып шығыңыз, өйткені бұлар қолайсыз болса да, осы сәттегі орындалатын оқиға маңызды болып табылады.

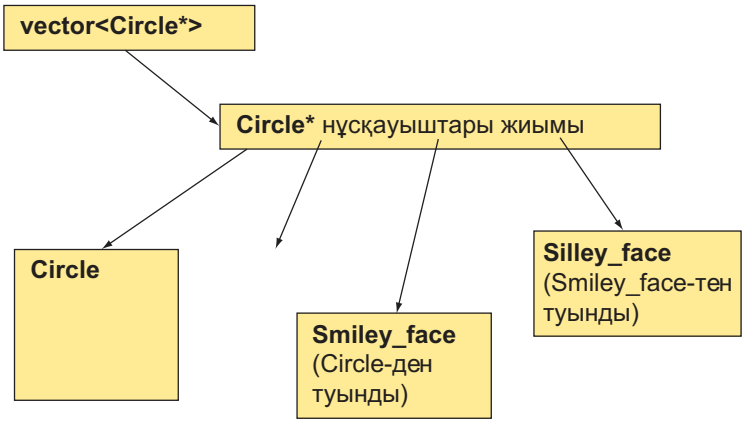
Оның үстіне, динамикалық полиморфизмді сақтау үшін, біз өзіміздің полиморфтық объектілерімізбен нұсқауыштар (немесе сілтемелер) арқылы жұмыс істеуіміз керек: `better()` функциясына кіретін `p[i].draw()` өрнегіндегі нүкте бұл талапқа қарсы болып отыр. Бұл өрнекте тілсызық (`->`) емес, нүкте тұрған соң, енді полиморфизммен мәселелер туындайтыны белгілі болып отыр.

Не істеуіміз керек? Біріншіден, біз объектілердің өздерімен емес, нұсқауыштармен (немесе сілтемелермен) жұмыс істеуіміз керек, сондықтан `Array_ref<Circle>`, `Array_ref<Shape>` кластарын, т.с.с. емес, `Array_ref<Circle*>`, `Array_ref<Shape*>` кластарын, тағы да осыларға ұқсас кластарды пайдалануға ұмтылу керек.

Дегенмен, біз бұрынғыша `Array_ref<Circle*>` класын `Array_ref<Shape*>` класына аудара алмаймыз, өйткені бізге кейіннен `Array_ref<Shape*>` контейнеріне `Circle*` типіндегі емес, басқа типтегі элементтерді орналастыру қажет болуы мүмкін. Бірақ бұдан шығатын бір жол бар.

- Біз `Array_ref<Shape*>` класындағы өз объектімізді өзгерткіміз келмейді; біз жай ғана `Shape` класы объектілерінің суретін салғымыз келеді. Бұл қызықты әрі ерекше кездесетін жағдай: біздің `Array_ref<Circle*>` типін `Array_ref<Shape*>` типіне түрлендіруге қарсы аргументіміз `Array_ref <Shape*>` класын өзгертпеуге тырысатын жағдайларымызға жатпайды.
- Нұсқауыштардың барлық жиымдарының сұлбалары (олар сілтеме жасап тұрған объектілерден тәуелсіз түрде) бірдей болып келеді, сондықтан 25.4.2 бөлімде көтерілген мәселе бізді толғандырмауы керек.

Басқаша айтқанда, егер `Array_ref<Circle*>` класының объектісі `Array_ref <Shape*>` класының өзгертілмейтін объектісі ретінде қарастырылатын болса, одан ешқандай да жамандық бомайды. Сонымен, бізге жай ғана осыны жүзеге асыратын тәсіл табу керек. Мысал қарастырайық.



Нұсқауыштардың `Circle*` типінде берілген осы жиымын `Shape*` типіндегі нұсқауыштардың өзгертілмейтін жиымы (`Array_ref` контейнерінен алынған) ретінде қарастыруға ешқандай да логикалық бөгеттер жоқ.

Біз сарапшылар территориясына кіріп кеткен сияқтымыз. Бұл проблема өте күрделі, оны бұрынғы қарастырылған құралдар көмегімен жоюға болмайды. Дегенмен, егер одан құтыла алсақ, онда біз дисфункционалдық, бірақ әлі де кеңінен қолданылып келе жатқан интерфейске идеалды түрдегі балама (альтернатива) ұсына аламыз (нұсқауыш плюс элементтер саны; 25.4.2 бөлімін қ.). Бірақ мынаны есте сақтаңыз: өзіңіздің соншалықты ақылды екеніңізді көрсету үшін сарапшылар территориясына кіріп кетіп жүрмеңіз. Көптеген жағдайларда бұдан гөрі басқа бір сарапшылар жобалап, жүзеге асырып және сіз үшін тесттен өткізіп қойған кітапхананы тауып алу әлдеқайда пайдалы болып табылады. Біріншіден, біз `better()` функциясын ол нұсқауыштарды пайдаланатындай етіп және ол біздің контейнер аргументтерімен ешнәрсені де шатастырып алмайтынымызға кепілдік беретіндей түрде қайта құрып шығамыз.

```
void better2(const Array_ref<Shape*const> a)
 {
        for (int i = 0; i<a.size(); ++i)
              if (a[i])
                   a[i]->draw();
 }
```

Енді біз нұсқауыштармен жұмыс істейміз, сондықтан нөлдік көрсеткішті тексеруді қарастыруымыз керек. `better2()` функциясының біздің жиымды өзгертпейтініне және векторлардың `Array_ref` контейнерінің қорғанысында болатындығына кепілдік беру үшін, біз бірнеше const квалификаторларын қостық. Бірінші const квалификаторы `Array_ref` класы объектісіне `assign()` және `reset()` сияқты өзгерту операцияларын қолданбайтынымызға кепілдік береді. Екінші const квалификаторы жұдызшадан (`*`) кейін орналасқан. Бұл біздің

өзімізде константалық нұсқауыш (константаларға нұсқауыш емес) болғанын қалайтынымызды білдіреді; басқаша айтқанда, тіпті, бізде мұны орындайтын операциялар болса да, біз элементтерге нұсқауыштарды өзгерткіміз келмейді.

Біз енді ары қарай ең басты проблемадан құтылуымыз керек, ол үшін **Array_ref<Circle*>** класының объектісін **Array_ref<Shape*>** класының объектісіне ұқсас (**better2()** функциясында пайдалануға болатын) басқа бір объектіге түрлендіруді қандай идея арқылы көрсете аламыз деген сауалға жауап беруіміз қажет, бірақ ол **Array_ref<Shape*>** класының объектісі өзгертілмейтін болса ғана жүзеге асады.

Мұны **Array_ref** класына түрлендіру операторын қосу арқылы жасауға болады:

```
template<class T>
class Array_ref {
public:
    // бұрынғыдай сияқты

    template<class Q>
    operator const Array_ref<const Q>()
    {
       // элементтердің тікелей емес түрлендірілуін тексеру:
       static_cast<Q>(*static_cast<T*>(0));
       // Array_ref класын келтіру:
       return Array_ref<const Q>
                              (reinterpret_cast<Q*>(p),sz);
    }
    // бұрынғыдай сияқты
};
```

Бұл бас қатыратын есеп тәрізді, бірақ бәрі бір оның негізгі сәттерін атап өтейік.

Оператор **Array_ref<T>** контейнерінің әрбір элементін **Array_ref<Q>** контейнері элементіне түрлендіре болады деген шарт орындалғанда, әрбір Q типін **Array_ref<cost Q>** типіне келтіреді (біз бұл келтірудің нәтижесін қолданбаймыз, тек осындай келтіруді орындауға болатынын тексереміз).

Біз керекті типтегі элементке нұсқауыш алу үшін "тікелей шешу" тәсілін (оператор **reinterpret_cast**) пайдалана отырып, **Array_ref<const Q>** класының жаңа объектісін жасаймыз. "Тікелей шешу" тәсілі арқылы алынған шешім көбінесе біраз шығын жұмсауды керек етеді; мұндағы жағдайда көпше мұралауды (А.12.4 бөлімі) қолдана отырып, ешқашанда **Array_ref** класына түрлендіруді пайдаланбау керек.

Array_ref<const Q> өрнегіндегі **const** квалификаторына назар аударыңыз: тек сол ғана біздің **Array_ref<const Q>** класының объектісін өзгертуге мүмкіндік беретін ескі **Array_ref<Q>** класының объектісіне көшіре алмайтынымызға кепілдік береді.

Біз сарапшылар территориясына кіріп кеткеніміз жайлы ескеру жасаған болатынбыз, содан болар бас қатыратын есепке де кездестік. Бірақ Array_ref класының осы нұсқасын пайдалану жеңіл орындалады (жалғыз қиындық оның анықталуы мен жүзеге асырылуында).

```
void f(Shape* q, vector<Circle*>& s2)
{
     Polygon* s1[10];
     Shape* s2[20];
     // инициалдау
     Shape* p1 = new Rectangle(Point(0,0),10);
     better2(make_ref(s0));
     // OK: Array_ref<Shape*const>-ке түрлендіру
     better2(make_ref(s1));
     // OK: Array_ref<Shape*const>-ке түрлендіру
     better2(make_ref(s2));
     // OK (түрлендіру қажет емес)
     better2(make_ref(p1,1));    // қате
     better2(make_ref(q,max)); // қате
}
```

Нұсқауыштарды пайдалануға ұмтылу қателерге алып келеді, өйткені олардың типі **Shape***, ал **better2()** функциясы **Array_ref<Shape*>** типіндегі аргументті күтеді; басқаша айтқанда, **better2()** функциясы нұсқауыштың өзін емес, нұсқауышпен байланысты басқа бір нәрсені күтеді. Егер **better2()** функциясына нұсқауышты бергіңіз келсе, онда оны контейнерге (мысалы, құрамдас жиымға немесе векторға) орналастырып қоюыңыз керек, тек содан кейін ғана оны функцияға бере аласыз. Жеке нұсқауыш үшін біз ыңғайсыз мынадай **make_ref(&p1,1)** өрнекті қолдана аламыз. Бірақ бұл шешім жиымдарға (бірден артық элементтері бар) сәйкес келмейді, өйткені ол объектілерге нұсқауыштар контейнерін жасауды қарастырмайды.

Қорытындылай келе айтарымыз, біз жиымдардың кемшіліктерін толтыра алатын қарапайым, қауіпсіз, ыңғайлы және тиімді интерфейстер жасай аламыз. Бұл осы бөлімнің негізгі мақсаты болатын. Дэвид Уилер (David Wheeler) айтқан: "Әрбір проблема жаңа абстракция арқылы шешіледі" деген сөз компьютерлік ғылымдардың бірінші ережесі болып саналады. Сонымен біз интерфейс мәселесін осылай шештік.

25.5 Биттер, байттар және сөздер

Жоғарыда біз компьютер жады құрылғысымен байланысты биттер, байттар және сөздер туралы түсініктер жайлы айтқан болатынбыз, бірақ ашып айтар

болсақ, олар программалаудың негізгі тұжырымдамаларына (концепцияларына) жатпайды. Оның орнына программалаушылар нақты типтегі `double`, `string`, `Matrix` және `Simple_window` сияқты объектілер жайлы ойлайды. Бұл бөлімде біз нақты компьютер жады құрылғысының программалау деңгейіне аздап көз саламыз да, соларды қарастырамыз.

Егер бүтін сандардың екілік және он алтылық түрде бейнеленуі ұмыт бола бастаса, онда А.2.1.1 бөлімін қайталап шығыңыз.

25.5.1 Бит және байтпен орындалатын операциялар

Байт – бұл сегіз биттер тұратын тізбек.

7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

Байт құрамындағы биттер оңнан (ең кіші биттен бастап) солға (ең жоғарғы битке) қарай нөмірленеді. Енді (машиналық) сөзді төрт биттен тұратын тізбек түрінде қарастырайық.

3:	2:	1:	0:
0xff	0x10	0xde	0xad

Сөздегі биттерді нөмірлеу де оңнан солға қарай, яғни төменгі биттерден жоғарғы биттерге қарай жүргізіледі. Бұл сурет нақты жұмыс жағдайын өте идеалды ғып көрсетіп тұр: бір байты тоғыз биттен тұратын да компьютерлер де бар (бірақ соңғы он жылдықта біз олардың бірін де көрмедік), ал бір сөзі екі биттен тұратын машиналар да кездеседі. Бірақ сіздің компьютерде байт сегіз биттен, ал сөз төрт биттен тұрады деп есептейік.

Сіздің биттер ұзындығы, яғни мөлшерлері туралы ұсыныстарыңыз дұрыс болуы үшін және программаңыздың ауысымды болуы үшін, `<limits>` тақырыбын (24.2.1 бөлімін қ.) пайдаланыңыз.

C++ тілінде биттер жиынын қалай бейнелеуге болады? Оның жауабы сізге неше бит керек екендігіне және олармен қандай операцияларды ыңғайлы әрі тиімді түрде орындағыңыз келетініне байланысты болады. Биттер жиыны ретінде бүтін сандық типтерді пайдалануға болады.

- `bool` – бір бит, бірақ ол ұзындығы 8 биттен тұратын ұяшықты алып тұрады.
- `char` – сегіз бит.
- `short` – 16 бит.

- `int` – әдетте, 32 бит, бірақ құрамдас жүйелерде 16 биттік бүтін сан түрінде де бола алады.
- `long int` – 32 немесе 64 бит.

Осы көрсетілген мөлшерлер типтік нұсқа түрінде жиі кездеседі, бірақ әртүрлі орталарда олар да әртүрлі болуы мүмкін, сондықтан әрбір нақты жағдайда оларды тест арқылы тексеріп отыру керек. Оның үстіне, стандартты кітапханалардың биттермен жұмыс істейтін өз құралдары болады.

- `std::vector<bool> – 8*sizeof (long)` биттен артық керек болған кезде қолданылады.
- `std: :bitset – 8*sizeof (long)` биттен артық керек болған кезде қолданылады.
- `std::set` – атаулы биттердің реттелмеген коллекциясы (21.6.5 бөлімді қ.).
- Файл: көптеген биттерден тұрады (25.5.6 бөлім).

Бұға қоса, биттерді бейнелеу үшін C++ тілінің екі мүмкіндігін пайдалануға болады.

- Тізбелер (`enum`); 9.5 бөлімді қ.
- Биттік өрістер; 25.5.5 бөлімді қ.

Мұндай биттерді бейнелеудің көп түрлілігі, негізінде, компьютер жадындағы мәліметтің барлығы да биттер жиынынан тұратындығымен түсіндіріледі, сондықтан адамдар оларды көрудің, ат қоюдың және олармен операциялар орындаудың көптеген тәсілдері болғанын қалайды. Компьютердің аппараттық жабдықтама қамтамасыз ететін логикалық операцияларды ыңғайлы жылдамдықпен орындауы үшін, ондағы құрамдас құралдардың бәрі де биттердің тұрақты санымен (мысалы, 8, 16, 32 және 64) жұмыс істейтіндігіне назар аударыңыз. Ал стандартты кітапхана құралдары оларға қарама-қарсы бағытта биттердің кез келген санымен жұмыс істеуге мүмкіндік береді. Бұл программаның жұмыс өнімділігін шектеуі мүмкін, бірақ бұл туралы алдын ала ойланудың қажеті жоқ: егер сіз таңдап алған биттер саны аппараттық жабдықтаманың талаптарына сай болатын болса, кітапханалық құралдар тиімді түрге келтіріліп, алдын ала оңтайландырылған болып табылады.

Алдымен бүтін сандарды қарастырайық. Бұлар үшін C++ тілінде аппараттық жабдықтама тікелей жүзеге асыра алатын, биттер бойынша орындалатын логикалық операциялар қарастырылған. Осындай операциялар өз операндыларының әрбір битіне қолданылады.

Биттер бойынша орындалатын операциялар		
`|`	Немесе	егер `x` санының `n`-биті немесе `y` санының `n`-биті 1-ге тең болса, `x|y` санының `n`-биті 1-ге тең болады.
`&`	Және	егер `x` санының `n`-биті және `y` санының `n`-биті 1-ге тең болса, `x&y` санының `n`-биті 1-ге тең болады.
`^`	Аластамалы немесе	егер `x` санының `n`-биті немесе `y` санының `n`-биті 1-ге тең болса, бірақ екеуі де қатарынан емес, `x^y` санының `n`-биті 1-ге тең болады.
`<<`	Солға жылжыту	`x<<s` санының `n`-биті `x` санының `(n+s)`-битіне тең.
`>>`	Оңға жылжыту	`x>>s` санының `n`-биті `x` санының `(n-s)`-битіне тең.
`~`	Толықтыру	`~x` санының `n`-биті `x` санының `n`-битіне қарама-қарсы болады.

Сізге іргелі операциялар қатарына біздің "аластамалы немесе" (`^`, оны кейде "`xor`" деп атайды) операциясын қосқанымыз түсініксіз болып отырған шығар. Бірақ бұл операция көптеген графикалық және криптографикалық программаларда маңызды рөл атқарады. Компилятор ешқашанда биттер бойынша орындалатын бұл `<<` логикалық операторды шығару операторымен шатастырмайды, ал сіз қателесуіңіз мүмкін. Бұлай болмас үшін мынаны есте сақтаңыз: шығару операторының сол жақ операнды болып ostream класының объектісі саналады, ал логикалық операторлардың сол жақ операнды – бүтін сан болады.

`&` операторы `&&` операторынан басқаша болатынын атап өтейік, ал `|` операторының `||` операторынан айырмашылығы ол өз операндының әрбір битіне жеке-жеке қолданылады (А.5.5 бөлімі) және олардың нәтижесіндегі биттер саны операндтарындағы биттердің санымен бірдей болады. Ал `&&` және `||` операторлары бұған қарама-қарсы, олар жай ғана `true` немесе `false` мәндерін қайтарады.

Бірнеше мысалдар қарастырайық. Әдетте, биттік комбинациялар он алтылық сан түрінде өрнектеледі. Жарты байт үшін (төрт бит) келесі кодтар қолданылады.

Он алтылық код	Биттер
`0x0`	`0000`
`0x1`	`0001`
`0x2`	`0010`
`0x3`	`0011`
`0x4`	`0100`
`0x5`	`0101`
`0x6`	`0110`
`0x7`	`0111`

Он алтылық код	Биттер
`0x8`	`1000`
`0x9`	`1001`
`0xa`	`1010`
`0xb`	`1011`
`0xc`	`1100`
`0xd`	`1101`
`0xe`	`1110`
`0xf`	`1111`

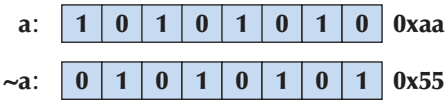
Тоғыздан аспайтын сандар үшін ондық цифрларды пайдалануға болар еді, бірақ он алтылық сан түрінде бейнелеу олардың биттік комбинациялар түрінде екенін

ұмыттырмайды. Байттар және сөздер үшін он алтылық түрде бейнелеу пайдалы да болып табылады. Байт құрамына кіретін биттерді екі он алтылық цифрлар арқылы өрнектеуге болады.

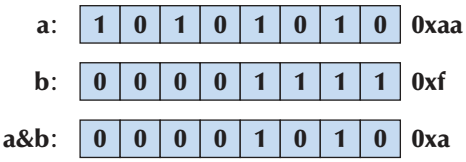
Он алтылық код	Биттер
0x00	0000 0000
0x0f	0000 1111
0xf0	1111 0000
0xff	1111 1111
0xaa	1010 1010
0x55	0101 0101

Сонымен, қарапайымдылығы үшін `unsigned` **типін** (25.5.3 бөлім) пайдаланып, келесі код фрагментін жаза аламыз:

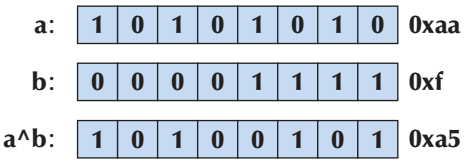
```
unsigned char a  = 0xaa;
unsigned char x0 = ~a;      //  a-ны толықтыру
```



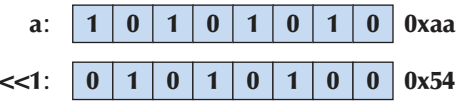

```
unsigned char b  = 0x0f;
unsigned char x1 = a&b;   //   a және b
```




```
unsigned char x2 = a^b;   //  аластамалы немесе: a xor b
```

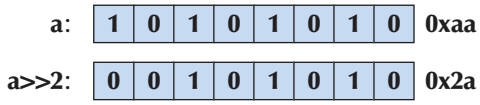



```
unsigned char x3 = a<<1;  // солға бір разрядқа жылжу
```



Жетінші позициядан ары қарай "шығып кеткен" бит орнына бірінші позицияда нөл пайда болады, сонымен байт бәрібір толық болады, ал сол жақ шеткі бит (жетінші) жай ғана жоғалып кетеді.

```
unsigned char x4 == a>>2;  // оңға екі разрядқа жылжу
```



Нөлдік позицияда нөл пайда болады, ол байттың толық болуын қамтамасыз етеді, ал оң жақ шеткі биттер (бірінші және нөлдік) жай ғана жоғалып кетеді.

Біз көптеген биттік комбинациялар жазып, олармен операциялар орындауға жаттығуымызға болады, бірақ олар көп ұзамай жалықтырып та жібереді. Бүтін сандарды олардың биттік бейнелеріне айналдыратын шағын программа қарастырайық.

```
int main()
{
    int i;
    while (cin>>i)
        cout << dec << i << "=="
             << hex << "0x" << i << "=="
             << bitset<8*sizeof(int)>(i) << '\n';
}
```

Бүтін санның жеке биттерін баспаға шығару үшін стандартты кітапхананың `bitset` класы қолданылады.

```
bitset<8*sizeof(int)>(i)
```

`bitset` класы биттердің бекітілген санын есте сақтайды. Мұнда біз `int` типінің ұзындығына (көлеміне) тең биттер санын – `8*sizeof(int)` қолдандық және `bitset` класының объектісін `i` бүтін санымен инициалдадық.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Биттік комбинациялармен жұмыс істеу үшін программаны компиляциядан өткізіңіз және бірнеше сандардың екілік және он алтылық бейнелерін анықтап шығыңыз. Егер сізге теріс сандардың бейнеленуі қиынға соқса, онда 25.5.3 бөлімді қайталап оқып, тағы бір рет байқап көріңіз.

25.5.2 bitset класы

Биттер жиынын бейнелеу үшін және солармен жұмыс істеу кезінде **<bitset>** тақырыбындағы **bitset** стандартты шаблондық класы пайдаланылады. **bitset** класындағы әрбір объектінің оны жасау кезінде бекітілген көлемі болады.

```
bitset<4> flags;
bitset<128> dword_bits;
bitset<12345> lots;
```

bitset класының объектісі келісім бойынша тек нөлдермен инициалданады, бірақ оның өз инициализаторы болады. **bitset** класы объектілерінің инициализаторлары болып таңбасыз бүтін сандар немесе нөлдерден және бірлерден тұратын сөз тіркестері бола алады:

```
bitset<4> flags = 0xb;
bitset<128> dword_bits(string("1010101010101010"));
bitset<12345> lots;
```

Мұнда **lots** объектісі тек нөлдерден тұрады да, ал **dword_bits** – 112 нөлден және соған жалғаса жазылған 16 тікелей берілген биттерден тұрады. Егер сіз **bitset** класы объектісін '0' мен '1'-ден басқа символдардан тұратын тіркеспен инициалдағыңыз келсе, онда **std::invalid_argument** аластамасы туындайтын болады:

```
string s;
cin>>s;
bitset<12345> my_bits(s);
// std::invalid_argument аластамасын туындата алады
```

bitset класы объектілеріне биттермен орындалатын кәдімгі операцияларды пайдалануға болады. **b1**, **b2** және **b3** айнымалылары **bitset** класы объектілеріне жататын болсын делік, сонда келесі операцияларды орындай аламыз:

```
b1 = b2&b3;       // және
b1 = b2|b3;       // немесе
b1 = b2^b3;       // xor
b1 = ~b2;         // толықтыру
b1 = b2<<2;       // солға жылжыту
b1 = b2>>3;       // оңға жылжыту
```

Негізінде, биттік операцияларды (разрядтар бойынша орындалатын логикалық операцияларды) орындау барысында **bitset** класының объектісі өзін қолданушы

берген кез келген көлемдегі **unsigned int** типіндегі айнымалы сияқты ұстайды (25.5.3 бөлім). Біз **unsigned int** типіндегі айнымалымен не істей алатын болсақ (арифметикалық операциялардан басқа), соларды **bitset** класының объектісімен де жасай аламыз. Анықтап айтар болсақ, **bitset** класының объектілері енгізу және шығару операцияларын орындау кезінде өте пайдалы болып табылады.

```
cin>>b;   // bitset класы объектісін енгізу ағымынан оқимыз
cout<<bitset<8>('c');
// 'c' символы үшін биттік комбинацияны шығарамыз
```

bitset класының объектісіне мәліметтерді оқу кезінде енгізу ағымы нөлдер мен бірлерді іздейді. Мысал қарастырайық.

```
10121
```

Осындағы **101** саны енгізіледе, ал **21** саны ағымда қалады.

Байттар мен сөздердегі сияқты **bitset** класының объектілеріндегі биттер оңнан солға қарай нөмірленеді (ең кіші биттен басталады және ең жоғарғы үлкен битпен аяқталады), сондықтан, мысалы, жетінші биттің сандық мәні 2^7 санына тең болады.

7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

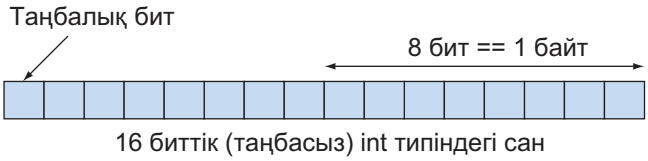
bitset класының объектілері үшін нөмірлеу жай ғана келісім емес, өйткені **bitset** класы биттерді индекстеуді сүйемелдейді. Мысал қарастырайық.

```
int main()
{
    const int max = 10;
    bitset<max> b;
    while (cin>>b) {
      cout << b << '\n';
      for(int i=0; i<max; ++i) cout<<b[i]; // кері тәртіппен
      cout << '\n';
    }
}
```

Егер сізге **bitset** класы жайлы толығырақ ақпарат керек болса, оны Интернеттен, анықтамалықтардан және күрделірек оқулықтардан іздеңіз.

25.5.3 Таңбалы және таңбасыз бүтін сандар

Көптеген программалау тілдеріндегі сияқты, бүтін сандар C++ тілінде де екі түрде: таңбалы және таңбасыз түрлерде болады. Таңбасыз бүтін сандарды компьютер жадында бейнелеу оңай: нөлдік бит бірді білдіреді, бірінші бит – екіні, екінші бит – төртті және т.с.с. жалғаса береді. Бірақ таңбалы бүтін санды бейнелеу аздап проблема туғызады: оң сандарды теріс сандардан қалай ажыратып алуға болады? C++ тілі аппараттық жабдықтама жасаушыларға бірсыпыра таңдау еркіндігін береді, бірақ практика жүзінде, барлық нұсқаларда да екілік толықтыру түрінде бейнелеу пайдаланылады. Мұнда сол жақ шеткі бит (ең жоғарғы үлкен бит) таңбалық бит болып саналады.



Егер таңбалық бит бірге тең болса, онда сан теріс болып саналады. Барлық жерлерде де таңбалы бүтін санды бейнелеу үшін екілік толықтыру тәсілі қолданылады. Орынды үнемдеу үшін төрт биттік таңбалы бүтін санды бейнелеуді қарастырайық:

Оң сан	0	1	2	4	7
	0000	0001	0010	0100	0111
Теріс сан	1111	1110	1101	1011	1000
	-1	-2	-3	-5	-8

`(x+1)` санының биттік комбинациясын `x` (`~x` ретінде де белгілі; 25.5.1 бөлімді қ.) санының биттерінің толықтырылуы деп сипаттауға болады.

Бұған дейін біз тек таңбалы бүтін сандарды (мысалы, `int`) пайдаланған болатынбыз. Таңбалы және таңбасыз бүтін сандарды пайдалану ережелерін былай деп тұжырымдауға болады:

- Сандық есептеулер үшін таңбалы бүтін сандарды (мысалы, `int`) пайдаланыңыз;
- Биттік жиындармен жұмыс істеу үшін таңбасыз бүтін сандарды (мысалы, `unsigned int`) пайдаланыңыз.

Бұл жаман емес эмпирикалық ереже, бірақ оны орындау оңай емес, өйткені кейбір арифметикалық есептеулерде таңбасыз бүтін сандармен жұмыс істеуді ұнататын адамдар бар және бізге кейде солардың программаларын қолдануға тура

келеді. Мысалы, С тілі шыққан алғашқы жылдарда пайда болған тарихи себептерге байланысты, ол кездерде `int` типіндегі сан тек 16 биттен ғана тұрып және әрбір бит есепке алынатын болғандықтан, `vector` класынан алынған `v.size()` функция-мүшесі таңбасыз бүтін сан қайтаратын еді.

Мысал қарастырайық.

```
vector<int> v;
// . . .
for (int i = 0; i<v.size(); ++i) cout << v[i] << '\n';
```

"Ақылды" компилятор біздің таңбалы мән (яғни `i` айнымалысын) мен таңбасыз (яғни, `v.size()`) мәнді араластырып отырғанымызды айтып ескерту жасай алады. Мұндай мәндерді араластыра отырып қолдану апатты жағдайға алып келуі мүмкін. Мысалы, `i` цикл санауышының толып кетуі мүмкін; басқаша айтқанда, `v. size ()` мәні таңбалы `int` типіндегі ең үлкен саннан да артық болып кетуі мүмкін.

Мұндағы жағдайда `i` айнымалысы `int` типіндегі таңбалы (екінің `int` типіндегі биттер санына тең дәрежесінен бірді азайтқандағы мәннен тағы да бірді азайту керек, яғни $2^{15} - 1$) сан арқылы бейнелеуге болатын ең үлкен (максимал) оң мәнге ие бола алады. Сонда келесі `++` операциясы ең үлкен оң бүтін саннан кейінгі мәнді есептеп шығара алмайды да, оның орнына теріс мән қайтарады. Бұл цикл ешқашанда аяқталмайды! Біз циклдегі әрбір ең үлкен оң санға жеткен сайын, бұл циклді қайтадан `int` типіндегі ең кіші теріс саннан басталап қайталай бастаймыз. Сонымен, егер `v. size ()` мәні `32*1024` санына тең болса немесе одан асып кетсе, `int` типіндегі 16 биттік сандар үшін бұл цикл қате (мүмкін, ең ірі қате шығар) болып табылады; `int` типіндегі 32 биттік бүтін сандар үшін бұл проблема тек циклдің `i` санауышы `2*1024*1024*1024` мәніне жеткен кезде ғана туындайды.

Сонымен, жасанды (формальді) көзқарас тұрғысынан алғанда, бұл кітаптағы көптеген циклдер – қате болып саналып, проблемалар туғызатын еді, яғни құрамдас жүйелер үшін циклдің ешқашанда сыни (критикалық) нүктеге жетпейтінін тексеру керек немесе оны басқа конструкциямен алмастыру қажет. Бұл проблеманы тудырмас үшін, біз `vector` класындағы `size_type` типін немесе итераторларды пайдалана аламыз.

```
for (vector<int>::size_type i = 0; i<v.size(); ++i)
            cout << v[i] << '\n';

for (vector<int>::iterator  p = v.begin();
            p!=v.end(); ++p) cout << *p << '\n';
```

`size_type` типінің таңбасы жоқ, сондықтан бүтін сандардың (таңбасыз) бірінші формасы, жоғарыда қарастырылған `int` типінің нұсқасына қарағанда, бейнеленетін бір битке артық болып табылады. Мұның өзіндік мәні болуы

мүмкін, бірақ оның ұлғаятын көлемі тек бір бит болатынын есте сақтау керек (яғни орындалатын операциялар саны екі есеге артық болуы ықтимал). Итераторларды пайдаланатын циклдерде мұндай шектеулер болмайды.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Келесі мысал қарапайым болып көрінуі мүмкін, бірақ ол шексіз цикл ұйымдастырады:

```
void infinite ()
{
    unsigned char max = 160; // өте үлкен
    for (signed char i=0; i<max; ++i)
        cout << int(i) << '\n';
}
```

Осы мысалды орындап, неге бұлай болатынын түсіндіріңіз.

Негізінде, кәдімгі бүтін сандарды бейнелеу үшін биттер жиынын (`+`, `-`, `*` және `/` операцияларын пайдаланбайтын) емес, таңбасыз `int` типін пайдаланудың екі себебі бар:

- Дәлдікті бір битке арттырады.
- Бүтін сандар теріс бола алмайтын жағдайларда, олардың логикалық қасиеттерін бейнелеу мүмкіндігін береді.

Программалаушылар жоғарыда көрсетілген себептерге байланысты таңбасыз цикл санауыштарын пайдаланудан бас тартты.

Таңбалы да және таңбасыз да бүтін сандарды пайдалану кезінде туындайтын бір проблема – C++ тілінде олардың бірінің екіншісіне түрленуі алдын ала болжам жасай алмайтындай және түсініксіз түрде жүргізіліп жатады.

Мысал қарастырайық.

```
unsigned int ui = -1;
int si = ui;
int si2 = ui+2;
unsigned ui2 = ui+2;
```

Таңғаларлық, бірақ бұл факт: бірінші инициалдау табысты түрде өтті де, `ui` айнымалысы 4294967295 мәніне тең болды. Бұл сан 32 биттік таңбасыз бүтін сан, яғни дәл сондай бейнелеу түріндегі (биттік комбинациядағы) таңбасыз -1 санын (кілең бірліктер) көрсетеді. Біреулер мұны пайдалануға болады деп санайды да, -1 санының кіл бірліктерден тұратын қысқаша жазылуы ретінде қабылдайды,

ал басқалары мұны проблема деп есептейді. Осы түрлендіру ережесі таңбасыз сандарды таңбалы сандарға айналдыруға болады деп саналады, сондықтан **si** айнымалысы -1 мәнін қабылдайды. Осыдан барып, **si2** айнымалысы **ui2** айнымалысы тәрізді 1-ге тең болатын шығар (-1+2 == 1) деп күтуге болатын еді. Бірақ бізді **ui2** айнымалысы қайтадан таңғалдырады: неге 4294967295+2 мәні 1-ге тең болады?

4294967295 санын он алтылық сан (**0xffffffff**) деп қарайтын болсақ, онда жағдайды түсіндіруге болады: 4294967295 саны – бұл таңбасыз ең үлкен 32 биттік бүтін сан, сондықтан ол, мейлі, таңбалы немесе таңбасыз сан болса да, 4294967297 санын 32 биттік бүтін сан ретінде бейнелеуге болмайды. Сонымен енді 4294967295+2 операциясы толып кетуге алып келеді деп айту керек немесе (осы дәлірек болады) таңбасыз бүтін сандар модулярлық арифметиканы сүйемелдейді деуіміз керек; басқаша айтқанда, 32 биттік бүтін сандар арифметикасы модулі 32 болып келетін арифметика болып табылады.

Сізге барлығы да түсінікті болды ма? Тіпті, солай болғанның өзінде де, дәлдікті бір битке арттыру үшін таңбасыз бүтін сандарды пайдалану, бұл – отпен ойнау деген сөз. Ол бір жағынан, шатасуға әкеліп соқтыруы мүмкін, ал екінші жағынан, қосымша қателер көзі болып табылады.

Бүтін сан разрядтары толып кеткенде не болады? Мысал қарастырайық.

```
Int i = 0;
while(++i) print(i);
// i-ді бос орыны бар бүтін сан ретінде шығару
```

Енді экранға мәндердің қандай тізбегі шығарылады? Әрине, бұл int типінің (мұндағы **I** бас әріпі қате басылып кеткен символ емес екенін айта кетейік) анықталуына тәуелді болатын шығар. Биттерінің саны шектеулі бүтін типпен жұмыс істей отырып, біз ақырында разрядтардың толып кетуіне душар боламыз. Егер тип **Int** типінің таңбасы болмаса (мысалы, **unsigned char**, **unsigned int** немесе **unsigned long long**), онда операция ++ операциясы модулярлық арифметика операциясы болып табылады, сондықтан біз бейнелеуге болатын ең үлкен саннан кейін нөл (және цикл аяқталады) мәнін аламыз. Егерде **int** типі таңбалы бүті сан болатын болса (мысалы, **signed char**), онда сандар кенеттен теріс болып кетеді де, цикл оның санауышы нөлге тең болғанша (сонда цикл аяқталады), жалғаса береді. Мысалы, signed char типі үшін біз экраннан 1 2... 126 127-128 -127... -2-1 сандарын көреміз.

Бүтін сандардың разрядтары толып кеткенде не болады? Мұндай жағдайда біз өз қолымызда толығынан жетерліктей биттер саны бар сияқты түрде жұмыс істейміз де, нәтиже сақтайтын орындағы жады аймағына сыймай қалған бүтін сан бөлігін алып тастаймыз. Бұл стратегия сол жақтағы ең шеткі (ең үлкен) биттерді жоғалтуға алып келеді. Мұндай нәтижені (эффектіні) келесі код арқылы да алуға болады:

```
int si = 257;     // char типіне сыймайды
char c = si;      // char типіне жанамалы түрде түрлендіру
unsigned char uc = si;
signed char sc = si;
print(si);print(c);print(uc);print(sc);cout << '\n';

si = 129;         // signed char  типіне сыймайды
c = si;
uc = si;
sc = si;
print(si); print(c); print(uc); print(sc);
```

Келесідей нәтиже аламыз:

```
257   1         1         1
129   -127      129       -127
```

Бұл нәтижені былай түсіндіреміз: 257 саны сегіз бит арқылы (255 саны "сегіз бірліктен" тұрады) бейнелеуге болатын ең үлкен саннан екіге артық, ал 129 саны жеті бит арқылы бейнеленетін ең үлкен саннан екіге артық (127 саны "жеті бірлікке" тең), сондықтан таңбалық бит орнатылады. Айтпақшы, бұл программа біздің компьютердегі **char** типінің таңбасы жоқ екенін көрсетеді (**c** айнымалысы өзін **uc** айнымалысы сияқты ұстайды да, ол **sc** айнымалысынан басқаша болады).

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Осы биттік комбинацияларды қағаз парағына жазып шығыңыз. Сонан соң **si=128** мәні үшін нәтижені есептеп шығаруға тырысыңыз. Мұнан кейін программаны орындаңыз да, өз ойыңыздағыны мәнді компьютерде есептелген мәнмен салыстырып көріңіз.

Айтпақшы, біз неге **print()** функциясын пайдаландық? Біз шығару операторын пайдалана алатын едік қой.

```
cout << i << ' ';
```

Дегенмен, егер **i** айнымалысы **char** типінде болған болса, біз экраннан бүтін сан емес, символ көретін едік. Осы себепке байланысты, барлық бүтін сандық типтерді бірыңғай түрде өңдеу үшін біз **print()** функциясын анықтап алдық.

```
template<class T> void print(T i) { cout << i << '\t'; }

void print(char i) { cout << int(i) << '\t'; }
```

```
void print(signed char i) { cout << int(i) << '\t'; }

void print(unsigned char i) { cout << int(i) << '\t'; }
```

Қорытынды: сіз таңбалы бүтін сандар орнына (кәдімгі арифметиканы қоса) таңбасыз бүтін сандарды қолдана аласыз, бірақ мұны істемеуге тырысыңыз, өйткені бұл сенімді емес және қателерге де алып келуі мүмкін.

- Дәлдікті тағы бір битке арттыру үшін ешқашанда таңбасыз бүтін сандарды пайдаланбаңыз.
- Егер сізге қосымша бір бит қажет болып жатса, кейінірек тағы да бір бит керек болады.

Өкінішке орай, біз таңбасыз бүтін сандар арифметикасынан түбегейлі түрде бас тарта алмаймыз.

- Стандартты кітапханада контейнерлерді индекстеу ісі таңбасыз бүтін сандар арқылы орындалады.
- Кейбір адамдар таңбасыз сандар арифметикасын ұнатады.

25.5.4 Биттермен жұмыс істеу (манипуляция)

Жалпы биттермен жұмыс істеу (манипуляция жасау) не үшін керек? Біздің көпшілігіміз мұны істемеген болар едік. "Биттерді сылап-сыйпау" істері төменгі деңгей командаларына жатады да, олар қателерге жол ашады, сондықтан, егер де сізде соның басқа бір баламалы мүмкіндігі болып жатса, соны пайдалану керек. Бірақ биттер соншалықты маңызды әрі пайдалы, сол себепті көптеген программалаушылар оларды айналып өте алмайды. Бұл даулы әрі дөрекілеу ескерту болуы мүмкін, бірақ ол дұрыс айтылған. Кейбір адамдар, шынында да, биттер мен байттарды өңдеуді ұнатады, сондықтан олармен жұмыс істеу кейде қажет те болып жатады (тіпті, қуантарлық нәтиже де беруі мүмкін), бірақ оларды шамадан тыс қолданбау керек. Джон Бентлидің айтқанынан үзінді келтіре кетейік: "Биттерді қолданғандар бұрыс кетеді" ("People who play with bits will be bitten").

Сонымен, біз биттерді қай кездерде қолдануымыз керек? Кейде олар біздің пәндік аймақтың табиғи объектілері болып жатады, сондықтан мұндай қосымшаларда табиғи операциялар ретінде биттерді өңдеу істері қарастырылады. Мұндай қосымшалар мысалдары болып аппараттық жабдықтама индикаторлары ("ылғалдар"), төменгі деңгейдегі коммуникациялар (байттар ағымынан әртүрлі типтегі мәндер алу қажет болған кезде), графика (бейнелердің бірнеше деңгейлерінен сурет құрастыру қажет болғанда) және кодтау істері (ол туралы келесі бөлімде толығырақ) саналады.

Мысал ретінде бүтін саннан (біз оны байттар жиыны ретінде енгізу-шығарудың екілік механизмі арқылы тасымалдауымыз керек болған кездерде) қалай ақпарат (төменгі деңгейдегі) алуға болатынын қарастырайық.

```
void f(short val)
// сан 16 биттен, яғни 2 байттан тұратын болсын делік
{
     unsigned char left = val&0xff;
     //  сол жақ шеткі (ең үлкен) байт
     unsigned char right = (val>>8)&0xff;
     // оң жақ шеткі (ең кіші) байт
     // . . .
     bool negative = val&0x8000;  // таңбалық бит
     // . . .
}
```

Мұндай операциялар жиі кездеседі. Олар жалпы "ығыстыру және бетперде құру" ("shift and mask") ретінде белгілі. Біз керекті биттерді, оларды өңдеу жеңіл орындалатын, оң жаққа (сөздің төменгі бөлігіне) жылжыту үшін, **<<** немесе **>>** операторларын пайдалана отырып, керекті бағытта ығыстырамыз ("shift"). Нәтижеден, онда болмауы тиіс биттерді (нөлге тең етіп орнату) алып тастау үшін, биттік комбинациямен (ол мұнда **0xff**) бірге "және" (**&**) операторын қолдана отырып, бетперде ("mask") құрамыз.

Биттерге ат беру (қою) қажет болып жатса, көбінесе, тізбелер қолданылады. Мысал қарастырайық.

```
enum Printer_flags {
     acknowledge=1,
     paper_empty=1<<1,
     busy=1<<2,
     out_of_black=1<<3,
     out_of_color=1<<4,
     // . . .
};
```

Бұл код арқылы мұнда жаңа тізбе анықталады, онда әрбір элемент өз атына сәйкес келетін мәнге ғана тең болып табылады.

out_of_color	16	0x10	0001 0000
out_of_black	8	0x8	0000 1000
busy	4	0x4	0000 0100
paper_empty	2	0x2	0000 0010
acknowledge	1	0x1	0000 0001

Мұндай мәндер пайдалы болып саналады, өйткені олар бір-бірінен тәуелсіз түрдегі комбинациялық байланыс арқылы құрылады.

```
unsigned char x = out_of_color | out_of_black;
// x = 24   (16+8)
x |= paper_empty;          // x = 26   (24+2)
```

Мұнда `|=` операторын "битті орнату" (немесе "әйтеуір бір битті орнату") деп оқуға болатынын атап өтейік. Сонымен, `&` операторын "бит орнатылған ба?" деп оқуға болады екен. Мысал қарастырайық.

```
if (x&  out_of_color) {
//out_of_colorорнатылғанба? (Иә, егеролорнатылғанболса)
//  . . .
}
```

`&` операторын бұрынғыша бетперде қою үшін пайдалануға болады.

```
unsigned char y = x &(out_of_color | out_of_black);
// y =  24
```

Енді `y` айнымалысы `x` санының 4 және 4 позициясындағы биттердің көшірмесін сақтайды (`out_of_color` және `out_of_black`).

Көбінесе `enum` типті айнымалылар биттер жиыны ретінде пайдаланылады. Мұндайда нәтиже тізбе түрінде болуы үшін кері түрлендіру жасау керек болады. Мысал қарастырайық.

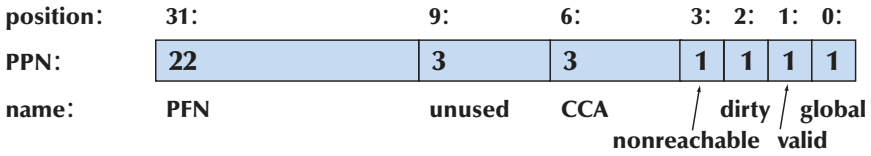
```
// қажетті келтіру операциясы
Flags z = Printer_flags(out_of_color | out_of_black);
```

Мұнда келтіру операциясы қажет, өйткені компилятор `out_of_color | out_of_black` өрнегінің нәтижесі `Flags` типіндегі айнымалының дұрыс мәні болып табылатынын білмейді. Компилятордың білмейтінін негіздеуге болады: басқаларын айтпағанның өзінде, тізбенің бір де бір элементінің 24-ке (`out_of_color | out_of_black`) тең мәні жоқ, бірақ мұнда орындалған меншіктеудің мағынасы бар екенін біз білеміз (ал компилятор білмейді).

25.5.5 Биттік өрістер

Бұрын көрсетілгендей, аппараттық жабдықтамалар интерфейсін программалау кезінде биттермен жұмыс істеу жиі кездеседі. Көбінесе мұндай интерфейстер әртүрлі мөлшердегі (көлемдегі) биттер мен сандардың араласуы ретінде

анықталады. Осындай биттер мен сандардың өзіндік аттары болады да, олар *құрылғы регистрі* (device register) деп аталатын машиналық сөздегі берілген позицияларда орналасады. C++ тілінде осындай бекітілген сұлбалармен жұмыс істейтін *биттік өрістер* (bitfields) деп аталатын арнайы конструкция бар. Парақтар менеджері пайдаланатын бір парақ нөмірін қарастырайық, ол әдетте, операциялық жүйенің ішінде тереңірек орналасады. Төменде операциялық жүйемен жұмыс істеу жайлы жазылған жетекші құралда берілген диаграмма келтірілген.



32 биттік сөз екі сандық өрістен (біреуі 22 биттен, екіншісі 3 биттен тұрады) және төрт жалаушадан (әрқайсысы бір-бір биттен) тұрады. Бұл фрагменттердің көлемдері мен позициялары тұрақты болады. Сөз ішінде тіпті, пайдаланылмайтын (аты да жоқ) өріс те бар. Бұл сұлбаны келесі құрылым арқылы сипаттауға болады:

```
struct PPN {        //Физикалық парақ нөмірі R6000 Number
    unsigned int PFN : 22;
    //Парақ блогы нөмірі қолданылмайды
    int : 3 ;              // unused
    unsigned int A : 3 ;
    // Кеш когеренттігін сүйемелдеу алгоритмі
    // (Cache Coherency Algorithm)
    bool nonreachable : 1 ;
    bool dirty : 1 ;
    bool valid : 1 ;
    bool global : 1 ;
};
```

PFN және **CCA** айнымалыларын таңбасыз бүтін сан түрінде түсініп білу үшін анықтаманы оқып шығу керек. Бірақ біз оның құрылымын диаграмма бойынша қалпына келтіре аламыз. Биттік өрістер сөзді солдан оңға қарай толтырады. Биттер саны қос нүктеден кейінгі бүтін сан ретінде көрсетіледі. Абсолюттік позицияны (мысалы, 8 бит) көрсетуге болмайды. Егер биттік өрістер сөзден артық жады аймағын алатын болса, онда бірінші сөзге сыймай қалған өрістер оның келесісіне жазылады. Бұлар сіздің ойларыңызға қайшы келмейтін болар. Анықталғаннан кейін биттік өрістер басқа айнымалылар сияқты қолданыла береді.

```
void part_of_VM_system(PPN * p )
{
    // . . .
    if (p->dirty) {      // мазмұны өзгерді
       // дискіге көшіреміз
       p->dirty = 0 ;
    }
    // . . .
}
```

Сөз ортасында орналасқан ақпаратты алу үшін биттік өрістер ығыстырулар мен бетперделер қоюды пайдаланбауға мүмкіндік береді. Мысалы, егер **PPN** класының объектісі **pn** деп аталатын болса, онда **CCA** биттік өрісін келесідей түрде шығарып алуға болады:

```
unsigned int x = pn.CCA; //CCA биттік өрісін шығарып аламыз
```

Егерде сол биттерді бейнелеу үшін **pni** атты **int** типіндегі бүтін санды пайдаланған болсақ, онда біз мынадай код жазған болар едік:

```
unsigned int y = (pni>>4)&0x7;
// CCA биттік өрісін шығарып аламыз
```

Басқаша айтқанда, **CCA** өрісі сол жақ шеткі бит болуы үшін, бұл код **pn** құрылымын оңға ығыстырады да, сонан кейін қалған биттерге **0x7** бетпердесін құрады (яғни соңғы үш битті орнатады). Егер сіз осының машиналық кодына қарасаңыз, онда сіз туындаған кодтың жоғарыдағы екі жолға сәйкес келетінін байқайсыз.

Мынадай аббревиатуралардың (**CCA**, **PPN**, **PFN**) араласуы төменгі деңгейдегі кодтарға сәйкес келеді де, олар өз тұрған орнынан (контексінен) басқа жерде онша мәлімет бере алмайды.

25.5.6 Мысал: қарапайым шифрлеу

Биттер мен байттар деңгейінде мәліметтермен жұмыс істеудің (манипуляция) мысалы ретінде шифрлеудің қарапайым Tiny Encryption Algorithm (TEA) алгоритмін қарастырайық. Оны Дэвид Уилер (David Wheeler) Кембридж университетінде (22.2.1 бөлімді қ.) жасап шығарған болатын. Ол онша үлкен емес, бірақ оның шифрын рұқсатсыз қалпына келтіруден керемет қорғанысы бар.

Бұл кодты тым тереңірек қарастырудың (егер сіз тым қызығушылық көрсетпесеңіз немесе басыңызды ауыртқыңыз келмесе) қажеті жоқ. Біз мұны тек сіздің нақты қосымшаны көріп, биттермен жұмыс істеудің пайдалы екендігін сезінуіңіз үшін келтіріп отырмыз. Егер шифрлау мәселелерімен айналысқыңыз келсе, басқа оқулықты қарауыңыз керек. Осы алгоритм жайлы ақпарат алып,

оны басқа программалау тілдерінде іске асыру нұсқаларымен танысқыңыз келсе, http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm парағын немесе Брэдфорд Университетінің (Bradford University, Англия) профессоры Саймон Шепердің (Simon Shepherd) TEA алгоритміне арналған сайтын қарап шығыңыз. Оның коды онша қарапайым емес (комментарийлері жоқ).

Шифрлау/қалпына келтіру (кодтау/қалпына келтіру) ісінің негізгі идеясы қарапайым. Мен сізге бір мәтін жібергелі отырмын, бірақ оны басқа біреудің оқығанын қаламас едім. Сондықтан мен мәтінімді оны қалай өзгерткенімді білмейтін адамдар оқи алмайтындай етіп түрлендіріп жіберемін, бірақ сіз оны өзгерту тәсілін білесіз, сондықтан қалпына келтіріп оқи аласыз. Осы процедура шифрлау болып табылады. Мәтінді шифрлау үшін мен бір алгоритмді (оны басқалар білмеуі тиіс) және оған қосымша кілт деп аталатын бір жолды қолданамын. Сізде осы кілт болуы тиіс (және ол басқаларда жоқ деп есептеледі). Сіз шифрланған мәтінді алғаннан соң, оны кілт арқылы мен жазған алғашқы қалпына келтіресіз.

Шифрлау үшін TEA алгоритмі аргумент ретінде `long` типіндегі екі таңбасыз сан `(v[01, v[1])` алады; ол сегіз символдан құралған `long` типіндегі екі таңбасыз саннан (`w[0], w[1]`) тұратын жиым түрінде болады, оған шифрлау нәтижесі жазылады; және де бұған қосымша кілт ретінде `long` типіндегі таңбасыз төрт саннан (`k[0] .. k[3]`) тұратын жиым беріледі.

```
void encipher(
     const unsigned long *const v,
     unsigned long *const w,
     const unsigned long * const k)
{
     unsigned long y = v[0];
     unsigned long z = v[1];
     unsigned long sum = 0;
     unsigned long delta = 0x9E3779B9;
     unsigned long n = 32;

     while(n-- > 0) {
        y += (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
        sum += delta;
        z += (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
     }
     w[0]=y; w[1]=z;
}
```

Мұндағы барлық мәліметтердің де таңбасы жоқ болғандықтан, теріс сандармен байланысты болатын келеңсіздіктерден қорықпай-ақ, биттер бойынша атқарылатын операцияларды орындай береміз. Негізгі есептеулер ығыстырулар (`<<` және `>>`), аластамалы "немесе" (`^`) және биттер бойынша "және" (`&`) операцияларымен бірге кәдімгі қосу (таңбасыз) амалы арқылы орындалады. Бұл код `long` типі төрт байттан тұратын машина үшін арнайы түрде жазылған.

Бірақ код магиялық константалармен (мысалы, ол **sizeof** (**long**) мәні 4-ке тең деп санайды) ластанған. Әдетте, бұлай істеу ұсынылмайды, бірақ осындағы нақты кодта мұның барлығы бір парақпен ғана шектеледі, оны зердесі жақсы программалаушы математикалық формула ретінде есте сақтауы тиіс. Дэвид Уиллер өз мәтіндерін сапарда жүргенде ноутбуксіз және де басқа құрылғылардың да көмегінсіз шифрлағысы келді. Кодтау және оны қалпына келтіру программасы шағын болуымен қатар, әрі жылдам да істейтін болуы керек. **n** айнымалысы итерациялар (қадамдар) санын анықтайды: итерациялар саны артқан сайын, шифр мықты болып саналады. TEA алгоритміндегі қадам саны **n==32** болғанда, оны ешкім де таба алмағаны белгілі.

Кодтауға сәйкес оны қалпына келтіру функциясын келтірейік:

```
void decipher(
     const unsigned long *const v,
     unsigned long *const w,
     const unsigned long * const k)
{
     unsigned long y = v[0];
     unsigned long z = v[1];
     unsigned long sum = 0xC6EF3720;
     unsigned long delta = 0x9E3779B9;
     unsigned long n = 32;
     // sum = delta<<5, жалпы sum = delta * n
     while(n-- > 0) {
        z -= (y << 4^y >> 5) + y^sum + k[sum>>11 & 3];
        sum -= delta;
        y -= (z << 4^z >> 5) + z^sum + k[sum&3];
     }
     w[0]=y; w[1]=z;
}
```

Біз TEA алгоритмін қорғанысы жоқ байланыс арнасы арқылы жіберілетін файл құру үшін пайдалана аламыз.

```
int main()       // жіберуші
{
    const int nchar = 2*sizeof(long);          //   64 бит
    const int kchar = 2*nchar;                 //  128 бит

    string op;
    string key;
    string infile;
    string outfile;
    cout << "енгізуге, шығаруға және кілтке арналған
                                  файл аттарын енгізіңіз:\n";
    cin >> infile >> outfile >> key;
```

```
    while (key.size()<kchar) key += '0'; // кілтті толтыру
    ifstream inf(infile.c_str());
    ofstream outf(outfile.c_str());
    if (!inf || !outf) error("bad file name");

    const unsigned long* k =
       reinterpret_cast<const unsigned long*>(key.data());

    unsigned long outptr[2];
    char inbuf[nchar];
    unsigned long* inptr =
       reinterpret_cast<unsigned long*>(inbuf);
    int count = 0;

    while (inf.get(inbuf[count])) {
        outf << hex;
        // он алтылық сан түрінде шығару қолданылады
        if (++count == nchar) {
            encipher(inptr,outptr,k);
            // алдыңғы нөлдермен толтыру:
            outf << setw(8) << setfill('0') <<outptr[0] << ' '
                 << setw(8) << setfill('0') <<outptr[1] << ' ';
            count = 0;
        }
    }

    if (count) {       // толтыру
        while(count != nchar) inbuf[count++] = '0';
        encipher(inptr,outptr,k);
        outf << outptr[0] << '  ' << outptr[1] << '  ';
    }
}
```

Кодтың негізгі бөлігі болып **while** циклі саналады; қалған бөлігі қосымша рөл атқарады. **while** циклі символдарды **inbuf** енгізу буферіне оқиды да, TEA алгоритміне кезектегі келесі сегіз символ керек болған сайын, оларды **encipher()** функциясына беріп отырады. TEA алгоритмі символдарды тексермейді; негізінде, ол шифрланатын ақпарат жөнінде ешнәрсе білмейді. Мысалы, сіз фотографияны немесе телефон әңгімесін шифрлай аласыз. TEA алгоритміне ол түрлендіретін 64 бит мәлімет (**long** типіндегі таңбасыз екі сан) оның кіріс нүктесіне келіп тұрса болғаны. Сонымен, **inbuf** тіркесіне сілтейтін нұсқауышты аламыз да, оны **unsigned long*** типіндегі таңбасыз нұсқауышқа түрлендіріп, TEA алгоритміне береміз. Кілтпен де осы әрекеттерді орындаймыз; TEA алгоритмі алғашқы 128 битті пайдаланады (**unsigned long** типіндегі төрт сан), сондықтан біз кіріс мәліметті ол 128 бит болатындай етіп толықтырамыз. Соңғы нұсқау, TEA алгоритмінің талаптарына сай, мәтінді 64 битке (8 байтқа) бөлінетіндей етіп нөлдермен толықтырады.

Шифрланған мәтінді қалай жібереміз? Мұнда бізге таңдауға болады, бірақ мәтін, ASCII немесе Unicode кодтарының символдары емес, жай ғана қарапайым биттер жиыны болғандықтан, біз оны кәдімгі мәтін ретінде қарастыра алмаймыз. Мұнда екілік сандар түріндегі енгізу-шығару амалын пайдалануға болар еді (11.3.2 бөлімін қ.), бірақ біз оларды он алтылық сандар түрінде шығарамыз деп шешім қабылдадық.

```
5b8fb57c 806fbcce 2db72335 23989d1d 991206bc 0363a308
8f8111ac 38f3f2f3 9110a4bb c5e1389f 64d7efe8 ba133559
4cc00fa0 6f77e537 bde7925f f87045f0 472bad6e dd228bc3
a5686903 51cc9a61 fc19144e d3bcde62 4fdb7dc8 43d565e5
f1d3f026 b2887412 97580690 d2ea4f8b 2d8fb3b7 936cfa6d
6a13ef90 fd036721 b80035e1 7467d8d8 d32bb67e 29923fde
197d4cd6 76874951 418e8a43 e9644c2a eb10e848 ba67dcd8
7115211f dbe32069 e4e92f87 8bf3e33e b18f942c c965b87a
44489114 18d4f2bc 256da1bf c57b1788 9113c372 12662c23
eeb63c45 82499657 a8265f44 7c866aae 7c80a631 e91475e1
5991ab8b 6aedbb73 71b642c4 8d78f68b d602bfe4 d1eadde7
55f20835 1a6d3a4b 202c36b8 66a1e0f2 771993f3 11d1d0ab
74a8cfd4 4ce54f5a e5fda09d acbdf110 259a1a19 b964a3a9
456fd8a3 1e78591b 07c8f5a2 101641ec d0c9d7e1 60dbeb11
b9ad8e72 ad30b839 201fc553 a34a79c4 217ca84d 30f666c6
d018e61c d1c94ea6 6ca73314 cd60def1 6e16870e 45b94dc0
d7b44fcd 96e0425a 72839f71 d5b6427c 214340f9 8745882f
0602c1a2 b437c759 ca0e3903 bd4d8460 edd0551e 31d34dd3
c3f943ed d2cae477 4d9d0b61 f647c377 0d9d303a ce1de974
f9449784 df460350 5d42b06c d4dedb54 17811b5f 4f723692
14d67edb 11da5447 67bc059a 4600f047 63e439e3 2e9d15f7
4f21bbbe 3d7c5e9b 433564f5 c3ff2597 3a1ea1df 305e2713
9421d209 2b52384f f78fbae7 d03c1f58 6832680a 207609f3
9f2c5a59 ee31f147 2ebc3651 e017d9d6 d6d60ce2 2be1f2f9
eb9de5a8 95657e30 cad37fda 7bce06f4 457daf44 eb257206
418c24a5 de687477 5c1b3155 f744fbff 26800820 92224e9d
43c03a51 d168f2d1 624c54fe 73c99473 1bce8fbb 62452495
5de382c1 1a789445 aa00178a 3e583446 dcbd64c5 ddda1e73
fa168da2 60bc109e 7102ce40 9fed3a0b 44245e5d f612ed4c
b5c161f8 97ff2fc0 1dbf5674 45965600 b04c0afa b537a770
9ab9bee7 1624516c 0d3e556b 6de6eda7 d159b10e 71d5c1a6
b8bb87de 316a0fc9 62c01a3d 0a24a51f 86365842 52dabf4d
372ac18b 9a5df281 35c9f8d7 07c8f9b4 36b6d9a5 a08ae934
239efba5 5fe3fa6f 659df805 faf4c378 4c2048d6 e8bf4939
31167a93 43d17818 998ba244 55dba8ee 799e07e7 43d26aef
d5682864 05e641dc b5948ec8 03457e3f 80c934fe cc5ad4f9
0dc16bb2 a50aa1ef d62ef1cd f8fbbf67 30c17f12 718f4d9a
43295fed 561de2a0
```

МЫНАНЫ ЖАСАП КӨРІҢІЗ

Кілт ретінде `bs` сөзі болған; мәтін қандай болады?

Қауіпсіздік жөніндегі кез келген сарапшы сізге бастапқы мәтінді оның шифрланған түрімен бірге сақтау дұрыс болмайды деп айтар еді. Оның үстіне, ол толтыру процедурасы, екі әріптен тұратын кілт, т.с.с. туралы міндетті түрде ескертулер жасайды, бірақ біздің кітабымыз компьютерлік қауіпсіздікке емес, программалауға арналған ғой.

Біз шифрланған мәтінді оқып, оны бастапқы қалыпқа түрлендіріп өз программамызды тексеріп шықтық. Программа жазу кезінде оның дұрыстығын тексеретін қарапайым тәсілдер жайлы ешқашанда ұмытып жүрмеңіз.

Шифрланған мәтінді қалпына келтіру программасының ортаңғы бөлігі келесідей түрде болады:

```
unsigned long inptr[2];
char outbuf[nchar+1];
outbuf[nchar]=0;      // терминалдық белгі
unsigned long* outptr =
         reinterpret_cast<unsigned long*>(outbuf);
inf.setf(ios_base::hex ,ios_base::basefield);
// он алтылық енгізу

while (inf>>inptr[0]>>inptr[1])  {
      decipher(inptr,outptr,k);
      outf<<outbuf;
}
```

Функцияның он алтылық сандарды оқу үшін қалай пайдаланылғанына назар аударыңыз.

```
inf.setf(ios_base::hex ,ios_base::basefield);
```

Шифрланған мәтінді қалпына келтіру үшін `outbuf` шығару буфері бар, біз келтіруді пайдаланып, оны биттер жиыны ретінде өңдейміз.

TEA алгоритмін құрамдас жүйелерді программалау мысалы ретінде қарастыруға болар ма еді? Ол міндетті емес, бірақ біз мынадай жағдайды – көптеген құрылғылар арқылы қаржы транзакцияларының қауіпсіздігін жабдықтамасыз етіп, оларды қорғау қажет болатын кезді көзге елестетейік. TEA алгоритмі жақсы құрылған құрамдас кодтың көптеген қасиеттерін көрсетеді: ол дұрыстығы күмән тудырмайтын түсінікті математикалық модельге негізделген; оның үстіне ол шағын, жылдам және аппараттық жабдықтаманың ерекшеліктерін тікелей пайдаланады.

encipher() және **decipher()** функциялары интерфейсінің стилі біздің көзқарасымызға сәйкес келе қоймайды. Дегенмен, бұл функциялар C және C++ тілдерінде жазылған программалардың үйлесімділігін жабдықтамасыз ету үшін жасалған болатын, сондықтан оларда C тілінде жоқ C++ тілі мүмкіндіктерін пайдалануға болмайтын еді. Оның үстіне, көптеген "магиялық константалар" математикалық формулалардың тікелей аудармасы болып табылады.

25.6 Программалау стандарттары

Көптеген қателер көзі бар. Ең күрделі және түзетілуі қиын қателер жоғары деңгейдегі қателерді өңдеудің жалпы стратегиясы, белгілі бір стандарттарға сәйкестік (немесе сәйкестіктің жоқтығы), алгоритмдер, идеяларды бейнелеу және тағы солар сияқты жобалық шешімдермен байланысты болып келеді. Бұл проблемалар мұнда қарастырылмайды. Оның орнына біз нашар стильге байланысты туындайтын қателерге, яғни мұндағы кодта программалау тілі құралдары келеңсіз немесе қате кететіндей түрде пайдаланылатын сәттерге тоқталайық.

Программалау стандарттары "фирмалық стиль" орната отырып, екінші проблеманы шешуге тырысады, осыған орай программалаушылар нақты қосымшаға сәйкес келетін C++ тілінің құралдарын пайдалануы тиіс. Мысалы, құрамдас жүйелерге арналған программалау стандарттары **new** операторын қолдануға тыйым салуы мүмкін. Бұдан бөлек, программалау стандарты, барлық стильдерді араластыра отырып, ешнәрсеге шек қоймаған авторлар жазған программаларға қарағанда, екі қарапайым программалаушы жазған программалардың бір-біріне ұқсас болуын жабдықтамасыз ету үшін де керек. Мысалы, программалау стандарты циклдер ұйымдастыру үшін **while** операторларын пайдалануға тыйым салып, тек қана **for** операторларын қолдануды талап етуі де мүмкін. Осыдан барып программалар бірыңғай стильде құрылады да, үлкен жобаларда оларды сүйемелдеу ісі жеңіл жүзеге асады. Негізінде, стандарттар нақты бір программалау аймақтарында кодтарды жазуды жақсартуға арналғандығына назар аударыңыздар және ондай талапты сол саланың программалаушылары ғана орната алады.

C++ тілінің барлық қосымшалары үшін сәйкес келетін және осы тілде жұмыс істейтін барлық программалаушыларды толық қанағаттандыратын жалпылама бір стандарт жоқ. Сонымен, кездесіп жататын осындай проблемаларды шешуге арналған программалау стандарттары шығарылатын есептердің ішкі күрделіліктерін емес, біздің шешімдерімізді өрнектейтін тәсілдер негізінде туындайды. Программалау стандарттары ішкі мәселелерді емес, жұмыс барысында пайда болатын қосымша қиындықтарды болдырмауға атсалысады.

Қосымша қиындықтарды тудыратын деректер көздерінің негізгілерін тізіп көрсетейік.

- *Тым ақылды программалаушылар,* олар өздері түсіне қоймайтын қасиеттерді қолдануға тырысады немесе аса күрделі шешімдер тауып, соларды қолданғанынан ләззат алады.

- *Білімдері жеткіліксіз программалаушылар,* олар тілдің және оның кітапханасының өз есептеріне сәйкес келетін мүмкіндіктерін білмейді.
- *Программалау стильдерінің негізделмеген вариациялары,* бұлар ұқсас есептерді шығару кезінде әртүрлі құралдарды қолдана отырып, осындай жүйелерді сүйемелдейтін программалаушыларды шатастырады.
- *Программалау тілін дұрыс таңдамау,* нақты бір қосымшаға немесе программалаушылардың нақты бір тобына сәйкес келмейтін тіл конструкцияларын пайдалануға әкеліп соқтырады.
- *Кітапханаларды жеткілікті түрде кеңінен пайдаланбау,* бұлар төменгі деңгейдегі ресурстармен жұмыс істеуді күрделендіріп жібереді.
- *Программалау стандарттарын дұрыс таңдай білмеу,* бұлар белгілі бір класс есептерін шығарудың тиімді тәсілін табуға мүмкіндік бермейді немесе қосымша жұмыс көлемін орындауға әкеліп соқтырады, соның салдарынан жаңа проблемалар туындайды, ал соларды болдырмау үшін жасалған программалау стандарттарын қолданбайды.

25.6.1 Программалау стандарты қандай болуы тиіс?

Программалаудың жақсы стандарты жақсы программалар жазуға итермелеуі тиіс; яғни ол нақты жағдайларда программалаушылардың шешімін табуына біраз уақыт кетіретін көптеген шағын сұрақтарына жауап беруі керек. Программалаушылардың бір ескі мақалы "Форма босатады" дейді. Кодтаудың ең жақсы (идеал) стандарты не істеу керек екендігін білдіретін нұсқаулық түрінде болуы тиіс. Бұл дұрыс айтылған сөз, бірақ көптеген программалау стандарттары қиналғанда, не істейтініміз жайлы берер түсініктемесі жоқ, тек тыйым салу тізімдері түрінде беріледі. Қарапайым тыйым салуға арналған нұсқаулықтар пайдалы болуы күмән туғызатын және кейде ызаңды да келтіретін нәрселер болып көрінеді.

Программалаудың жақсы стандартының ережелері оны программалар арқылы тексеру мүмкіндігін беруі тиіс. Басқаша айтқанда, сіз бір программаны жазып болысымен: "Мен программалау стандартының бір де бір ережесін бұзған жоқпын ба?" – деген сұраққа жеңіл жауап беретіндей күйде болуыңыз керек. Программалаудың жақсы стандарты өз ережелеріне тиянақты түрде негіздеме беруі тиіс. Программалаушыларға қарап тұырып: "Сендер осылай істеуге тиіссіңдер!" – дей салу дұрыс шешім емес. Мұндай жауап оларды қанағаттандырмайды, қайта ызасын келтіреді және одан да жаманы, программалаушылар өздері келіспейтін программалау стандартының кейбір бөліктерін дәлелдеп жоққа шығаруға тырысады да, осындай сәттер оларды жұмысынан қалдырып, біраз уақытын алады. Программалау стандарттары сіздің барлық сұрақтарыңызға жауап береді деп күтпеңіз. Тіпті, ең жақсы деген программалау стандарттарының өзі келісімдер

(компромистер) нәтижесі болып табылады да, сіздің тәжірибеңізде ешқашан кездеспесе де, олар кейде проблема туғызатын заттарды тексеруге де тыйым салдырады. Мысалы, көбінесе объектілерге ат берудің өзі түсініксіздік туғызатын келеңсіз ережелерден құралып та жатады. Бірақ адамдар объектілерге ат беру жайлы бұрын бекітілген келісімдерге тоқталады да, қалғандарына тіпті, көңіл бөлмейді. Мысалы, мен **CamelCodingStyle** ("түйетабан стиль". – *Ред. ескертуі)* сияқты идентификатор аттарын өте тұрпайы деп санаймын да, оның орнына **underscore_style** ("астын сызу стилі". – *Ред. ескертуі) тәрізді* атауларды ұнатамын, олар өте түсінікті, онымен көптеген мамандар келіседі деп ойлаймын. Басқа жағынан, көптеген ақылды деп саналатын адамдар мұнымен келіспейді де. Әрине, бір де бір ат беру стандарты барлық адамдарды толық қанағаттандырады деп айту да қиын; бірақ мұндағы жағдайда, басқа да көптеген кездердегі сияқты бірізділік жүйелілігі жоқ ұсыныстардан әлдеқайда басым болатыны сөзсіз...

Енді қорытынды жасайық.

- Программалаудың жақсы стандарты нақты пәндік аймаққа және нақты бір программалаушылар тобына арналады.
- Программалаудың жақсы стандарты тыйым салу түрінде емес, нұсқаулық беру түрінде болуы керек.
  - Кейбір негізгі кітапханалық мүмкіндіктердің ұсыныстары нұсқаулық ережелерді қолданудың ең тиімді тәсілі болып табылады.
- Программалау стандарты – кодты қажетті түрде жазуды сипаттайтын ережелер жиыны, ашып айтар болсақ:
  - идентификаторларға ат беру мен жолдарды туралауды шектейтін тәсіл, мысалы, " Страуструп сұлбасын пайдаланыңыз";
  - тілдің нақты ішкі жиынын қолдануды көрсету, мысалы, "**new** немесе **throw** операторын пайдаланбаңыз";
  - түсініктеме (комментарий) беру ережесін көрсету, мысалы, "Әрбір функцияның не істейтіні жайлы сипаттама болуы тиіс";
  - нақты кітапханаларды пайдалануды талап ету, мысалы, "**<stdio.h>** емес, **<iostream>** кітапханасын қолданыңыз" немесе "C тілі стиліндегі құрамдас жиымдар мен тіркестерді емес, vector және string кластарын пайдаланыңыз".
- Программалау стандарттары көпшілігінің мақсаттары ортақ болып келеді, олар:
  - Сенімділігі;
  - Ауысымдылығы;
  - Сүйемелдеудің ыңғайлылығы;

- Тесттен өткізудің ыңғайлылығы;
- Қайталап қолдану мүмкіндігі;
- Кеңейту мүмкіндігі;
- Жеңіл оқылатындығы.

- Программалаудың жақсы стандарты оның жоқ болғанына қарағанда жақсы болып табылады. Біз бір де бір өндірістік ірі жобаны (яғни, көптеген адамдар қатынасатын және бірнеше жылға созылатын жоба) программалау стандартын жасамай тұрып бастамаймыз.

- Нашар жасалған программалау стандарты оның жоқ болғанынан да жаман болуы мүмкін. Мысалы, C++ тіліндегі программалау стандарттары оның мүмкіндіктерін C тіліне дейін тарылтатын болса, олар қауіпті болып табылады. Өкінішке орай, программалаудың нашар жасалған стандарттары, біз күткеннен гөрі, жиі кездесіп жатады.
- Программалаушылар тіпті, жақсы деген программалау стандарттарының өзін ұнатпайды. Программалаушылардың көпшілігі өз программаларын өздеріне ұнайтындай етіп жазғысы келеді.

25.6.2 Ережелер мысалдары

Бұл бөлімде біз кейбір ережелерді көрсете отырып, оқырмандарға программалау стандарттары жайлы мәлімет бергелі отырмыз. Әрине, біз сіздерге пайдалы деген ережелерді ғана таңдап алдық. Дегенмен, біз 35 парақтан аспайтын бір де бір нақты программалау стандартын көрмеппіз. Олардың көбісі көлемді болып келеді. Сонымен, біз бұл жерде толық ережелер жиынын келтіруге тырыспаймыз. Оған қоса, әрбір жақсы программалау стандарты нақты бір пәндік аймаққа және программалаушылардың да белгілі бір тобына арналып жазылады. Сол себепті біз ешқандай әмбебаптыққа да қол созбаймыз.

Ережелер нөмірленген және олардың (қысқаша) негіздемесі де бар. Біз кейде программалаушылар көңіл бөлмей кететін *ұсыныстар* мен олар міндетті түрде орындауға тиіс *қатаң ережелер* арасындағы айырмашылықтарды да қарастырып шықтық. Әдетте, қатаң ережелер жетекшінің жазбаша берген рұқсатынан кейін ғана өзгертіліп қолданылады. Программадағы ұсыныстың немесе қатаң ережелердің әрбір бұзылған жерлеріне түсініктеме беру талап етіледі. Ережеден кеткен кез келген ауытқулар оның сипаттамасында жазылып көрсетілуі тиіс. Қатаң ереже оның нөмірімен бірге жазылатын `R` бас әрпімен ерекшеленіп тұрады. Ұсыныс нөмірінде кіші `r` әрпі көрсетілетін болады.

Ережелер:

- Жалпы ережелер.
- Препроцессор ережелері.
- Атауларды пайдалану және мәтінді орналастыру ережелері.
- Кластарға арналған ережелер.
- Функциялар мен өрнектерге арналған ережелер.
- Нақты уақыт кезеңінің қатаң шарттары бар жүйелерге арналған ережелер.
- Қауіпсіздік мәселелеріне ерекше талап қоятын жүйелерге арналған ережелер сияқты бірнеше санаттарға (категорияларға) бөлінеді.

Нақты уақыт кезеңінің қатаң шарттары бар жүйелер мен қауіпсіздік мәселелеріне ерекше талап қоятын жүйелерге арналған ережелер тікелей осындай түрде жарияланған жобалар үшін қолданылады.

Нақты әрі жақсы программалау стандарттарына қарағанда, біздің терминология жеткілікті түрде дәл берілген болып саналмайды (мысалы, "қауіпсіздік мәселелеріне ерекше талап қоятын жүйелерге арналған ережелер" деген нені білдіреді), ал ережелер қысқа әрі нұсқа болуы керек.

Бұл ережелер мен JSF++ ережелерінің (25.6.2 бөлімін қ.) ұқсастығы кездейсоқ нәрсе емес, мен өзім тікелей JSF++ ережелерін тұжырымдауға көмектескен болатынмын. Дегенмен, бұл кітаптағы мысалдар осы ережелерді толық сақтамайды – өйткені кітап қауіпсіздік мәселелеріне ерекше талап қоятын жүйелерге арналған программа болып табылмайды ғой.

Жалпы ережелер

R100. Кез келген функция немесе класс коды 200 логикалық жолдан аспауы тиіс (комментарийлер есепке алынбайды).
Себебі: функция немесе класс кодының ұзындығы оның күрделілігінің белгісі болып табылады, сондықтан оларды түсіну және тесттен өткізу қиындап кетеді.

r101. Кез келген функция немесе класс бір экранға толық сыйып тұруы тиіс және бір есепті шығаруы керек.
Себебі. Тек функцияның немесе кластың бөлігін ғана көріп тұрған программалаушы проблеманы толық көрмеуі мүмкін. Бірнеше есепті шешетін функция көбінесе бір есепті ғана шешетін функцияға қарағанда, ұзынырақ және күрделірек болады.

R102. Кез келген программа C++ ISO/IEC 14882:2003(E) тілінің стандартына сәйкес келуі керек.
Себебі. Тілдің кеңейтілген түрлері немесе ISO/IEC 14882 стандартынан

ауытқыған түрлері тұрақсыздау болып келеді, олар нашар анықталған және программаның ауысымдылығын төмендетеді.

Препроцессор ережелері

R200. Бастапқы мәтіндерді басқаратын `#ifdef` және `#ifndef` директиваларынан басқа ешқандай да макростарды пайдалануға болмайды.
Себебі. Макрос көріну аймағын есепке алмайды және типтермен жұмыс істеу ережелеріне бағынбайды. Макростарды пайдалануды бастапқы мәтінді қарап отырып анықтау қиын.

R201. `#include` директивасы тек қана тақырыптық файлдарды (`*.h`) қосу үшін пайдаланылуы тиіс.
Себебі. `#include` директивасы іске асыру нақтылықтарына емес, интерфейс жариялауларына қол жеткізу үшін қолданылуы тиіс.

R202. `#include` директивасы препроцессорға қатынасы жоқ барлық жариялаулар алдында тұруы керек.
Себебі. Файл ортасында орналасқан `#include` директивасын оқырман байқамай қалады да, ол әртүрлі орындардағы әртүрлі атаулардың көріну аймағы әртүрлі болып шешілетініне байланысты түсінбестік туғызады.

R203. Тақырыптық файлдарда (`*.h`) константалық емес айнымалылардың немесе қойылмайтын шаблондық емес функциялардың анықталуы болмауы тиіс.
Себебі. Тақырыптық файлдарда іске асыру нақтылықтары емес, интерфейстердің жариялануларының болуы тиіс. Дегенмен, көбінесе, константалар интерфейс бөлігі ретінде қарастырылады; кейбір өте қарапайым функциялар жұмыс өнімділігін арттыру үшін қойылатындай түрде болуы керек (сондықтан тақырыптық файлдарда жарияланады), ал ағымдағы шаблондық жүзеге асырулар тақырыптық файлдарда шаблондардың толық анықтаулары болғанын талап етеді.

Атауларды пайдалану және мәтіндерді орналастыру ережелері

R300. Бір бастапқы файлдың басынан соңына дейінгі мәтінде келісілген туралау тәсілін қолдану керек.
Себебі. Оқылуы жеңіл және стилі түсінікті.

R301. Әрбір жаңа нұсқау жаңа жолдан басталуы тиіс.
Себебі. Оқылуы жеңіл.
Мысал:

```
int a = 7; x = a+7; f(x,9);      // бұзылды
int a = 7;       // OK
```

```
x = a+7;          // OK
f(x,9);           // OK
```

Мысал:

```
if(p<q) cout<<*p; // бұзылды
```

Мысал:

```
if(p<q)
   cout<<*p;      // OK
```

R302. Идентификаторлар мәлімет беретіндей түрде болуы тиіс.

Идентификаторлар жалпы бекітілген аббревиатурлар мен акронимдерден тұруы тиіс.

Кейбір жағдайларда `x`, `y`, `i`, `j` атаулары және т.б. мәлімет беретіндей түрде болып табылады.

`numberOfElements` түріндегі емес, `number_of_elements` түріндегі стильді пайдалану керек.

Венгр стилін пайдаланбау керек.

Тек типтер, шаблондар және атаулар кеңістігі аттары бас әріптен бастала алады.

Өте ұзын атауларды пайдаланбаңыз.

Мысал: `Device_driver` және `Buffer_pool`.
Себебі. Оқылуы жеңіл.
Ескерту. Астын сызу символынан басталатын идентификаторлар C++ тілі стандарты бойынша қорға (резервке) қойылған, сондықтан оларды пайдалануға болмайды.
Аластама. Пайдаланылатын кітапханадан функцияларды шақыру кезінде онда анықталған атауларды көрсету қажет болуы мүмкін.
Аластама. `#include` директивасы үшін сақтауыш ретінде пайдаланылатын макростар атаулары.

R303. Тек төмендегідей белгілері бойынша ғана ажыратылатын идентификаторларды пайдаланбаған дұрыс.

- Бас әріп пен кіші әріптерді араластыру.
- Астын сызу символының болуы/болмауы.
- `O` әріпін `0` цифрымен немесе `D` әріпімен алмастыру.
- *I* әріпін `1` цифрымен немесе `l` әріпімен алмастыру.
- `S` әріпін `5` цифрымен алмастыру.
- `Z` әріпін `2` цифрымен алмастыру.
- `n` әріпін `h` әріпімен алмастыру.

Мысал: `Head` және `head` `//` бұзылуы
Себебі. Жеңіл оқылуы.

R304. Идентификаторлар тек бас әріптерден немесе асты сызылған бас әріптерден тұрмауы тиіс.
Мысал: `BLUE` және `BLUE_CHEESE` `//` бұзылуы
Себебі. Тек бас әріптерден тұратын идентификаторлар программаға директива арқылы қосылған кітапхананың тақырыптық файлдарында кездесетін макростардың аттарын белгілеу үшін кең қолданылады.

Функциялар мен өрнектерге арналған ережелер

r400. Ішкі көріну аймағындағы идентификаторлар сыртқы көріну аймағындағы идентификаторлармен бірдей болмауы тиіс.
Мысал:

```
int var=9; {int var = 7;   ++var;}
// қате: var өзін өзі бетперделеп тұр
```

Себебі. Жеңіл оқылуы.

R401. Жариялаулардың мүмкіндігінше барынша шағын көріну аймағы болуы керек.
Себебі. Айнымалыны инициалдау мен оны пайдалану, шатасу болмауы үшін, мүмкіндігінше, бір-біріне жақын орналасуы керек; айнымалының көріну аймағынан тысқары шығуы оның ресурстарын босатады.

R402. Айнымалылар инициалдануы тиіс.
Мысал:

```
int var;      // қате: var айнымалысы инициалданбаған
```

*Себебі. И*нициалданбаған айнымалылар қателер шығатын дәстүрлі мәлімет көздері болып табылады.
Аластама. Бірден енгізу ағымынан алынатын мәліметтермен толтырылатын жиымды немесе контейнерді инициалдау міндетті емес.

R403. Келтіру операторларын қолданбаған дұрыс.
Себептері. Келтіру операторлары көбінесе қате көздері болып табылады.
Аластама. `dynamic_cast` операторын пайдалануға рұқсат етіледі.
Аластама. Жаңа стильдегі келтіру амалын аппараттық жабдықтама адрестерін нұсқауыштарға түрлендіру үшін және сырт жақтан (мысалы, графикалық қолданушы интерфейсі кітапханасынан) алынған `void*` типіндегі нұсқауыштарды соларға сәйкес типтердегі нұсқауыштарға түрлендіру үшін пайдалануға болады.

R404. Құрамдас жиымдарды интерфейстерде пайдалануға болмайды. Басқаша айтқанда, функция аргументі ретінде пайдаланылатын нұсқауыш тек жеке элементке нұсқауыш ретінде ғана қарастырылуы тиіс. Жиымдарды тасымалдау үшін `Array_ref` класын пайдаланыңыз.
Себебі. Жиым оны шақыратын функцияға нұсқауыш арқылы беріліп, оның элементтерінің саны берілмейтін жағдайларда, қате туындауы мүмкін. Оның үстіне, жиымды жанамалы түрде нұсқауышқа түрлендіру мен туынды класс объектісін де жанамалы түрде базалық класс объектісіне түрлендіру комбинациясы жадының зиян шегуіне алып келуі мүмкін.

Кластарға арналған ережелер

R500. Ашық мәлімет-мүшелері жоқ кластар үшін `class` түйінді сөзін, ал жабық мәлімет-мүшелері жоқ кластар үшін `struct` түйінді сөзін пайдаланыңыз. Ашық және жабық мүшелері араласып келген кластарды пайдаланбаңыз.
Себебі. Анықтылығы.

r501. Егер кластың сілтемелі типке нұсқауыш болып табылатын деструкторы немесе мүшесі бар болса, онда оның көшіретін конструкторы болуы керек және мұнда көшіретін меншіктеу операторы анықталуы тиіс немесе оған тыйым салынуы керек.
Себебі. Әдетте, деструктор ресурсты босатады. Келісім бойынша, көшіру семантикасы нұсқауыш немесе сілтеме болып табылатын класс мүшелеріне және деструкторы жоқ кластарға да қатысты жиі дұрыс болмай жатады.

R502. Егер кластың виртуалды функциясы бар болса, онда оның виртуалды конструкторы болуы керек.
Себебі. Егер кластың виртуалды функциясы бар болса, онда оны базалық интерфейс класы ретінде пайдалануға болады. Бұл объектіні тек осы базалық класс арқылы пайдаланатын функция оны өшіре алады, сондықтан туынды кластардың жадыны тазарту (өз деструкторлары арқылы) мүмкіндігі болуы керек.

r503. Бір аргументті қабылдайтын конструктор `explicit` түйінді сөзі арқылы жариялануы тиіс.
Себебі. Алдын ала күтілмеген жанамалы түрдегі түрлендірулерді болдырмас үшін.

Нақты уақыт кезеңінің қатаң шарттары бар жүйелерге арналған ережелер

R800. Аластамаларды қолданбаған дұрыс.
Себебі. Нәтижесін болжауға болмайды.

R801. `new` операторын тек іске қосу кезеңінде ғана пайдалануға болады.
Себебі. Нәтижесін болжауға болмайды.

Аластама. Стектен бөлінген жады үшін орналастыру синтаксисін (оның стандартты мәнінде) пайдалануға болады.

R802. `delete` операторын қолданбаған дұрыс.
Себебі. Нәтижесін болжауға болмайды; жады аймағының фрагменттерге бөлінуі орын алуы мүмкін.

R803. `dynamic_cast` операторын пайдалану қажет емес.
Себебі. Нәтижесін айта алмаймыз (операторды дәстүрлі тәсілмен жүзеге асыруда).

R804. `std::array` класынан басқа стандартты кітапханалық контейнерлерді пайдалану керек емес.
Себебі. Нәтижесін айта алмаймыз (операторды дәстүрлі тәсілмен жүзеге асыруда).

Қауіпсіздік сұрақтарына ерекше талаптар қойылатын жүйелерге арналған ережелер

R900. Инкременттеу және декременттеу операцияларын өрнектер элементтері ретінде пайдалану қажет емес.
Мысал:

```
int x = v[++i];        // бұзылу
```

Мысал:

```
++i;
int x = v[i];          // OK
```

Себебі. Мұндай инкрементацияны байқау оңай емес.

R901. Арифметикалық өрнектер деңгейінен төмен операциялар басымдылығы (приоритеті) ережелеріне код тәуелді болмауы тиіс.
Мысал:

```
x = a*b+c;             // OK
```

Мысал:

```
if (a<b || c<=d)
// бұзылу: нұсқауларды жақшаға алыңыз (a<b) және (c<=d)
```

Себебі. C/C++ тілін әлсіз білетін авторлар программаларында приоритеттер жайлы шатасу тұрақты түрде кездесіп отырады.

Біздің нөмірлеуіміз тізбекті түрде жасалмаған, өйткені жалпы жіктеу тәсілін бұзбай, бұған жаңа ережелерді қосып кірістіру мүмкіндігі сақталуы тиіс. Көбінесе,

ережелерді олардың нөмірлері арқылы есте сақтау қалыптасқан, сондықтан олардың нөмірленуі қолданушылардың ренішін туғызуы мүмкін.

25.6.3 Программалаудың нақты стандарттары

C++ тілі үшін программалаудың көптеген стандарттары бар. Олардың басым бөлігін қолдану корпорация қабырғаларынан шықпайтындай түрде шектеулі және жалпы көпшілікке қолжетімді де емес. Көптеген жағдайларда стандарттар пайдалы болып табылады, бірақ, бәлкім, сол корпорацияларда істейтін программалаушылар үшін олар ондай дәрежеде болмай отырған сияқты. Өз пәндік аймақтарында жақсы болып саналатын стандарттар туралы айтып өтейік.

Henricson, Mats, and Erik Nyquist. *Industrial Strength C++: Rules and Recommendations*. Prentice Hall, 1996. ISBN 0131209655. Телекоммуникациялық компаниялар үшін жасалған ережелер жиыны. Өкінішке орай, бұл ережелер біршама ескіріп кетті, оның кітабы ISO C++ стандартына дейін шыққан болатын. Анығын айтар болсақ, оларда шаблондар жайлы да ойдағыдай түрде кеңірек айтылмаған.

Lockheed Martin Corporation. "Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program". Document Number 2RDU00001 Rev C December 2005. Қолданушылардың шектеулі ортасында "JSF++" деген атпен белгілі. Ол – қанатты аппараттардың (ұшақтардың) программалық жабдықтамалары үшін Lockheed-Martin Aero компаниясында жазылған ережелер жиыны. Бұл ережелерді адамдар өмірлерінің қауіпсіздігіне байланысты программалық жабдықтамалар жасайтын программалаушылар жазған болатын (*www.research.att.com/ ~bs/JSF-AV-rules.pdf*).

Programming Research. High-integrity C++ Coding Standard Manual Version 2.4. (*www.programmingresearch.com*).

Sutter, Herb, and Andrei Alexandrescu. C++ *Coding Standards: 101 Rules, Guidelines, and Best Practices.* Addison-Wesley, 2004. ISBN 0321113586. Бұл еңбекті басқаларына қарағанда, метапрограммалау стандартына жатқызуға болады; яғни мұндағы авторлар нақты ережелерді тұжырымдаудан гөрі қандай ережелер және неге жақсы болып саналатынына жауап беруге тырысқан.

Пәндік аймақты, тілді және технологияны білген дұрыс, бірақ олар бірін-бірі алмастыра алмайтынына назар аударыңыз. Қосымша программалардың көпшілігінде – әсіресе, құрамдас жүйелердегі программалауда – операциялық жүйелерді де және/немесе аппараттық жабдықтаманың архитектурасын да білу керек. Егер сізге C++ тілінде төменгі деңгейдегі кодтау жұмысын орындау қажет болса, онда стандарттау бойынша ISO комитетінің жұмыс өнімділігіне арналған есеп беру құжаттарын оқып шығыңыз (ISO/IEC TR 18015; *www.research.att.com/~bs/performanceTR.pdf*); жұмыс өнімділігі ретінде авторлар (және біз де) мұнда құрамдас жүйелердегі программалаудың жұмыс өнімділігін түсінеді.

Құрамдас жүйелер әлемінде көптеген программалау тілдері мен солардың әртүрлі нұсқалары (диалектері) бар, бірақ керекті жерінде сіз соның ішіндегі қолдануға болатын стандартты тілдерін (мысалы, ISO C++), құрал-саймандарын және кітапханаларын пайдалануыңыз қажет. Бұл сіздің оқып үйренуге жіберетін уақытыңызды үнемдейді және сізді жақын арада бұл жұмыстан шығармайтынына кепілдік береді.

ТАПСЫРМА

1. Келесі код фрагментін орындаңыз:

```
int v = 1; for (int i = 0; i<sizeof(v)*8; ++i) {
                        cout << v << ' '; v <<=1; }
```

2. Осы код фрагментін тағы бір рет орындаңыз, бірақ енді `v` айнымалысын `unsigned int` деп жариялаңыз.
3. Оналтылық литералдарды қолдана отырып, келесі `short unsigned int` типтес айнымалылардың неге тең екенін анықтаңыз.
   3.1. Әрбір бит бірге тең.
   3.2. Ең кіші бит бірге тең.
   3.3. Ең үлкен бит бірге тең.
   3.4. Ең кіші байт тек бірліктерден тұрады.
   3.5. Ең үлкен байт тек бірліктерден тұрады.
   3.6. Әрбір екінші бит бірге тең (ең кіші бит те бірге тең).
   3.7. Әрбір екінші бит бірге тең (ал ең кіші бит нөлге тең).
4. Жоғарыда көрсетілген мәндердің әрқайсысын ондық және оналтылық сандар ретінде баспаға шығарыңыз.
5. 3,4-тапсырмаларды биттер бойынша орындалатын операцияларды (`|`, `&`, `<<`) және тек 1 мен 0 литералдарын (тек қана) пайдалана отырып, орындап шығыңыз.

БАҚЫЛАУ СҰРАҚТАРЫ

1. Құрамдас жүйе дегеніміз не? Он мысал келтіріңіз, оның үшеуі осы тарауда келтірілмеген болсын.
2. Құрамдас жүйелердің қандай ерекшеліктері бар? Барлық құрамдас жүйелерге тән бес ерекшелікті келтіріңіз.
3. Құрамдас жүйелер контексінде болжау түсінігін анықтаңыз.
4. Неліктен құрамдас жүйелерді кейде толықтыру және жөндеу қиын?
5. Неліктен жүйенің жұмыс өнімділігін оңтайландыру кейде ұсынылмайды?

6. Неліктен біз программалаудың төменгі деңгейіне түспей, абстракцияның жоғарғы деңгейінде қалғанды жөн деп санаймыз?
7. Қандай қателерді өтпелі деп атайды? Олар несімен аса қауіпті?
8. Өзінің жұмысындағы ақауларды кейін қайтадан қалпына келтіре алатын жүйелерді қалай құруға болады?
9. Ақаулардың алдын алу неліктен мүмкін емес?
10. Пәндік аймақ дегеніміз не? Пәндік аймаққа мысалдар келтіріңіз.
11. Құрамдас жүйелерді программалау кезінде неліктен пәндік аймақты білу қажет?
12. Ішкі жүйе дегеніміз не? Мысалдар келтіріңіз.
13. C++ тілінің тұрғысынан компьютер жадының үш түрін атаңыз.
14. Неліктен сіз бос жады аймағын пайдалануды дұрысырақ деп санайсыз?
15. Неліктен құрамдас жүйедегі бос жадыны пайдалану көбінесе, ұсынылмайды?
16. Құрамдас жүйеде **new** операторын қалай қауіпсіз түрде пайдалануға болады?
17. Құрамдас жүйелер контексінде **std::vector** класымен қандай әлеуетті мәселелер байланысқан?
18. Құрамдас жүйелердегі аластамалармен қандай әлеуетті мәселелер байланысқан?
19. Функцияны рекурсивті түрде шақыру деген не? Құрамдас жүйені құрастырушы кейбір программалаушылар неліктен аластамаларды пайдаланудан қашады? Олар оның орнына нені пайдаланады?
20. Компьютер жадын фрагменттерге бөлу деген не?
21. Қоқыс жинаушы деген не (программалау контексінде)?
22. Жадының азаюы деген не? Ол неліктен мәселеге айналуы мүмкін?
23. Ресурс деген не? Мысал келтіріңіз.
24. Ресурстардың азаюы деген не және оны қалай жүйелі түрде болдырмауға болады?
25. Неліктен біз объектілерді бір жады аймағынан басқасына орын ауыстыра алмаймыз?
26. Стек деген не?
27. Пул деген не?
28. Неліктен стек және пул жадыны фрагменттерге бөлуге әкелмейді?
29. **reinterpret_cast** операторы не үшін қажет? Ол несімен жаман?
30. Функция аргументтері ретінде нұсқауыштарды беру неге қауіпті? Мысалдар келтіріңіз.
31. Нұсқауыштар мен жиымдарды пайдаланған кезде қандай мәселелер туындауы мүмкін? Мысалдар келтіріңіз.
32. Интерфейстердегі нұсқауыштарды (жиымдарға) пайдаланудың баламалы тәсілдерін тізіп көрсетіңіз.

33. Компьютерлік ғылымдардың бірінші заңы қалай айтылады?
34. Бит дегеніміз не?
35. Байт дегеніміз не?
36. Байт неше биттен тұрады?
37. Биттер жиынымен қандай операциялар орындауға болады?
38. Аластамалы "немесе" дегеніміз не және ол несімен пайдалы болып табылады?
39. Биттер жиынтығын (тізбегін) қалай бейнелеуге болады?
40. Сөз (машиналық) неше биттен тұрады?
41. Сөз неше байттан тұрады?
42. Сөз (машиналық) дегеніміз не?
43. Сөз көбінесе, неше биттен тұрады?
44. `0xf7` санының ондық мәні нешеге тең?
45. `0xab` саны қандай биттер тізбегіне сәйкес келеді?
46. `bitset` класы деген не және ол қай кезде керек болады?
47. `unsigned int` типінің `signed int` типінен қандай айырмасы бар?
48. Қандай жағдайларда біз `unsigned int` типін пайдалануды `signed int` типін қолданудан гөрі дұрысырақ деп санаймыз?
49. Егер жиымдағы элементтер саны өте көп болса, циклді қалай ұйымдастыруға болады?
50. `-3` санын меншіктегеннен кейін `unsigned int` типіндегі айнымалы мәні нешеге тең болады?
51. Біздің неге биттермен және байттармен (одан гөрі жоғары дәрежедегі типтермен емес) жұмыс істегіміз келеді?
52. Биттік өріс дегеніміз не?
53. Биттік өрістер не үшін қажет?
54. Кодтау (шифрлеу) дегеніміз не? Ол не үшін қолданылады?
55. Фотографияны шифрлеуге бола ма?
56. TEA алгоритмі не үшін қажет?
57. Санды он алтылық жүйеде қалай шығаруға болады?
58. Программалау стандарттары не үшін керек? Себептерін айтыңыз.
59. Неге әмбебап программалау стандарты жасалмаған?
60. Жақсы программалау стандартының кейбір қасиеттерін тізіп көрсетіңіз.
61. Программалау стандарты қалай зиян тигізе алады?
62. Оннан кем болмайтын программалау ережелерінен (пайдалы болып саналатын) тұратын тізім жасаңыз. Олар несімен пайдалы?
63. Біз неге ALL_CAPITAL түріндегі идентификаторларды пайдаланбаймыз?

ТЕРМИНДЕР

bitset
unsigned
адрес
азаю
аластамалы "немесе"
бит
биттік өріс
болжарлық
қоқыс жинауыш
құрамдас жүйе
құрылғы
нақты уақыт кезеңі
нақты уақыттың қатаң шарттары
нақты уақыттың жұмсақ шарттары
программалау
пул
ресурс
стандарты
шифрлеу

ЖАТТЫҒУЛАР

1. Егер бұған дейін орындамасаңыз, "**МЫНАНЫ ІСТЕП КӨРІҢІЗ**" бөліміндегі жаттығуларды орындап шығыңыз.
2. Сандарды он алтылық санау жүйесінде жазудан алуға болатын сөздер тізімін құрастырыңыз, мұнда `0` `o` болып, `1` `l` болып, `2` `to` болып, т.с.с. жалғастырылып жазыла береді. Мысалы, Fool және Beef. Оларды бағалауға беру алдында, тиянақты түрде тексере отырып, анайы сөздерді алып тастаңыз.
3. 32 биттік бүтін санды инициалдап, баспаға шығарыңыз, ол биттік комбинациялы таңбасы бар бүтін сан болуы тиіс: барлығы нөлдер, барлығы да бірлер, кезектесіп келетін нөлдер мен бірлер (сол жақ шеткі цифры бірден басталады), кезектесіп келетін нөлдер мен бірлер (сол жақ шеткі цифры нөлден басталады), 110011001100, 001100110011, кезектесіп орналасқан тек бірлерден ғана және тек нөлдерден ғана тұратын байттар (сол жақ шеткі байт нөлдерден тұрады). Осы жаттығуды 32 биттік таңбасыз бүтін санмен де қайталап орындап шығыңыз.
4. 7-тараудағы калькуляторға биттер бойынша орындалатын логикалық операторларды `operators` `&`, `|`, `^` және `~` қосыңыз.
5. Шексіз цикл жазыңыз да, соны орындаңыз.
6. Шексіз цикл ретінде көріне қоймайтын (байқалмайтын) шексіз цикл жазыңыз. Негізінде шексіз болып табылмайтын циклді де пайдалануға болады, ол ресурстары аяқталған соң, тоқталуы тиіс.
7. 0-ден 400-ге дейінгі он алтылық мәндерді шығарыңыз; -200-ден 200-ге дейінгі он алтылық мәндерді шығарыңыз.
8. Өз пернетақтаңыздағы әрбір символдың кодын экранға шығарыңыз.
9. Стандартты тақырыптарды да (мысалы, `<limits>` тәрізді), құжаттамаларды да пайдаланбай, `int` типіндегі биттер санын есептеп шығарыңыз да, өз программаңыздағы C++ тілінің `char` типінің таңбасы бар немесе жоқ екенін анықтаңыз.
10. 25.5.5 бөліміндегі биттік өріс мысалын талдап шығыңыз. `PPN` құрылымы инициалданып, оның әрбір өрісінің мәні баспаға шығарылады да, сонан

соң әрбір өрісінің мәні қайта өзгертіліп (меншіктеу арқылы), нәтижесі қайтадан баспаға шығарылатын мысал жазыңыз. `PPN` құрылымындағы ақпаратты 32 биттік таңбасыз бүтін сан түрінде есте сақтап, сол сөздегі әрбір битке қол жеткізу үшін биттермен жұмыс істейтін (25.5.4 бөлімін қ.) операторларды қолданыңыз.

11. Алдыңғы жаттығуды биттерді `bitset<32>` класы объектісінде сақтай отырып қайталап орындаңыз.
12. 25.5.6 бөліміндегі мысал үшін түсінікті программа жазып шығыңыз.
13. Екі компьютер арасында мәлімет тасымалдау үшін TEA алгоритмін (25.5.6 бөлімін қ.) пайдаланыңыз. Электрондық поштаны пайдалану мұқиятты түрде ұсынылмайды.
14. N-нен аспайтын элементтер санын пулдан бөлініп берілген жады аймағында сақтайтын қарапайым векторды іске асырыңыз. Оны `N==1000` болған кезде және бүтін сандық элементтер үшін тесттен өткізіңіз.
15. [1000:0) диапазонынан алынған кездейсоқ мөлшердегі 10 мың объектіні орналастыруға кететін уақытты `new` операторы арқылы өлшеп шығыңыз; сонан кейін `delete` операторы арқылы осы элементтерді өшіруге кететін уақытты өлшеңіз. Мұны екі рет орындаңыз: бір рет жады аймағын кері бағытта босата отырып, ал екінші ретте – кездейсоқ тәртіппен орындап шығыңыз. Сонан кейін осыған эквивалентті тапсырманы мөлшерлері 500 байттан болатын 10 мың объект үшін жады аймағын пулдан бөле және босата отырып орындап шығыңыз. Осыдан кейін [1000:0) байттар диапазонында кездейсоқ мөлшердегі 10 мың объект үшін жады аймағын стектен бөліп беріп және оларды кері бағытта босата отырып орналастырып шығыңыз. Өлшеу нәтижелерін салыстырыңыз. Нәтижелердің үйлесімділігіне сенімді болу үшін әрбір өлшеуді үш реттен кем орындамаңыз.
16. Программалау стилін шектейтін (регламенттейтін) жиырма ереже (25.6 бөліміндегі ережелерді көшіріп алмаңыз) жазып шығыңыз. Оларды өзіңіз жақын арада құрған 300 жолдан кем түспейтін программаға қолданыңыз. Осы ережелерді пайдалану туралы қысқаша комментарий (бір-екі беттен тұратын) жазыңыз. Сіз программадан қателер таптыңыз ба? Кодтарыңыз түсініктірек болып шықты ма? Ол, мүмкін, бұрынғыдан гөрі түсініксіздеу болып шыққан шығар. Енді өз тәжірибеңізге сүйене отырып ережелеріңізді толықтырыңыз.
17. 25.23-25.24 бөлімдерде біз жиым элементтеріне қарапайым және қауіпсіз түрде қол жеткізуді жабдықтамасыз ететін `Array_ref` класын сипаттаған болатынбыз. Ашып айтар болсақ, сол жерде біз енді мұралау дұрыс өңделетін болды деп айтқан едік. `Array_ref<Shape*>` класын пайдалана отырып, типтерге келтіруді және де орындалу барысын болжауға болмайтын басқа да операцияларды қолданбай, `vector<Circle*>` жиымы элементіне `Rectangle*` нұсқауышын алуға болатын әртүрлі тәсілдерді зерттеп шығыңыз. Бұл мүмкін емес болып көрінуі тиіс.

СОҢҒЫ СӨЗ

Сонымен, құрамдас жүйелерді программалау, негізінде, "биттерді толтыруға" келіп тіреле ме екен? Сіз әсіресе, әлеуетті қателер көзі ретінде биттерді толтыруды азайтуға тырысатын болсаңыз, бұл олай емес тәрізді болып көрінеді. Дегенмен кейде жүйедегі биттер мен байттарды "толтыруға" тура келеді; мәселе тек мұны қайда және қалай істеу керек екендігінде. Жүйелердің көпшілігінде төменгі деңгейлі код жекеленіп жазыла алады және солай жазылуы да тиіс. Біз жұмыс істеуге тура келген көптеген қызықты жүйелер құрамдас жүйелер болып келген еді, ал программалаудың ең қызық әрі күрделі есептері де осы пәндік аймақта жиі туындап отырады.

Тесттен өткізу

"Мен тек кодтың дұрыстығын тексердім,
бірақ оны тесттен өткізген жоқпын".

- Дональд Кнут (Donald Knuth)

Бұл тарауда тесттен өткізу мәселелері мен программалар жұмысының дұрыстығын тексеру талқыланады. Бұлар өте ауқымды тақырыптар, сондықтан біз оларды тек үстіртін ғана қарастырып өтеміз. Біздің мақсатымыз – кейбір практикаға бейім идеялар және функциялар мен кластар сияқты модульдерді тесттен өткізу тәрізді әдістерді сипаттау. Біз интерфейстерді пайдалануды және программаларды тексеруге арналған тесттерді таңдауды талқылаймыз. Тесттен өткізуді қарапайым ететін және соларды жұмыстың алғашқы кезеңдерінде қолданатын жүйелерді жобалау мен жасауға көп көңіл бөлінеді. Және де программалардың дұрыстығын дәлелдеу әдістері мен жұмыс өнімділігіне байланысты проблемаларды шешу жолдары да қарастырылады.

26.1 Біз нені қалаймыз?

Қарапайым тәжірибе (эксперимент) жүргізейік. Бинарлық іздеу программасын жазып, оны орындаңыз. Бұл тарауды немесе бөлімді оқып шығудың аяғын күтпеңіз. Сіздің бұл тапсырманы бірден қазір орындауыңыз өте маңызды! Бинарлық іздеу – бұл керекті элементті реттелген тізбектің ішінен оның ортасынан бастап іздеу.

- Егер ортадағы элемент ізделетін элементке тең болса, біз іздеуді доғарамыз.
- Егер ортадағы элемент ізделетін элементтен кіші болса, біз бинарлық іздеуді оң жақ бөлікте жүргіземіз.
- Егер ортадағы элемент ізделетін элементтен үлкен болса, біз бинарлық іздеуді сол жақ бөлікте жүргіземіз.
- Іздеу нәтижесі табыстың индикаторы болып табылады және ол ізделетін элементті нақтылауға көмектеседі. Ол үшін мұндай индикатор ретінде индекс, нұсқауыш немесе итератор қолданылады.

Салыстыру (сұрыптау) критериі ретінде "кіші" (`<`) операторын пайдаланыңыз. Кез келген мәліметтер құрылымын, функцияларды шақыру мен нәтижені қайтарудың да кез келген тәсілін таңдай аласыз, бірақ сол программаны міндетті түрде өзіңіз жазып шығыңыз. Бұл – басқа біреу жазған функцияны пайдалану, егер ол функция тіпті жақсы жазылып шықса да, нәтижесі контрөнімді болып табылатын сирек кездесетін жағдай. Мысалы, стандартты кітапханадан алынған алгоритмдерді (`binary_search` немесе `equal_range`), олар, басқа бір жағдайларда, ең жақсы таңдау болып есептелетін болса да, пайдаланбаңыз. Бұл программаны құруға өте көп уақыт жіберуіңіз мүмкін.

Сонымен, сіз бинарлық іздеу үшін функция жазып шықтыңыз делік. Егер олай болмаса, алдыңғы абзацқа қайтып оралыңыз. Неге сіз өзіңіз жазған іздеу функциясының дұрыс екендігіне сенімдісіз? Программаңыздың дұрыстығын дәлелдейтін аргументтеріңізді келтіріңіз.

Сіз өз аргументтеріңізге сенімдісіз бе? Сіздің аргументтеріңіздің осал жерлері жоқ па? Бұл өте қарапайым және танымал алгоритмді жүзеге асыратын жай ғана программа болатын. Сіздің компиляторыңыздың бастапқы мәтіні 200 Кбайттай жады көлемін алатын болсын, сіздің операциялық жүйеңіздің алатын көлемі 10-нан 50 Мбайт аралығында делік, ал сізді кезекті дем алысыңызға немесе конференцияға алып баратын ұшағыңыздың жұмыс қауіпсіздігін қамтамасыз ететін кодтың көлемі 500 Кбайттан 2 Мбайтқа дейін орын алатын болсын. Бұл сізді қанағаттандыра ала ма? Сіздің бинарлық іздеуге арналған функцияңызды тексеруге арналған әдістерді көлемі мұнан үлкен болып келетін нақты программалық жабдықтамаға қалай қолдануға болар екен?

Барлық күрделіліктеріне қарамастан, программалық жабдықтаманың басым бөлігі біраз уақыт аралығында дұрыс жұмыс істейді екен. Программаларға қойылатын осындай маңызды сыни талаптар санатына біз дербес компьютерлердегі ойын программаларын жатқызбаймыз. Қауіпсіздігіне ерекше талаптар қойылатын осындай программалық жабдықтамалар көбінесе дұрыс жұмыс атқаратынын айта кетейік. Біз бұған байланысты соңғы кездері жұмыстарынан кемшіліктер табылып жатқан авиалайнерлердің борттағы компьютерлеріне немесе автомобильдерге қатысты программалық жабдықтамаларды қарастырмаймыз. Қазіргі кезде банктегі 0,00 доллар көрсетілген чектерге байланысты жұмыс істей алмай тұрып қалған программалық жабдықтамалар жайлы әңгімелер ескірген; ондай жағдайлар енді қайталанбайды да. Дегенмен, программалық жабдықтамаларды сіздер сияқты адамдар жазады ғой. Сіз қате жіберіп қоюыңыздың мүмкін екенін білесіз; олай болатын болса, неге "басқалар" қате жібермейді деп ойлаймыз.

Көбінесе біз сенімсіз бөліктерден сенімді жұмыс істейтін жүйе жасауды білеміз деп санаймыз. Біз әрбір программаны, әрбір класты және әрбір функцияны қинала отырып құрастырамыз, бірақ көбінесе олар бірінші рет іске қосылғанда жұмыс істемей қалады. Сонан соң біз оны түзетіп жөндейміз, тесттен өткіземіз, оның қателерін барынша дұрыстай отырып, программаны қайта жобалаймыз. Дегенмен, әрбір қарапайым емес жүйеде мұнан кейін де, бірнеше жасырын қателер қалып қояды. Біз олар жайлы білеміз, бірақ таба алмаймыз немесе (сирегірек) дер кезінде таппаймыз. Мұнан кейін біз программадағы күтілмеген жайттарды және "мүмкін емес" деп есептелген оқиғаларды анықтау үшін жүйені қайта жобалаймыз. Нәтижесінде сенімді жұмыс істейтін сияқты тағы бір жүйе пайда болады. Мұндай жүйеде де бұрынғыша жасырын қателер кездесуі (көбінесе солай болады да) мүмкін, оның үстіне олар біз күткеннен гөрі аз уақыт қана жұмыс істейді. Дегенмен де ол бірден тұрып қалмайды және де өзінің аздаған жұмыс функцияларын орындап тұратын болады. Мысалы, телефонға өте көп қоңыраулар түсіп жатса, ол жүйе әрбір қоңырауды дұрыс өңдемейді, бірақ толығынан тұрып та қалмайды.

Осындай күтілмеген қателерді нақты қателер деп есептеуге бола ма, жоқ па, сол жағынан да әңгіме-дүкен құруға да болар еді, бірақ олай істемейік. Жүйе жасаушылар үшін олар өз жүйелерін қалай сенімді етіп жасайтынын бірден шешіп алуы керек.

26.1.1 Сақтандыру

Тесттен өткізу – өте ауқымы кең тақырып. Тесттен қалай өткізу керек деген сауалға жауап ретінде қалыптасқан бірнеше көзқарастар бар, мысалы, әртүрлі қолданбалы пәндік аймақтарда өз дәстүрлері мен тесттен өткізу стандарттары қалыптасқан. Бұл, әрине, табиғи нәрсе: бізге бейнеойындар үшін және авиалайнерлердің борттағы компьютерлерінің программалық жабдықтамалары үшін бірдей сенімділік стандарттары қажет емес, бірақ осының нәтижесінде терминдердің шатасуы мен керекті құралдардың да бірсыпыра түрлері пайда болады. Бұл тарауды біздің дербес жобаларымызбен қатар ірі жүйелерді де тесттен өткізуге арналған идеялар көзі деп қарастыруға болады. Ірі жүйелерді тесттен өткізу кезінде бірнеше түрлі құралдар комбинациялары және ұйымдық құрылымдар да пайдаланылады, олардың барлықтарын бұл жерде сипаттап отыру қажет те, әрі мүмкін де емес.

26.2 Дәлелдемелер

Тұра тұрыңыз! Программаларымыздың дұрыс екенін дәлелдеп, тесттермен бас қатырмай-ақ қойсақ қайтеді? Эдсгер Дейкстра (Edsger Dijkstra) қысқа, әрі нұсқа етіп көрсеткендей: "Тесттен өткізу қателердің жоқ екенін емес, бар екенін көрсете алады". Бұл математиктер теоремаларын дәлелдегені сияқты программалардың да дұрыстығын дәлелдеуге ұмтылуға итермелейді.

Өкінішке орай, өте қарапайым емес программалардың дұрыстығын дәлелдеу қазіргі кездегі мүмкіндіктерден (кейбір өте шектеулі қолданбалы аймақтардан басқа) тыс жатыр және дәлелдеудің өзінде де қате болуы мүмкін (математикалық теоремаларда да солай). Оның үстіне, программалардың дұрыстығын дәлелдеу теориясы мен практикасы өте күрделі болып табылады. Сонымен, біз өз программаларымызды құрылымдай алатын болғандықтан, олар жайлы ойлана отырып, дұрыс жұмыс істейтіндігіне көз жеткізе аламыз. Бірақ біз программаларды тесттен өткіземіз (26.3 бөлім) және кодтарды басқа бір қателерге (26.4 бөлім) бой алдырмайтындай түрде ұйымдастыруға талпынамыз.

26.3 Тесттен өткізу

5.11 бөлімде біз тесттен өткізуді қателерді жүйелі түрде іздеу деп атаған болатынбыз. Осындай іздеудің тәсілдерін қарастырайық.

Олар *модульдерді тесттен өткізу* (unit testing) және *жүйелерді тесттен өткізу* (system testing) болып бөлінеді. Модуль деп толық программаның бөлігі болып саналатын функцияны немесе класты айтады. Егер біз осындай модульдерді жеке-жеке тесттен өткізетін болсақ, онда қате шықса, оны қайдан іздейтінімізді білетін боламыз; біз тапқан қателердің бәрі де тексеріліп жатқан модульде (немесе

біз тесттен өткізуге пайдаланған кодта) болады. Ал жүйе толығынан тексерілетін жүйені тесттен өткізу кезінде қате "жүйенің ішінде" деп қана білеміз. Көбінесе жүйелерді тесттен өткізу кезінде табылған қателер – егер біз жеке модульдерді жақсылап тесттен өткізсек – модульдердің өзара дұрыс әрекеттеспеуіне байланысты болады. Жүйедегі қатені табу модульдегі қатені табудан гөрі қиынырақ, әрі оған уақыт та, күш те көп жұмсалады.

Әрине, модуль (мысалы, класс) басқа модульдерден (мысалы, функциялардан немесе басқа кластардан) тұра алады және жүйелер де (мысалы, электрондық коммерциялық жүйелер) басқа жүйелерден (мысалы, мәліметтер базасынан, графикалық қолданушы интерфейсінен, желілік жүйеден және тапсырыстарды тексеру жүйесінен) тұруы мүмкін, сондықтан модульдерді тесттен өткізу мен жүйелерді тесттен өткізу айырмашылықтары, біз ойлағандай, онша айқын емес, бірақ жалпы идея дұрыс тесттен өткізу кезінде біз өз күшімізді үнемдейміз және қолданушылардың да нервісін көп қозғамаймыз деген ойға келіп тіреледі.

Тесттен өткізудің бір тәсілі күрделірек жүйелерді құру кезінде олар, өз кезегінде, өздерінен шағын модульдерден құрастырылады деген ұғымға негізделген. Сонымен, тесттен өткізуді ең кіші модульдерден бастаймыз, сонан соң осындай модульдерден тұратын ірілеу модульдерді тексереміз және т.с.с жүйені толық тесттен өткізгенше, осы тәсілді қолданамыз. Басқаша айтқанда, жүйе мұнда ең үлкен модуль ретінде (егер ол басқа бір үлкенірек жүйенің бөлігі ретінде пайдаланылмаса) қарастырылады.

Алдымен модуль (мысалы, функция, класс, кластар иерархиясы немесе шаблон) қалай тесттен өткізілетінін қарастырайық. Тесттен өткізу мөлдір жәшік (тесттен өтетін модульдің жүзеге асырылу нақтылықтарын көре аламыз) тәсілімен немесе қара жәшік (тесттен өтетін модульдің интерфейсін ғана көреміз) тәсілімен өтеді.

Біз бұл тәсілдердің айырмашылықтарын онша терең қарастырмаймыз; кез келген жағдайда да тесттен өткізілетін код оқылуы тиіс. Бірақ есте болатын нәрсе: кейінірек мұны біреу қайта жазып шығады, сондықтан интерфейсте берілмеген ақпаратты пайдалануға тырыспаңыз. Негізінде, кез келген тесттен өткізу түріндегі негізгі идея ақпарат енгізуге интерфейстің әсерін зерттеу болып табылады.

Сіз тесттен өткізгеннен кейін біреу (мүмкін сіздің өзіңіз) сол кодты өзгертетін болса, ол бізді регрессивтік тесттен өткізу идеясына алып келеді. Негізінде, егер сіз бір нәрсені өзгертсеңіз, бірден тесттен өткізуді қайталаңыз, сонда ештеңе бұзылмағанына көз жеткізетін боласыз. Сонымен, егер сіз модульді жақсартсаңыз, онда тесттен өткізуді қайталауыңыз керек, толығынан аяқталған жүйені басқаларға беру үшін де (немесе өзіңіз пайдалануыңыздың алдында) жүйені толық тесттен өткізуге тиістісіз. Жүйені осылай толық тесттен өткізуді көбінесе *регрессивтік тесттен өткізу* (regression testing) деп атайды, өйткені ол жаңадан қате пайда болмағанына сенімді болу үшін, бұрын қателерді тапқан тестті қайталап орындауды жүзеге асырады. Егер қайталағанда қате шықса, онда программа регрессивтеле орындалады да, шыққан қателерді тауып түзету керек болады.

26.3.1 Регрессивтік тесттер

Бұған дейін қателерді табуға көмектескен тесттердің ірі топтамасын жасау кез келген жүйе үшін тиімді тесттер жиынын құрастырудың негізгі тәсілі болып саналады. Байқалған кемшіліктер жайлы сізге мәлімет беріп отыратын сіздің қолданушыларыңыз (әріптестеріңіз) бар делік. Ешқашанда олардың қателер жайлы айтып жазғандарын бос қалдырмаңыз! Кез келген жағдайда олар жүйеде нақты қате бар екенін көрсетеді немесе сол қолданушылардың жүйе жайлы онша көп білмейтіндерін білдіреді. Осы туралы біліп отыру әрқашанда пайдалы болып саналады.

Көбінесе қателер жайлы есеп беруде сыртқы ақпарат өте аз болады және оны өңдеудегі алғашқы міндет осы қатені көрсететін өте шағын көлемдегі қарапайым программа құру болып табылады. Ол үшін жазылған кодтың басым бөлігін алып тастауға тура келеді: жекелеп айтсақ, біз қатеге әсер етпейтін кітапхананы пайдалануды және қолданбалы (қосымша) кодтарды алып тастауға тырысамыз. Осындай шағын программа құру қате бар жерді қусырып, жүйелік кодтағы қате тұрған орынды нақтылауға мүмкіндік береді және осындай программаны тесттік жинақтарға қосып отыру керек. Шағын көлемді программа құрып алу үшін кодты біртіндеп, қатенің өзі жоғалып кеткенше қысқартып отыру керек – осындай сәттерде ең соңғы алынып тасталған код бөлігін қайта қосу қажет. Осы әрекетті қатеге қатысы жоқ барлық фрагменттер алынып тасталғанша, жалғастырып отыру керек.

Бұрын болған қателерді анықтау кезінде құрылған жүздеген (немесе ондаған мың) тесттерді жай ғана орындау жүйелі түрде дұрыс та бола қоймас, бірақ осындай жолмен біз қолданушылар мен программа жасаушылардың тәжірибесін нақты түрде пайдалана аламыз. Регрессивтік тесттер жиыны программа құрушылардың ұжымдық жадының ең басты бөлігі болып саналады. Ірі жүйелер жасау кезінде программаны жобалау мен оларды жазу кезеңдерін бізге түсіндіріп айтып отыратын бастапқы кодты жазған мамандармен біз тұрақты байланыс жасап тұра алмаймыз. Тек осы регрессивтік тесттер ғана программа жасаушылар мен оны қолданушыларды үйлестірілген түрде алған дұрыс бағыттан ауытқымауға мүмкіндік береді.

26.3.2 Модульдік тесттер

Сонымен сөзді көбейту жеткілікті болған шығар! Нақты мысал қарастырайық: бинарлық іздеуді орындайтын программаны тесттен өткізейік. ISO стандартынан алынған оның спецификациясы төменде келтірілген (25.3.3.4 бөлімі).

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first,
                   ForwardIterator last, const T& value );
```

```
template<class ForwardIterator,class  T,class  Compare>
bool binary_search(ForwardIterator first,
     ForwardIterator last, const T& value,Compare comp );
```

Талап етеді. [first,last) диапазонының e элементтері e<value және !(value<e) немесе comp(e,value) және !comp(value,e)қатынастарына сәйкес бөлінген. Оған қоса, [first,last) диапазонының барлық e элементтері үшін e<value шартынан !(value <e) шарты шығады, ал comp(e, value) шартынан !comp(value,e) шарты шығады.
Қайтарады. егер [first,last) диапазонында i итераторы бар болатын болса және мынадай шарттарды: !(*i<value) && !(value<*i) немесе comp(*i,value) == false && comp(value,*i) == false қанағаттандыратын болса, true мәнін қайтарады.
Күрделілігі. log(last - first) + 2 салыстыруынан артық емес.

Бұлармен таныс емес адамға осындай формальді (жарайды, жартылай формальді делік) спецификацияны оқу оңай болады деп айта алмаймыз. Дегенмен, егер сіз осы тараудың басында біз ұсынған бинарлық іздеуді жобалауға және іске асыруға арналған жаттығуды нақты түрде орындасаңыз, онда бинарлық іздеу кезінде не болатынын және оны қалай тесттен өткізу керек екенін жақсы түсінуге тиіссіз. Бинарлық іздеуге арналған функцияның осы (стандартты) нұсқасы аргументтер ретінде екі бірбағытты итераторлар (20.10.1 бөлімін қ.) мен бір мәнді қабылдап алады да, егер ол осы итераторлармен көрсетілген диапазонда жататын болса, **true** мәнін қайтарады. Осы итераторлар реттелген тізбекті беруі тиіс. Салыстыру критериі (сұрыптау) болып **<** операторы саналады. Салыстыру критериі қосымша аргумент ретінде берілетін **binary_search** функциясының екінші нұсқасын біз оқырмандарға жаттығу ретінде өздері орындауы үшін қалдырамыз.

Мұнда біз тек компилятор ұстай алмайтын қателерді ғана кездестіреміз, сондықтан осыған ұқсас мысалдар кейбіреулер үшін мәселеге айналуы да мүмкін.

```
binary_search(1,4,5) ;
// қате: int – бірбағытты итератор емес
vector<int> v(10);
binary_search(v.begin(),v.end(),"7");
// қате: бүтін сандар векторында тіркесті табу мүмкін емес
binary_search(v.begin(),v.end());  // қате: мән ұмытылған
```

binary_search() функциясын *жүйелі түрде* қалай тесттен өткізуге болады? Әрине, біз барлық аргументтерді біртіндеп қарастырып шыға алмаймыз, өйткені мұндай аргументтер ретінде мүмкін болатын кез келген типтегі кез келген мәндер тізбегі бола алады – мұндай тесттер саны шексіздікке ұмтылады! Сонымен,

біз тесттерді таңдап алуымыз керек және де оларды таңдаудың кейбір қағидаларын да анықтауымыз қажет.

- *Мүмкін болатын қателерге арналған т*ест (көптеген қателерді табады).
- *Қауіпті қателерге арналған т*ест (мүмкін болатын ең жаман нәтижелер беретін қателерді табады).

Қауіпті деп біз ең жаман нәтижелерге кезіктіретін қателерді айтып отырмыз. Жалпы бұл түсінік анықталмаған сипатта тұр, бірақ нақты программалар үшін оны айқындауға да болады. Мысалы, егер бинарлық іздеуді басқа есептерден бөліп қарастыратын болсақ, онда барлық қателерді де бірдей қауіптілерге жатқызуға болар еді. Бірақ егер біз барлық жауаптары екі реттен тексерілетін программада `binary_search` функциясын пайдаланатын болсақ, онда `binary_search` функциясынан қате жауап алу ешқандай да жауап алмағаннан гөрі күтілген нұсқа болуы мүмкін, өйткені соңғы жағдайда шексіз цикл туындауы да ықтимал. Мұндай жағдайда `binary_search` функциясының дұрыс жауап бермейтін нұсқаларын зерттеуді салыстыруға қарағанда, бұл функциядан шексіз (немесе өте ұзақ) цикл алатын қулық тәсілді табу үшін көбірек күш жұмсауға тура келеді. Осы сөйлемдегі "қулық" деген сөзді ойға алыңыз. Басқасынан бөлек, тесттен өткізу –"кодты дұрыс жұмыс істемеуге қалай мәжбүр етеміз" деген есепке шығармашылық көзқарасты талап ететін әрекет болып табылады.

Ең жақсы тесттен өткізушілер тек әдіскер ғана жандар емес, олар қиынды қиыстыра білетін де адамдар (әрине, сөздің жақсы мағынасында).

26.3.2.1. Тесттен өткізу стратегиясы

`binary_search` функциясын зерттеуді біз неден бастаймыз? Біз оның талаптарына, яғни оның кіріс мәліметтерінің мүмкін мәндеріне қараймыз. Тесттен өткізушілердің өкінішіне орай, талаптарда тікелей `[first,last)` диапазоны реттелген тізбек болуы керек деп көрсетілген. Басқаша айтқанда, шақыратын модуль осыған кепілдік беруі тиіс, сондықтан біз `binary_search` функциясына реттелмеген тізбекті немесе `last<first` шарты орындалатын `[first,last)` диапазонын бере алмаймыз. `binary_search` функциясының талаптарында егер біз осы шартты бұзатын болсақ, ол не істеуі керек екендігі көрсетілмегендігіне назар аударыңыз.

Стандарттың кез келген басқа фрагменттерінде мұндай кезде функция аластама туындатуы мүмкін деп айтылған, бірақ ол мұны жасауға міндетті емес. Дегенмен `binary_search` функциясын тесттен өткізу кезінде мұндай нәрселерді қатаң есте сақтау керек, өйткені, егер шақыратын модуль `binary_search` функциясы сияқты функцияның талабын бұзатын болса, онда, әрине, қателер туындайды.

`binary_search` функциясы үшін келесідей қателер болуы мүмкін:

- Функция ешнәрсе қайтармайды (мысалы, шексіз циклге байланысты).
- Ақау шықты (мысалы, атаусыз ету дұрыс орындалмаған, шексіз рекурсия).
- Көрсетілген тізбекте болуына қарамастан, керекті мән табылмады.

Бұдан басқа, қолданушы қателері үшін келесідей мүмкіндіктердің бар екені де есте болуы керек:

- Тізбек реттелмеген (мысалы, `{2,1,5,-7,2,10}`).
- Тізбек дұрыс емес (мысалы, `binary_search(&a[100], &a[50],77)`).

`binary_search(pi,p2,v)` функциясын жай шақыру кезінде функцияны іске асыратын программалаушы қандай қате (тест өткізуші тұрғысынан қарағанда) жіберуі мүмкін? Қателер көбінесе ерекше жағдайларда туындайды. Жекелеп айтсақ, тізбектерді талдауда (кез келген түрін) біз әрқашанда оны басы мен соңын іздейміз. Одан басқа, тізбектің бос емес екендігін әрқашанда тексеріп отыру керек. Есептің талабы бойынша реттелген бүтін сандардың бірнеше жиымдарын қарастырайық:

```
{ 1,2,3,5,8,13,21 }            // "кәдімгі тізбек"
{ }                            // бос тізбек
{ 1 }                          // тек бір элемент
{ 1,2,3,4 }                    // элементтердің жұп саны
{ 1,2,3,4,5 }                  // элементтердің тақ саны
{ 1, 1, 1, 1, 1, 1, 1 }        // барлық элементтер бірдей
{ 0,1,1,1,1,1,1,1,1,1,1,1,1 } //соңғы элементтері бірдей
{0,0,0,0,0,0,0,0,0,0,0,0,0,1} //соңғы элементтері әртүрлі
```

Кейбір тест тізбектерін программа арқылы жасаған дұрыс.

- ```
  vector<int> v1;     // өте ұзын тізбек
  for (int i=0; i<100000000; ++i) v.push_back(i);
  ```
- Элементтерінің саны кездейсоқ сан түріндегі тізбектер.
- Элементтері кездейсоқ сан түріндегі тізбектер (бұрынғыша реттелген).

Дегенмен, бұл тест біз қалағандай, жүйелі түрде емес. Ойға алғандай, біз бірнеше тізбектерді қарастырдық. Біз мұнда көптеген мәндермен жұмыс істегенде пайдалы болып саналатын кейбір ережелерге сүйендік; соларды тізіп көрсетейік:

- Бос жиын.
- Шағын жиындар.
- Үлкен жиындар.
  ```

- Экстремальді таралымдағы жиындар.
- Соңында қызғылықты бір нәрсе болатын жиындар.
- Дубликаттары бар жиындар.
- Элементтерінің саны жұп және тақ болып келген жиындар.
- Кездейсоқ сандар арқылы пайда болған жиындар.

Біз күтпеген бір қатені табатын шығар деген оймен кездейсоқ тізбектерді пайдаланамыз. Бұл тәсіл тым "жалаң" сипатта болады, бірақ уақыт тұрғысынан қарағанда, ол өте тиімді.

Біз неге жұп және тақ сандарды қарастырамыз? Әңгіме мынада: көптеген алгоритмдер кіріс тізбектерді бөліктерге, мысалы екі бөлікке бөліп тастайды, ал программалаушы элементтерінің саны тек тақ болып немесе тек жұп болып келгендерін есепке алуы мүмкін. Негізінде, егер тізбек бөліктерге бөлінсе, онда бөлу нүктесі бір ішкі тізбектің соңы болып табылады, ал қателердің көбісі тізбек соңында туындайтыны белгілі.

Жалпы біз келесі шарттарды іздейміз.

- Экстремальді жағдайлар (үлкен немесе кіші тізбектер, кіріс мәліметтерінің әртүрлі таралымдары және т.б.).
- Шекаралық шарттар (шекара маңында болатын жайттардың барлығы).

Бұл түсініктердің шынайы мағыналары тесттен өткізілетін нақты программаға байланысты болады.

26.3.2.2 Қарапайым тест сұлбасы

Тесттердің екі түрлі санаттары болады: табысты түрде өтуі тиіс тесттер (мысалы, тізбекте бар мәнді іздеу) және қате шығып тұрып қалуы тиіс тесттер (мысалы, бос тізбектен мән іздеу). Жоғарыда келтірілген әрбір тізбектер үшін бірнеше табысты және қате беретін тесттер жасайық. Ең қарапайым және оңай орындалатын тесттен бастайық та, сонан кейін оны біртіндеп анықтап, **binary_search** функциясы үшін қабылдауға болатын деңгейге дейін алып барайық:

```
int a[] = { 1,2,3,5,8,13,21 };
if (binary_search(a,a+sizeof (a)/sizeof (*a) ,1) == false) cout << "қате";
if (binary_search(a,a+sizeof (a)/sizeof (*a) ,5) == false) cout << "қате";
if (binary_search(a,a+sizeof (a)/sizeof (*a) ,8) == false) cout << "қате";
if (binary_search(a,a+sizeof (a)/sizeof (*a) ,21) == false) cout << "қате";
if (binary_search(a,a+sizeof (a)/sizeof (*a) ,-7) == true) cout << "қате";
if (binary_search(a,a+sizeof (a)/sizeof (*a) ,4) == true) cout << "қате";
if (binary_search(a,a+sizeof (a)/sizeof (*a) ,22) == true) cout << "қате";
```

Бұл көңілсіз және адамды шаршатады, бірақ бұл тек басы ғана. Негізінде көптеген қарапайым тесттер – бұл ұқсас шақырулардың көлемді (ұзын) тізімдері ғана. Бұл оңай тәсілдің жақсы жағы оның өте қарапайымдылығы. Тіпті, тест өткізушілер қатарындағы жаңа ғана үйреніп жүрген адам да бұған өз үлесін қоса алады. Дегенмен әдетте біз мұны жақсырақ етіп жасаймыз. Мысалы, егер жоғарыдағы кодтың бір жерінен қате шықса, біз оның қай жерден шыққанын біле алмаймыз. Оны анықтау мүмкін емес. Сондықтан бұл код фрагментін басқаша етіп жазамыз:

```
int a[] = { 1,2,3,5,8,13,21 };
if(binary_search(a,a+sizeof(a)/sizeof(*a),1)==false) cout<<"1қате";
if(binary_search(a,a+sizeof(a)/sizeof(*a),5)== false) cout<<"2қате";
if(binary_search(a,a+sizeof(a)/sizeof(*a),8)==false) cout<<"3қате";
if(binary_search(a,a+sizeof(a)/sizeof(*a),21)==false) cout<<"4қате";
if(binary_search(a,a+sizeof(a)/sizeof(*a),-7)==true) cout<<"5қате";
if (binary_search(a,a+sizeof(a)/sizeof(*a),4) == true) cout <<"6қате";
if(binary_search(a,a+sizeof(a)/sizeof(*a),22)==true) cout<<"7қате";
```

Егер сіз осындай оншақты тест жасасаңыз, сонда айырмашылығын жақсы түсінесіз. Шынайы жүйелерді тесттен өткізгенде, біз көбінесе мыңдаған тесттерді тексеруіміз керек, сондықтан қатенің қай жерден шыққанын білу өте маңызды болып табылады.

Ары қарай жылжу үшін тесттен өткізу әдістемесінің тағы бір мысалын (жартылай жасанды) атап өтейік: біз дұрыс мәндерді тесттен өткізген болатынбыз, кейде оларды тізбектің соңынан, ал кейде ортасынан алғанбыз. Берілген тізбек үшін біз барлық мәндерді де қарастырып шығамыз, бірақ практикада оны жүзеге асыру мүмкін емес. Қате шығаруға бағытталған тесттер үшін тізбектің ортасынан бір мән және соңынан бір мән алайық. Бұл тәсіл тізбектермен немесе мәндер диапазондарымен жұмыс істегенде қолдануға болатын кең таралған үлгі болғанымен, оның жүйелі мысал емес екенін тағы да айтып өтейік.

Көрсетілген тесттердің қандай кемшіліктері бар?

- Бір кодты бірнеше рет қайталап жазу керек.
- Тесттер қолмен нөмірленген.
- Шығарылатын мәлімет өте қысқа (аз мәлімет береді).

Ойлана келіп, біз тесттерді файлға жазбақшы болып шештік. Әрбір тесттің айқындау белгісі, ізделетін мәні, берілген тізбегі және күтілетін нәтижесі болуы тиіс. Мысалы:

```
{ 27 7 { 1 2 3 5 8 13 21} 0 }
```

Бұл тест нөмірі – `27`. Ол `7` санын мынадай `{ 1,2,3,5,8,13,21 }` тізбектен іздейді де, нәтижесін `0` болады деп күтеді (яғни `false`). Неге біз бұлт тестті

файлға жаздық, неге программа мәтініне қоспадық? Мұндағы жағдайда біз бұлт тестті бірден бастапқы кодқа жаза алатын едік, бірақ программа мәтініндегі мәліметтердің үлкен көлемі оны күрделендіріп жібереді. Оның үстіне, тесттер көбінесе басқа программалар арқылы жасалады. Сондықтан программалар арқылы жасалған тесттер бірден файлдарға жазылады. Оған қоса, енді біз әртүрлі тесттік файлдармен жұмыс істейтін тесттік программалар жаза аламыз:

```
struct Test {
      string label;
      int val;
      vector<int> seq;
      bool res;
};

istream& operator>>(istream& is,Test& t);
// сипатталған формат қолданылады

int test_all()
{
      int error_count = 0;
      Test t;
      while (cin>>t) {
          bool r=binary_search(t.seq.begin(),
                                        t.seq.end(),t.val);
            if (r !=t.res) {
               cout << "қате: тест " << t.label
                    << " binary_search: "
                    << t.seq.size() << " элементтері,val=="
                    <<   t.val << " -> " << t.res << '\n';
               ++error_count;
            }
      }
      return error_count;
}

int main()
{
          int errors = test_all();
          cout << "қателер саны: " << errors << "\n";
}
```

Кейбір тест мәліметтері мынадай түрде болады:

```
{ 1.1 1 { 1,2,3,5,8,13,21 } 1 }
{ 1.2 5 { 1,2,3,5,8,13,21 } 1 }
{ 1.3 8 { 1,2,3,5,8,13,21 } 1 }
{ 1.4 21 { 1,2,3,5,8,13,21 } 1 }
{ 1.5 -7 { 1,2,3,5,8,13,21 } 0 }
{ 1.6 4 { 1,2,3,5,8,13,21 } 0 }
{ 1.7 22 { 1,2,3,5,8,13,21 } 0 }

{ 2 1 { } 0 }

{ 3.1 1 { 1 } 1 }
{ 3.2 0 { 1 } 0 }
{ 3.3 2 { 1 } 0 }
```

Мұнан біздің неге сан емес, тіркестік белгі пайдаланғанымыз көрініп тұр: бұл бір тізбек үшін әртүрлі тесттерді қолдануды белгілейтін ондық нүкте арқылы тесттерді икемдірек нөмірлеуге мүмкіндік береді. Тесттердің күрделірек форматы мәліметтер файлында бір тесттік тізбекті бірнеше рет қайталамайтындай етеді.

26.3.2.3 Кездейсоқ тізбектер

Тесттен өткізуге мәндер таңдай отырып, біз функция құрған мамандарды (көбінесе біз өзіміз боламыз) алдап өтуге тырысып, қателерді жасырып тұрған кодтың осал жерлерін анықтайтын мәндерді пайдалануға талпынамыз (мысалы, шарттардың күрделі тізбектері, тізбектердің соңғы жақтары, циклдер және т.с.с.). Дегенмен, біз өз кодымызды жазып, оны дұрыстағанда да солай істейміз. Сонымен, тестті жобалау барысында біз программа құру кезінде жіберген логикалық қатемізді қайталап, мәселені толығынан тексермей жіберуіміз де мүмкін. Осының себебінен тестті программа авторы емес, басқа біреу жобалап жасағаны дұрыс болып саналады.

Кейде осы мәселені шешетін бір тәсіл бар: ол көптеген кездейсоқ мәндер беру (туындату, генерациялау) керек. Мысалы, төменде тесттің сипатталуын **cout** ағымына 26.7 бөлімнен алынған **rand_int()** функциясы арқылы жазатын мысал көрсетілген.

```
void make_test(const string& lab,int n,int base,int spread)
// lab белгісі бар тесттің сипатталуын cout ағымына жазады
// base позициясынан бастап, n элементтер тұратын тізбекті
// spread – элементтер арасындағы орташа қашықтық туындатады
{
   cout << "{" << lab << " " << n << "{";
   vector<int> v;
   int elem = base;
```

```
    for (int i = 0; i<n; ++i) {       // элементтер құрамыз
          elem+= rand_int(spread);
          v.push_back(elem);
    }
    int val=base+ rand_int(elem-base);
    // ізделетін мәнді құрамыз
    bool found = false;
    for (int i = 0; i<n; ++i) {
         // элементтерді баспаға шығарамыз
         // және val элементі табылды ма, соны тексереміз
         if (v[i]==val) found = true;
         cout << v[i] << " ";
    }
    cout << "} " << found << " }\n";
}
```

Айта кететін нәрсе, кездейсоқ тізбектен **val** элементі табылды ма, жоқ па, соны тексеру үшін біз **binary_search** функциясын пайдаланбадық. Тесттің дұрыстығын қамтамасыз ету үшін біз тексеріліп жатқан функцияны пайдаланбауымыз керек.

Негізінде, **binary_search** функциясы кездейсоқ сандар арқылы осындай қарапайым тәсілмен тесттен өткізуге онша қолайлы мысал емес. Біз "қолдан" жасалған тесттер көмегімен алдыңғы кезеңдерде жіберілген жаңа қате таба аламыз ба деп күмәнданамыз, дегенмен, бұл әдіс жиі-жиі пайдалы болып та жатады. Сонда да бірнеше кездейсоқ тесттер жасап көрейік:

```
int no_of_tests = rand_int(100);
// 50 шақты тест жасаймыз
for (int i = 0; i<no_of_tests; ++i) {
   string lab = "rand_test_";
   make_test(lab+to_string(i),
   // 23.2 бөлімінен to_string to__string из раздела  23.2
   rand_int(500),                    // элементтер саны
   0,                                // base
   rand_int(50));                    // spread
}
```

Кездейсоқ сандарға негізделіп жасалған тесттер, нәтижелері алдыңғы кезең операцияларының қалай өңделгеніне, яғни жүйенің қалып-күйіне тәуелді болып келетін көптеген операциялардың кумулятивтік әсерлерін тесттен өткізу қажет болғанда, өте пайдалы болып табылады (5.2 бөлімін қ.).

binary_search функциясын тесттен өткізу үшін кездейсоқ сандардың алыну себебі – тізбектегі мәнді кез келген іздеу нәтижесі осы тізбектегі басқа іздеу нәтижелеріне байланысты болмайды.

Бұл, әрине, **binary_search** функциясының мағынасыз құрылған қателері өте көп кодтан тұрмайтынын есепке алады, мысалы, ол тізбекті өзгертіп жібермейді. Бізде мұндай жағдайға арналған жақсырақ тест бар (5-жаттығу).

26.3.3 Алгоритмдер және алгоритм еместер

Мысал ретінде біз **binary_search()** функциясын қарастырдық. Бұл алгоритмнің қасиеттері төменде келтірілген.

- Кіріс мәліметтерге қоятын дәл анықталған талаптары бар.
- Оның кіріс мәліметтермен не істеуге болатыны және не істеуге болмайтыны жайлы дәл анықталған нұсқаулары бар (мұнда ол бұл мәліметтерді өзгертпейді).
- Оның кіріс мәліметтеріне тікелей жатпайтын объектілермен байланысы жоқ.
- Оның айналасына ешқандай салмақты шектеулер қойылмаған (мысалы, оның шектелген уақыты, жады көлемі немесе қарамағындағы ресурстар көлемі көрсетілмеген).

Бинарлық іздеу алгоритмінің айқындалған және ашық тұжырымдалған алғы және соңғы шарттары бар (5.10 бөлімін қ.). Басқаша айтқанда, бұл алгоритм – тест өткізушінің арманы. Біздің бұлай жақсы жолымыз бола бермейді, көбінесе шала құрастырылған ағылшын тіліндегі түсініктемелермен және бір-екі диаграммамен ғана сүйемелденетін нашар жазылған кодтарды (ең аз дегенде) тесттен өткізуге тура келеді.

Тоқтай тұрыңыз! Біз адасып басқа жолға түсіп кеткен жоқпыз ба? Егер бізде кодтың не істеуі керектігі жайлы толық сипаттама жоқ болса, оның дұрыстығы мен тесттен өткізілуі жайлы қалай айтуға болады? Мәселе мынада, программалық жабдықтама орындауға тиіс әрекеттердің басым бөлігін дәлме-дәл математикалық терминдер арқылы өрнектеудің өзі оңай емес. Оған қоса, көптеген жағдайларда, бұл әрекет теориялық тұрғыдан мүмкін болғанмен, программалаушының математикалық білімінің көлемі ондай кодты жазып, тесттен өткізуге жетіспей жатады. Сондықтан біз нақты әрі дәл спецификациялар жайлы идеалды көзқарастардан алшақ болып, бізге байланысты емес шарттары бар шынайы нақтылыққа бой ұсынуымыз керек.

Ал енді біз тесттен өткізуіміз керек болатын нашар бір функцияны қарастырайық. Нашар функция деп біз мынаны түсінеміз.

- *Кіріс мәліметтер. Кіріс мәліметтер*ге қойылатын талаптар (тікелей немесе жанамалы), біз ойлағандай, айқын түрде тұжырымдалмаған.
- *Шығыс мәліметтер.* Нәтижелер де (тікелей немесе жанамалы), біз ойлағандай, айқын түрде тұжырымдалмаған.

- *Ресурстар.* Ресурстарды пайдалану шарттары (уақыт, жады, файлдар және т.б.), біз ойлағандай, айқын түрде тұжырымдалмаған.

Біз тікелей немесе жанамалы (яғни, тікелей емес) түрде дегенде, тек қана формальді параметрлер мен қайтарылатын мәнді ғана тексеріп қоймай, оған қоса, ауқымды айнымалылардың әсерін, енгізу-шығару ағымдарын, файлдарды, бос жады аймағын бөлуді және т.с.с. ойға аламыз. Мұнда біз не істей аламыз? Біріншіден, мұндай функция, практика жүзінде, әрқашанда өте көлемді (ұзын) болады, әйтпесе оның талаптары мен әрекеттерін дәлірек сипаттауға болар еді. Бұл жердегі әңгіме – ұзындығы бес беттей болатын функция жайлы немесе қосымша функцияны күрделірек жолмен пайдаланатын басқа функция туралы болуы мүмкін. Функция үшін бес бет көлем – бұл, әрине, көбірек. Дегенмен, біз одан да әлдеқайда көлемді функцияларды да кездестіргенбіз. Өкінішке орай, бұл жиі кездеседі.

Егер сіз өз кодыңызды тексеріп жатсаңыз және сіздің уақытыңыз болса, онда алдымен нашар функцияны одан кіші бірнеше жай функцияларға бөліп жіберуге тырысыңыз, сонда олардың әрқайсысы нақты спецификациясы бар ойдағы (идеал) функцияға жақын болады және, бірінші кезекте, оларды тесттен өткізіңіз. Дегенмен, айтайын деген ойымыз, қазіргі сәттегі біздің мақсат – программалық жабдықтаманы тесттен өткізу, яғни жай ғана табылған ақаулықтарды түзету емес, мүмкін болатын барлық қателерді жүйелі түрде іздеу болып табылады.

Сонымен, біз нені іздейміз? Тесттен өткізушілер ретінде біздің міндет — қателерді іздеу. Олар көбінесе қайда жасырынып тұрады? Қатесі бар программалардың басқалардан қандай айырмашылықтары болады?

- Жай көзге байқалмай тұратын кодтардың бір-бірінен тәуелділігі. Ауқымды айнымалыларды қараңыз, константалық сілтеме, нұсқауыш, т.с.с. арқылы берілетін аргументтерді пайдалануды іздеңіз.
- Ресурстарды басқару. Компьютер жадын басқаруға (`new` және `delete` операторлары), файлдарды пайдалануға, бұғаттауларға және т.с.с. назар аударыңыз.
- Циклдерді іздеңіз. Солардан шығу шарттарын тексеріңіз (`binary_search()` функциясындағы сияқты).
- `if` және `switch` нұсқаулары (тармақталу нұсқаулары деп аталады). Солардың логикасындағы қатені іздеңіз.

Осы көрсетілген пункттердің әрқайсысын бейнелейтін мысалдарды қарастырайық.

26.3.3.1 Тәуелділіктер

Келесі мағынасы жоқ функцияны қарастырайық.

```
int do_dependent(int a,int& b)
// нашар функция ұйымдастырылмаған тәуелділіктер
{
   int val ;
   cin >> val;
   vec[val] += 10;
   cout << a;
   b++;
   return b;
}
```

`do_dependent()` функциясын тесттен өткізу үшін біз оның аргументтері жиынын жай ғана синтездеп, функцияның олармен не істейтінін бақылап қана қоймаймыз. Біз бұл функцияның `cin`, `cout` және `vec` сияқты ауқымды айнымалыларды пайдаланатындығын есепке алуымыз керек. Бұл жағдай мұндай шағын және мағынасыз программада үйлесімді болғанмен, бірақ көлемді программаларды жасырын тұруы мүмкін. Қуанышымызға орай, осындай тәуелділіктерді табуға мүмкіндік беретін программалық жабдықтама бар. Бірақ ол әрқашанда қолжетімді бола бермейді және өте сирек қолданылады. Мысалы, бізде кодты талдайтын программалық жабдықтама жоқ болсын делік, мұндайда біз функциядағы тәуелділіктерді іздестіре отырып, оның әрбір жолын тізбекті түрде біртіндеп толық қарастырып шығуымыз керек.

`do_dependent()` функциясын тесттен өткізу үшін, біз оның бірқатар қасиеттерін талдап шығуымыз керек.

- Функцияның кіріс мәліметтері
  - `a` айнымалысының мәні.
  - `b` айнымалысының мәні және ол сілтеме жасап тұрған `int` типті айнымалының мәні.
  - `cin` ағымынан (`val` айнымалысына) мәндер енгізу және `cin` ағымының қалып-күйі.
  - `cout` ағымының қалып-күйі.
  - `vec` айнымалысының мәні, оның ішіндегі `vec [val]` мәні.
- Функцияның шығыс мәліметтері
  - Қайтарылатын мән.
  - `b` айнымалысы (біз оны инкременттегенбіз) сілтеме жасап тұрған `int` типті айнымалының мәні.
  - `cin` объектісінің қалып-күйі (ағымның және форматтың қалып-күйін тексеріңіз).

- `cout` объектісінің қалып-күйі (ағымның және форматтың қалып-күйін тексеріңіз).
- `vec` жиымының қалып-күйі (біз `vec [val]` элементіне мән меншіктегенбіз).
- `vec` жиымы (`vec [val]` ұяшығы мүмкін болатын диапазоннан тыс жатуы мүмкін) туындата алатын кез келген аластамалар.

Бұл ұзақ тізім. Негізінде, ол функциясының өзінен де көлемдірек. Ол біздің ауқымды айнымалыны қадағалайтынымызды және константалық емес сілтемелер (және нұсқауыштар) туралы да ойланатынымызды білдіреді. Дегенмен жай ғана өз аргументтерін оқып, қайтарылатын мәнді ғана шығаратын функциялардың өз артықшылықтары бар: оларды оңай түсінуге және тесттен өткізуге де болады.

Біз кіріс және шығыс мәліметтерді айқындап (идентификациялап) алғаннан кейін, бірден `binary_search()` функциясын тесттен өткізу кезіндегі жағдайға қайтадан тап боламыз. Әрине, мұнда біз қажетті нәтижелерге қол жеткізуге (тікелей немесе жанамалы түрде) болатынын білу үшін кіріс мәндері бар тесттер (тікелей немесе жанамалы түрде енгізу үшін) жасаймыз (генерациялаймыз). `do_dependent()` функциясын тесттен өткізе отырып, алда не болатынын білу үшін біз `val` айнымалысының ең үлкен мәнін және теріс мәнін енгізуден бастауымыз керек. Бірақ `vec` жиымы диапазонды тексеруді (әйтпесе біз қауіпті қателер шығарып алуымыз мүмкін) қарастыратын вектор болатын болса, жақсы болар еді. Әрине, біздің бұл туралы құжаттамада не жазылғанын қарап шығуымызға болар еді, бірақ мұндай жағымсыз функциялар тиянақты әрі толық спецификациялармен өте сирек сүйемелденеді, сондықтан біз бұл функцияны жай ғана "бұзуға" (яғни қателерін табуға) тырысамыз да, оның дұрыстығы жайлы сұрақтар қоя бастаймыз. Көбінесе мұндай тесттен өткізу мен сұрақтар қоюдың араласуы функцияны қайта құруға алып келеді.

26.3.3.2 Ресурстарды басқару

Мағынасы жоқ функция қарастырайық.

```
void do_resources1(int a,int b,const char* s)
// нашар функция ресурстарды тиянақсыз пайдалану
{
   FILE* f = fopen(s,"r");   // файл ашамыз (C тілі стилінде)
   int* p = new int[a];        // жады бөлеміз
   if (b<=0) throw Bad_arg(); // аластама туындата алады
   int* q = new int[b];       // тағы да аздап жады бөлеміз
   delete[] p;
   // p нұсқауышы сілтеме жасап тұрған жады аймағын босату
}
```

`do_resources1()` функциясын тесттен өткізу үшін, біз ресурстардың дұрыс берілгенін, яғни бөлінген ресурс босатылды ма немесе басқа функцияға берілді ме, соны тексеріп шығуымыз керек.

Мүмкін болатын кемшіліктерін атап көрсетейік.

- `s` файлы жабылмаған.
- Егер `b<=0` немесе екінші `new` операторы аластама туындататын болса, `p` нұсқауышына бөлінген жады босатылмайды.
- Егер `0<b` болса, `q` нұсқауышына бөлінген жады босатылмайды.

Бұған қоса, біз әрқашанда файлды ашу әрекеті орындалмай қалатын мүмкіндікті қарастырып отыруымыз керек. Мұндай жағымсыз нәтижені алу үшін, біз әдейі программалаудың ескірген стилін пайдаландық (`fopen()` функциясы – бұл файл ашудың C тіліндегі стандартты тәсілі). Егерде біз жай ғана төменде келтірілген кодты жазатын болсақ, онда тесттен өткізушілердің жұмысын жеңілдеткен болар едік:

```
void do_resources2(int a,int b,const char* s) // нашарлау код
{
   ifstream is(s);              // файл ашамыз
   vector<int>v1(a);      // вектор құрамыз (жады бөлеміз)
   if (b<=0) throw Bad_arg(); // аластама туындата алады
   vector<int>v2(b);  // басқа вектор құрамыз (жады бөлеміз)
}
```

Енді әрбір ресурс объектіге тиісті болады және оның деструкторымен босатылады. Кейде тесттен өткізу идеясын жасап алу үшін функцияны қарапайым және түсінікті етуге тырысқан пайдалы болып саналады. Ресурстарды басқару есептерін шешудің жалпы стратегиясын 19.5.2 бөлімінде сипатталған RAII (Resource Acquisition Is Initialization — ресурс алу дегеніміз – инициалдау) тәсілі қамтамасыз етеді.

Ресурстарды басқару ісінің әрбір бөлінген жады аймағының босатылғанын жай ғана тексерумен шектелмейтінін айта кетейік. Кейде біз ресурстарды сырттан аламыз (мысалы, аргумент ретінде), ал кейде өзіміз оны басқа бір функцияға (қайтарылатын мән түрінде) беріп те жатамыз. Мұндай жағдайларда ресурстардың дұрыс бөлініп жатқанын түсіну де қиын. Мысал қарастырайық.

```
FILE* do_resources3(int a,int* p,const char* s)
// нашар функция ресурстың дұрыс берілмеуі
{
     FILE* f = fopen(s,"r");
     delete p;
     delete var;
```

```
        var = new int[27];
        return f;
    }
```

`do_resources3()` функциясының ашық файлды қайтадан қайтарылатын мән ретінде беруі (болжамды түрде) дұрыс па? `do_resources3()` функциясының оған p аргументі ретінде берілген жады аймағын босатуы дұрыс па? Біз және де var ауқымды айнымалысын (әрине, нұсқауыш) пайдаланудың бір қитұрқы нұсқасын қостық. Негізінде, функцияға ресурстарды беру және одан алу кең таралған және пайдалы практикаға жатады, бірақ бұл операцияның дұрыс орындалып жатқанын түсіну үшін ресурстарды басқару стратегиясын білу қажет. Ресурстар кімге тиесілі? Кім оны өшіруі/босатуы керек? Құжаттама бұларға айқын әрі анық жауап беруі тиіс. (Армандаңыз). Кез келген жағдайда ресурстарды беру қателер шығуы мүмкін болатын сәтті көрсетеді де, тесттен өткізуді де қиындатады.

Біз ауқымды айнымалыны пайдаланып, ресурстарды басқару мысалын (әдейі) қиындатқанымызға назар аударыңыз. Егер программада бірнеше қате көздері араласып кетсе, жағдай бірден қатты қиындап кетуі мүмкін. Программалаушылар ретінде біз мұндай жағдайларды болдырмауға тырысамыз. Тесттен өткізушілер ретінде – оларды табуға талпынамыз.

26.3.3.3 Циклдер

Біз `binary_search()` функциясын талқылау кезінде циклдерді қарастырған болатынбыз. Қателердің басым бөлігі цикл соңында пайда болады.

- Айнымалылар цикл басында дұрыс инициалданған ба?
- Цикл дұрыс аяқтала ма (көбінесе соңғы элементінде)?

Ішінде қатесі бар мысал келтірейік.

```
int do_loop(vector<int>& v)   // нашар функция
                              // дұрыс емес цикл
{
   int i;
   int sum;
   while(i<=vec.size()) sum+=v[i];
   return sum;
}
```

Мұнда оңай анықталатын үш қате бар. (Олар қандай қателер?) Оған қоса, жақсы тесттен өткізуші `sum` айнымалысына сан қосылғанда, толып кету әрекеті болатынын бірден аңғарады.

- Көптеген циклдер мәліметтермен байланысты болады да, үлкен санды енгізгенде, оларда толып кету әрекеті орын алады.

Циклдерге байланысты пайда болып, буфердің толып кетуінен шығатын кең таралған және өте қауіпті қате бар, ол циклдер туралы тұрақты түрде екі түйінді сұрақ беріп анықтауға болатын қателер санатына жатады.

```
char buf[MAX];          // тұрақты көлемдегі буфер
char* read_line()       // қауіпті функция
{
   int i = 0;
   char ch;
   while(cin.get(ch) && ch!='\n') buf[i++] = ch;
   buf[i+1] = 0;
   return buf;
}
```

Әрине, сіз мұндай нәрсені жазбас едіңіз! (Ал неге жазбасқа? **read_line()** функциясының несі жаман?) Бірақ бұл қате, өкінішке орай, өте кең таралған және оның әртүрлі нұсқалары да бар.

```
// қауіпті фрагмент:
gets(buf);              // жолды buf айнымалысына оқимыз
scanf("%s",buf);        // жолды buf айнымалысына оқимыз
```

gets() және **scanf()** функцияларының сипаттамаларын өз құжаттамаларыңыздан оқып алып, олардан обадан қашқандай, алшақ жүріңіз. "Қауіпті" сөзінен біз буфердің толып кетуі компьютерлерді бұзудың құралы болып табылатынын түсінеміз. Қазірге кездегі программалар **gets()** функциясын және соның аналогтарын пайдаланудың қауіпті екені жайлы ескертеді.

26.3.3.4 Тармақталу

Әрине, таңдау жасай отырып, біз дұрыс шешім қабылдамауымыз да мүмкін. Осыған орай **if** және **switch** нұсқаулары тесттен өткізушілердің назар салатын негізгі мақсаттарының бірі болып табылады. Мұнда зерттелуге тиіс екі мәселе бар:

- Барлық мүмкін нұсқалар қарастырылған ба?
- Таңдаудың дұрыс нұсқаларымен дұрыс орындалатын әрекеттер байланысқан ба?

Келесі мағынасы жоқ функцияны қарастырайық:

```
void do_branch1(int x, int y)
// нашар функция if нұсқауын дұрыс қолданбау
{
   if (x<0) {
         if (y<0)
             cout << "үлкен теріс сан\n";
         else
             cout << "теріс сан\n";
   }
   else if (x>0) {
         if (y<0)
             cout << "үлкен оң сан\n";
         else
             cout << "оң сан\n";
   }
}
```

Бұл фрагменттегі ең мүмкін болып отырған қате – біз **x** айнымалысының нөлге тең болатын нұсқасы жайлы ұмытып кеттік. Сандарды (оң немесе теріс) нөлмен салыстыра отырып, программалаушылар ол туралы көбінесе ұмытып кетеді немесе басқа бір тармағын көрсетіп жібереді (мысалы, оны теріс сандарға жатқызады). Оған қоса, бұл фрагментте жасырынып тұрған бұдан гөрі жұқалау (бірақ кең тараған) қате бар: мынадай шарттары **(x>0 && y<0)** және **(x>0 && y>=0)** бар әрекеттердің орындары ауысып кеткен. Программалаушылар көбінесе "көшіріп алып кірістіріп қою" командаларын қолданған кездері осындай қателер пайда болады.

if нұсқауларын қолдану шарттары күрделенген сайын, олардан шығатын қателер де көбейе түседі. Тесттен өткізушілер осындай кодтарды талдай отырып, олардың бір де бір тармағын қалыс қалдырмауға тырысады. **do_branch1()** функциясы үшін тесттер жиынын төмендегідей етіп құруға болады:

```
do_branch1(-1,-1);
do_branch1(-1, 1);
do_branch1(1,-1);
do_branch1(1,1);
do_branch1(-1,0);
do_branch1(0,-1);
do_branch1(1,0);
do_branch1(0,1);
do_branch1(0,0);
```

Негізінде, бұлай біз қолданған "барлық баламаларды теріп шығу" өте қарапайым тәсіл, мұндағы **do_branch1()** функциясы мәндерді нөлмен

< және > операторы арқылы салыстырады. **x** айнымалысының оң мәндеріндегі дұрыс орындалмаған әрекеттерді анықтау үшін, біз функцияны шақыруды қажетті нәтижелермен біріктіруіміз керек.
switch нұсқауын өңдеу де осы **if** нұсқауын өңдеуге ұқсас болып келеді:

```
void do_branch1(int x, int y)
// нашар функция switch нұсқауын дұрыс қолданбау
{
   if (y<0 && y<=3)
        switch (x) {
        case 1:
            cout << "бір\n";
            break;
        case 2:
        cout << "екі\n";
        case 3:
        cout << "үш\n";
   }
}
```

Мұнда төрт классикалық қате жіберілген.

- Біз дұрыс емес айнымалы мәнін тексеріп отырмыз (**y**, **x** емес).
- Біз **break** нұсқауын ұмытып кеттік, ол **x==2** болғанда, дұрыс нәтиже бермейді.
- Біз **default** бөлімі жайлы да (оны **if** нұсқауы қарастырады деп есептедік) ұмыттық.
- Біз **0<y** өрнегінің орнына **y<0** деп жаздық.

Тесттен өткізушілер ретінде біз әрқашанда күтілмейтін жағдайларды қарастырамыз. Қатені тек түзету жеткіліксіз болып табылады. Күтпеген кездерде ол қайтадан пайда болуы мүмкін. Біз қателерді жүйелі түрде анықтайтын тесттер жазғымыз келеді. Егер біз жай ғана кодты түзете салсақ, онда есепті дұрыс шығармауымыз мүмкін немесе жаңадан қате енгізуіміз мүмкін. Кодты талдау мақсаты тек қатені анықтау ғана емес (бұл әрқашанда пайдалы), барлық қателерді (немесе нақтысын айтқанда, қателердің басым бөлігін) таба алатындай қолайлы тесттер жиынын жасау болып саналады.

Циклдерде әрқашанда жасырын түрде тұрған **if** нұсқаулары болады, олар циклден шығу шартын тексереді. Сол себепті циклдер де тармақталу нұсқаулары болып табылады. Тармақталу нұсқаулары бар программаларды талдағанда, алғашқы қойылатын сұрақ: барлық тармақтар тексерілді ме? Таң қалатын бір нәрсе, нақты программаларда мұны тексеру кейде мүмкін болмай жатады (өйткені

нақты кодтарда функциялар басқа функцияларға ыңғайлы түрде шақырылады және кез келген тәсілмен емес). Сонан соң келесі сұрақ туындайды: Біз кодтың қай бөлігін тексеріп шықтық? Бұған жиі берілетін жауап: "Біз тармақтардың біраз бөлігін тексердік" дейміз де, басқа тармақтарды неге тексере алмағанымыз жайлы айтып кетеміз. Тесттен өткізудің идеалды жағдайында біз кодты 100% тексеруіміз керек.

26.3.4 Жүйелік тесттер

Кез келген үлкен немесе шағын жүйені тесттен өткізу үшін тәжірибе керек.

Мысалы, телефон жүйелерін басқаратын компьютерлерді тесттен өткізу ондаған мың адамдардың жұмыс трафигін көрсететін компьютерлермен толтырылған сөрелері бар арнайы жабдықталған бөлмелерде өтеді. Мұндай жүйелер миллиондаған доллар тұрады және олар өте тәжірибелі инженерлер ұжымы жұмысының нәтижесі болып табылады. Оларды жұмысқа қосқан соң негізгі телефондық коммутаторлар жиырма жыл бойы үздіксіз жұмыс істей алады деп саналады, олардың жалпы тұрып қалу уақыты жиырма минуттан аспайды (энергияның болмай қалуын, су тасқынын және жер сілкінуді қоса есептеп, барлық себептерді қосқанның өзінде). Біз оның нақтылықтарына тереңдемейміз – одан да физиканы білмейтін жай адамды Марс бетіне қонуға бағытталған ғарыштық аппараттың жүру жолын есептеуге өзгерістер енгізуге үйрету оңай болады – бірақ шағынырақ жобаларды тесттен өткізу немесе ірі жүйелерді тесттен өткізу қағидаларын түсіну үшін пайдалы болатын идеяларды баяндауға талпынып көрейік.

Алдымен тесттен өткізу мақсаты өте жиі ұшырасатын және кейінгі қаупі мол қателерді іздеу болып табылатынын еске түсірейік. Тесттердің үлкен көлемін жазып және орындап шығу оңай жұмыс емес. Осыдан барып тесттен өткізушіге тексерілетін жүйенің болмысын түсіну өте қажет екені шығады. Жүйелерді тиімді түрде тесттен өткізу үшін жеке модульдерді тексеруден гөрі сол жүйе қолданылатын аймақты білу аса маңызды болады. Мұндай жүйені жасау үшін тек қана программалау тілі мен компьютерлік ғылымдарды біліп қана қоймай, қолданбалы аймақты және де оны пайдаланатын адамдарды да білу қажет. Бұл программамен жұмыс істеуге итермелейтін бір ойтүрткі (мотвация) болуы тиіс: сіз көптеген қосымша программалармен танысып, қызықты мамандармен кездесесіз.

Жүйені толық тесттен өткізу үшін оның құрамындағы жеке модульдерін де жасау керек. Ол көп уақыт алуы мүмкін, сондықтан толық жүйелік тест тәулігіне бір-ақ рет (көбінесе түнде оны жасаушылар ұйықтап жатқанда) оның жеке модульдерін тексеріп болған соң орындалады .

Бұл процесте басты рөлді регрессивтік тесттер атқарады. Программаның қателері бар ең күмәнді бөлігі болып жаңа кодтар мен оның бұрын қате шыққан бөліктері болып табылады. Осындай себептермен тесттен өткізудің маңызды бөлігі (регрессивтік тесттер негізінде) болып бұрын қолданылған тесттерді орындау ісі есептеледі; мұны орындамаса, ірі жүйе ешқашанда тұрақты болып саналмайды.

Біз қандай жылдамдықпен ескі бөліктерді алатын болсақ, жаңа қателерді де сондай жылдамдықпен енгізіп отырамыз.

Ескі қателерді түзету кезінде жаңа қателерді кездейсоқ түрде енгізіп алу жиі кездесетін әрекет болатынына назар аударыңыз. Біз түзетілген ескі қателерден гөрі жаңа қателер аз болады деп санаймыз, мұндайда жаңа қателердің әсері бұрынғыдан әлсіздеу болады. Дегенмен, әзірше біз регрессивтік тесттерді қайталауға, ал жаңа кодтар үшін, ол біздің жүйеге әсер етті (түзету кезінде енгізілген жаңа қателер үшін) деп санап, жаңа тесттер қосуға мәжбүр болып келеміз.

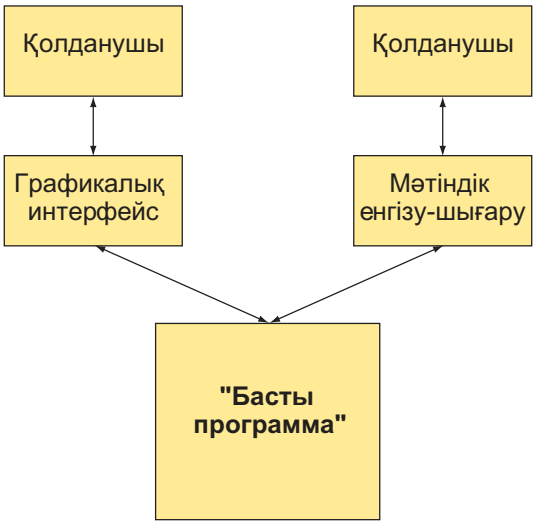
26.3.4.1 Тәуелділіктер

Сіз экран алдында отырып күрделі графикалық интерфейсі бар программаны тесттен өткізуді ойластырып отырсыз делік. Тышқанмен қай жерді шертесіз? Қандай реттілікпен? Қандай мәндерді енгізу керек? Кез келген күрделі программа үшін бұл сұрақтардың бәріне жауап беру мүмкін емес. Мұнда көптеген мүмкіндіктер бар, мысалы, экранды кездейсоқ түрде шұқылап отыратын (олар тек тамақ үшін жұмыс істер еді!) бір топ кептер әрекеттерін ұсыныс ретінде қарастыруға болар еді. Көптеген жаңа үйренушілерді жұмысқа тартып, олардың экранды қалай "шұқылайтынын" бақылау кең таралған тәжірибенің бірі, бірақ оны жүйелік стратегия деп айта алмаймыз ғой. Кез келген нақты программа белгілі бір қайталанатын тесттер жиынымен сүйемелденеді. Олар көбінесе графикалық қолданушы интерфейсін алмастыра алатын интерфейсті жобалаумен байланысты болып келеді.

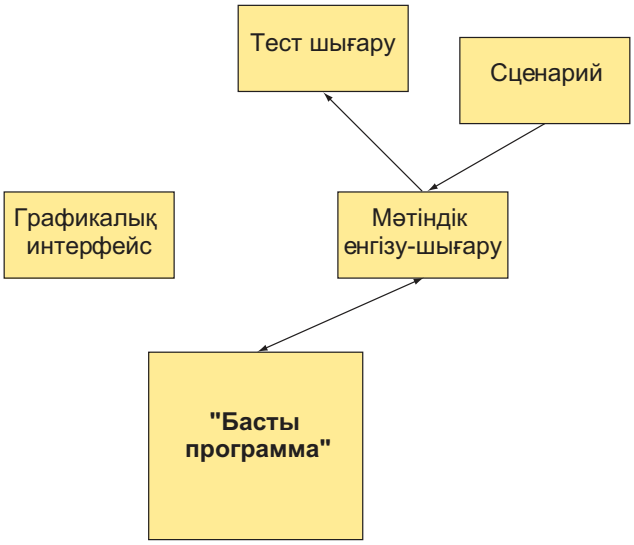
Адамға неге графикалық қолданушы интерфейсі бар экран алдында отырып, "шұқылау" керек? Мұның себебі тесттен өткізушілер қолданушының қателесіп, немесе байқамай қалып асығыстықпен, немесе білместікпен істеуі мүмкін болатын әрекеттерін алдын ала біле алмайды ғой. Тіпті ең жақсы және жүйелі түрде тесттен өткізу кезінде де жүйені нақты адамдар тексеруі қажет болып саналады. Жасалған тәжірибелер мынаны көрсетті: кез келген бір белгілі жүйені пайдаланушылар тіпті тәжірибелі жобалаушылар, конструкторлар немесе тест жасаушылар көздеріне елестете алмайтын әрекеттерді орындайды екен.

Программалаушылардың бір мақалы былай дейді: "Сен ақымақтардан қорғалған бір жүйені жасап шықсаң, табиғат онан да асқан бір ақымақты тудырады".

Сонымен, егер графикалық қолданушы интерфейсі негізгі программаның дәлме-дәл анықталған қарапайым интерфейсін пайдаланудан тұратын болса, тесттен өткізу үшін ең қолайлысы сол болар еді. Басқаша айтқанда, графикалық қолданушы интерфейсі тек енгізу-шығару мүмкіндіктерін жүзеге асырады, ал кез келген маңызды мәлімет өңдеу ісі енгізу-шығарудан бөлек орындалады. Ол үшін басқа (графикалық емес) интерфейс құру керек.



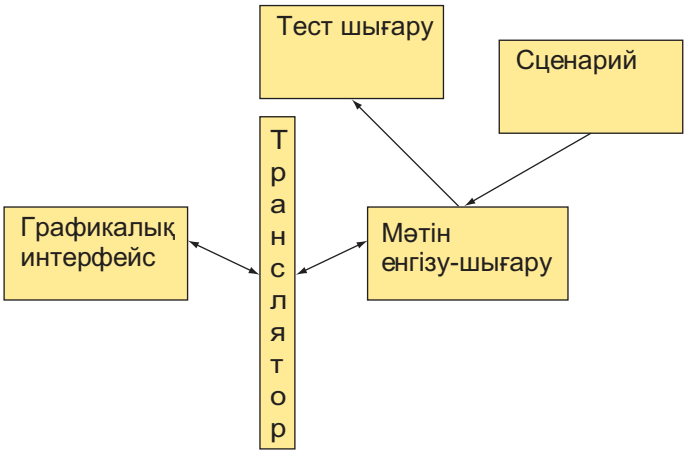
Бұл басты программа үшін сценарийлер жазуға немесе туындатуға, біз оны жеке модульдерді тесттен өткізу кезінде істегеніміздей етіп жасауға мүмкіндік береді (26.3.2 бөлімді қ.). Сонан соң біз басты программаны графикалық қолданушы интерфейсінен бөлек тесттен өткізе аламыз.



Бұл бізге графикалық қолданушы интерфейсін жүйелі түрде жартылай тесттен өткізуге мүмкіндік беретіні қызық нәрсе: біз мәтіндік енгізу-шығару әрекетін пайдалана отырып, сценарийлерді іске қоса аламыз және оның графикалық қолданушы интерфейсіне (басты программа жұмысы нәтижелерін графикалық қолданушы интерфейсіне және де мәтіндік енгізу-шығару жүйесіне де жібереміз

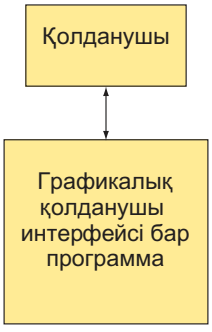
деп есептей отырып) тигізетін әсерін де бақылай аламыз. Біз онан да ары барып, графикалық қолданушы интерфейсіне шағын командалар трансляторы арқылы тікелей мәтіндік командалар жіберу жолымен графикалық интерфейсті тесттен өткізе отырып, басты программаны айналып өте аламыз.

Төменде келтірілген сурет тесттен жақсылап өткізудің екі маңызды аспектісін бейнелейді.



- Жүйенің бөліктерін, (мүмкіндігінше) жеке-жеке тесттен өткізу қажет. Айқын түрде анықталған интерфейсі бар модульдерді ғана жеке-жеке тесттен өткізуге болады.
- Тесттер (мүмкіндігінше) қайталанып орындалатындай болуы тиіс. Негізінде, адамдар араласатын бір де бір тестті дәл сол күйінде қайтадан қайталауға болмайды.

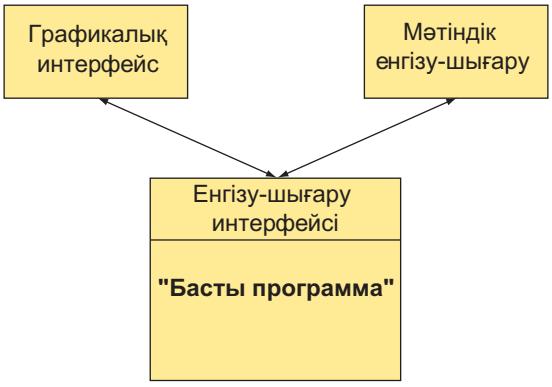
Бұрын біз айтып кеткен жобалауды тесттен өткізуді есепке ала отырып қарастырып шығайық: егер біз жобаның басынан бастап, оны тесттен өткізу жайлы ойлаған болсақ, онда жақсы ұйымдастырылған және жеңіл тесттен өтетін жүйе (26.2 бөлімді қ.) жасаған болар едік, осындай программаларды басқаларына қарағанда, тесттен өткізу анағұрлым жеңіл орындалады. Яғни, жақсы ұйымдастырылған жүйе құру? Мысал қарастырайық.



Бұл диаграмма алдыңғысына қарағанда, біраз қарапайым. Біз өз жүйемізді біраз алға қарамай-ақ, керек жерінде қолданушы мен программаның өзара қарым-қатынасын қамтамасыз ете отырып, өзімізді графикалық интерфейсіміздің кітапханасын пайдаланып құрастыра аламыз. Мүмкін бұл үшін әрі мәтіндік, әрі графикалық интерфейсі бар біздің гипотетикалық қосымшамыздан гөрі азырақ код қажет болар. Қалайша тікелей интерфейсі бар және көптеген бөліктерден тұратын біздің қосымшамыз графикалық қолданушы интерфейсі барлық код бойынша араласа орналасқан логикасы қарапайым әрі айқын қосымшадан жақсы ұйымдастырылған болып табылады?

Екі интерфейсіміз болуы үшін біз басты программа мен оның енгізу-шығару механизмі арасындағы интерфейсті тиянақты түрде анықтап алуымыз керек. Негізінде біз енгізу-шығару интерфейсінің жалпы қабатын анықтап алуымыз қажет (графикалық қолданушы интерфейсін басты программадан бөліп алу үшін қолданған транслятор тәрізді).

Біз мұндай мысалды 13-16 тараудағы графикалық интерфейс кластарынан көрген болатынбыз. Олар басты программаны (яғни, сіздің жазған кодыңызды) графикалық қолданушы интерфейсінің дайын жүйесінен: FLTK, Windows, Linux және т.с.с. бөліп тұрады. Мұндай сұлбада біз кез келген енгізу-шығару жүйесін пайдалана аламыз.



Бұл маңызды ма? Біз мұны өте маңызды деп санаймыз. Біріншіден, бұл тесттен өткізуді жеңілдетеді, ал жүйелі түрде тесттен өткізбей кодтың дұрыстығы жайлы ешнәрсе айтуға болмайды. Екіншіден, бұл программаның ауысымдылығын қамтамасыз етеді. Келесі сценарийді қарастырайық. Сіз шағын компания құрдыңыз да, өзіңіздің алғашқы программаңызды Apple жүйесі үшін жасадыңыз делік, өйткені сізге осылардың операциялық жүйесі ұнайды. Қазіргі кезде сіздің компанияңыздың жұмысы жақсы жүріп жатыр, осы кездерде сіз клиенттеріңіздің басым бөлігі өз программаларын Windows немесе Linux операциялық жүйелерінің басқаруымен орындайтынын байқадыңыз. Не істеу керек? Графикалық интерфейс (Apple Mac) командалары бүкіл программа бойынша араласып шашырай орналасқан код жұмысын қарапайым ұйымдастыру кезінде, сіз кодты қайта жазып шығуға мәжбүр боласыз. Бұл бір жағынан жақсы да, өйткені жүйелі түрде тесттен

өткізбегендіктен, оның біраз қателері де бар шығар. Дегенмен, басты программа графикалық қолданушы интерфейсінен (жүйелі түрде тесттен өткізуді жеңілдету үшін) бөлінген балама жолды қарастырып көріңізші. Мұндайда сіз басқа графикалық қолданушы интерфейсін өз интерфейстік кластарыңызбен (диаграммадағы транслятор) байланыстырып қоясыз, ал жүйенің қалған кодының басым бөлігі өзгеріссіз қалады.

Негізінде, бұл сұлба программа бөліктерін бір-бірінен бөліп тұратын тікелей "жұқа" интерфейстерді пайдалану мысалы болып саналады. Бұл біз 12.4 бөлімде көрген деңгейлерді қолдануға өте ұқсас болып келеді. Тесттен өткізу программаны айқын түрде жеке-жеке модульдерге (тесттен өткізу үшін қолдануға болатын интерфейстерімен) бөлу әрекетін орындауға итермелейді.

26.3.5 Кластарды тесттен өткізу

Жалпылама түрде айтатын болсақ, кластарды тесттен өткізу, әрбір кластың бірнеше функция-мүшесі болатындығын есепке ала отырып, модульдерді тесттен өткізу сияқты болып табылады; кластарды тесттен өткізуде жүйелерді тесттен өткізу белгілері болады. Бұл, әсіресе әрбір код ерекшеліктерін (әртүрлі туынды кластармен анықталған) қарастыратын базалық кластарға қатысты болып келеді. 14.2 бөлімдегі **Shape** класын қарастырайық.

```
class Shape {                  // түс пен стильді береді,
                               // сызықтар тізбегін сақтайды
public:
    void draw() const;
    // түсті тағайындап, сызықтар салады
    virtual void move(int dx,int dy);
    // фигураларды +=dx және +=dy қашықтыққа жылжытады
```

```
    void set_color( Color col);
    Color color() const;

    void set_style(Line_style sty);
    Line_style style() const;

    void set_fill_color(Color col);
    Color fill_color() const;

    Point point(int i) const;
    // толықтырусыз нүктелерге қол жеткізу
    int number_of_points() const;

    virtual ~Shape() { }
protected:
    Shape();
    virtual void draw_lines() const;
    // сәйкес нүктелерді сызады
    void add(Point p);               // p нүктесін қосады
    void set_point(int i,Point p);    // points[i]=p;
private:
    vector<Point> points;
    // барлық фигуралармен қолданылмайды
    Color lcolor;         // сызықтар мен символдар түсі
    Line_style ls;
    Color fcolor;         // толтыру түсі

    Shape(const Shape&); // көшіруді болдырмайды
    Shape& operator=(const Shape&);
};
```

Бұл класты тесттен өткізуді қалай бастаймыз? Алдымен тест өткізу тұрғысынан алғанда, **Shape** класының **binary_search** функциясынан нендей айырмашылығы бар екенін қарастырайық:

- **Shape** класының бірнеше функциялары бар.
- **Shape** класы объектісінің қалпы өзгеруі мүмкін (нүктелер қоса аламыз, түсін өзгертеміз, т.с.с.), яғни бір функция екінші бір функцияға әсер ете алады.
- **Shape** класының виртуалдық функциялары бар. Басқаша айтқанда, **Shape** класы объектісінің тәртібі оның негізінде (егер мұндай класс бар болса) қандай туынды класс жасалды, соған байланысты болады.

- **Shape** класы алгоритм болып саналмайды.
- **Shape** класы объектісінің өзгеруі экран мазмұнына әсер ете алады.

Соңғы әрекет ерекше жағымсыз болып саналады. Негізінде, бұл компьютер алдына **Shape** класының объектісі өзін дұрыс ұстап отырғанын бақылайтын адам отырғызып қою керектігін көрсетеді. Бұл жүйелі, әрі қайталанатын қол жетімді тесттен өткізу қағидаларына сәйкес келмейді. Біз бұдан құтылу үшін, 26.3.4.1 бөлімінде көрсетілгендей, әртүрлі қулықтарға баруымыз керек болады. Дегенмен, біз әзірше бейненің берілген қалыптан өзгеруін қадағалап отыратын бақылаушы бар деп санаймыз.

Басты бір нәрсені атап өтейік: қолданушы нүктелер қоса алады, бірақ оларды өшіре алмайды. Қолданушы немесе **Shape** класының функциялары нүктелерді оқи алады, бірақ оларды өзгерте алмайды. Тесттен өткізу тұрғысынан қарағанда, өзгертулер енгізбейтіннің (немесе, ең аз дегенде, енгізбеуі тиіс) бәрі де жұмысты жеңілдетеді деп есептеледі.

Біз нені тесттен өткізе аламыз, нені өткізе алмаймыз? **Shape** класын тесттен өткізу үшін, біз оның жеке өзін және де басқа туынды кластарымен бірге де тесттен өткізуге талпынуға тиіспіз. Дегенмен, **Shape** класының нақты бір туынды класпен дұрыс жұмыс істейтінін тексеру үшін біз сол туынды класты тесттен өткізуіміз керек.

Бұрынырақ біз **Shape** класының объектісінің төрт мәлімет-мүшелермен анықталатын өзіндік қалпы (мәні) бар деп атап өткен болатынбыз:

```
vector<Point> points;
Color lcolor;      // сызықтар мен символдар түсі
Line_style ls;
Color fcolor;      // толтыру түсі
```

Біздің **Shape** класының объектісімен бар жасай алатынымыз – оған өзгертулер енгізіп, не болатынын қадағалау. Қуанышқа орай, мәлімет-мүшелерді функция-мүшелермен анықталған интерфейс арқылы ғана өзгертуге болады.

Shape класының ең қарапайым объектісі болып **Line** класы саналады, сондықтан осындай бір объект жасаудан бастаймыз да, мүмкін болатын өзгертулер енгіземіз (тесттен өткізудің ең қарапайым стилін пайдалану арқылы):

```
Line ln(Point(10,10), Point(100, 100));
ln.draw();                    // не болғанын көрейік

// check the points:
if (ln.number_of_points() != 2)
                      cerr << "нүктелер саны дұрыс емес ";
if (ln.point(0)!=Point(10,10)) cerr << "1 қате нүкте";
if (ln.point(1)!=Point(100,100)) cerr << "2 қате нүкте";
```

```
for (int i=0; i<10; ++i) {  // объектінің жылжығанын көреміз
    ln.move(i+5,i+5);
    ln.draw();
}

for (int i=0; i<10; ++i) {
//объект бастапқы орнына келді ме, соны тексереміз
    ln.move(i-5,i-5);
    ln.draw();
}
if (point(0)!=Point(10,10))
                cerr << "жылжығаннан кейінгі 1 қате нүкте";
if (point(1)!=Point(100,100))
                cerr<< "жылжығаннан кейінгі 2 қате нүкте";

for (int i = 0; i<100; ++i) {
// түстердің дұрыс өзгергенін тексереміз
    ln.set_color(Color(i*100));
    if (ln.color() != i*100)
                cerr << "set_color мәні дұрыс емес";
    ln.draw();
}

for (int i = 0; i<100; ++i) {
// стильдің дұрыс өзгергенін тексереміз
    ln.set_style(Line_style(i*5));
    if (ln.style()!=i*5) cerr<<"set_style мәні дұрыс емес";
    ln.draw();
}
```

Негізінде, бұл программа (нүктелер, сызықтар) жасау, жылжыту, түсі және стилі сияқты параметрлерді тесттен өткізеді. Практика жүзінде, біз **binary_search** функциясын тесттен өткізген сияқты мұнан да көптеген факторларды (сценарийден ауытқуларды есепке ала отырып) есепке алуымыз керек. Біз қайтадан тестердің сипаттамаларын файлдан оқу ыңғайлы екеніне көз жеткіземіз, оған қоса, қателер жайлы да толығырақ мәліметтер ойластырамыз.

Оның үстіне, біз компьютер экраны алдына **Shape** класының объектілерін қадағалайтын адам отырғызып қою міндетті емес екеніне де көз жеткіземіз. Сонымен, бізде екі түрлі балама бар:

- адамның қадағалауына ыңғайлы түрде программа жұмысын аялдату;
- өзіміз үшін **Shape** класын оқу мен талдауды программа арқылы орындай алатын жол табу.

Біз әлі **add** (**Point**) функциясын тесттен өткізбегенімізді айта кетейік. Оны тексеру үшін біз, бәлкім, **Open_polyline** класын пайдалануға тиіс шығармыз.

26.3.6 Орындалмайтын ұсыныстарды іздеу

binary_search класының спецификациясы іздеу жүргізілетін тізбек реттелген болуы керек екендігін айқын көрсетеді. Бұл бізге бірсыпыра күрделенген модульдік тесттер жасауға мүмкіндік бере қоймайды. Дегенмен, мұнда барлық қателерді анықтай алатын (жүйелік тесттерден басқа) тест жасау әрқашанда жүзеге аса бермейтінін де айта кетейік. Біз бұдан гөрі жақсырақ тесттер жасау үшін жүйелік модульдер жайлы өз білгендерімізді (функциялар, кластар, т.с.с.) пайдалана аламыз ба?

Өкінішке орай, жоқ, пайдалана алмаймыз. Біз тест арқылы тексеруші ғана болғандықтан, кодты өзгерте алмаймыз, ал интерфейс талаптарының бұзылғанын анықтау үшін оларды әрбір шақыру алдында тексеріп отыру керек немесе әрбір шақырудың құрамына тексеруді де енгізіп қоюмыз қажет (5.5 бөлімін қ.). Егер біз өз жазған кодымызды тексеретін болсақ, онда мұндай тесттерді енгізе аламыз. Ал егер біз тек тест арқылы тексеруші ғана болатын болсақ, бірақ код жазған адамдар біздің кеңесімізді тыңдайтын болса (әрқашанда бұлай бола бермейді), онда оларға тексерілмейтін талаптар жайлы айтып, соларды тексере алатын өзгертулер енгізу керектігін түсіндіруіміз қажет.

binary_search функциясын тағы да бір рет қарастырып шығайық: біз берілген **[first:last)** кіріс тізбегінің нақты түрдегі тізбек екенін және оның реттеліп берілгендігін де (26.3.2.2 бөлімді қ.) тексере алмаймыз. Дегенмен, осындай тексеруді жүргізе алатын келесідей функция жаза аламыз:

```
template<class Iter, class T>
bool b2(Iter first, Iter last, const T& value)
{
   //[first:last) диапазоны тізбек бола ма, жоқ па, соны тексереміз:
   if (last<first) throw Bad_sequence();

   // тізбек реттеліп орналасқан ба, соны тексереміз:
   if (2<last-first)
      for (Iter p = first+1; p<last; ++p)
          if (*p<*(p-1)) throw Not_ordered();

   // бәрі жақсы, енді binary_search функциясын шақырамыз:
   return binary_search(first,last,value);
}
```

binary_search функциясына мұндай тексерулер енгізу қажет етілмейтін кездегі себептерді айтып берейік:

- **last < first** шартын бір бағытты итераторлар үшін тексеруге болмайды, мысалы, **std::list** контейнерінің итераторында **<** операторы (Б.3.2 бөлімі) жоқ. Жалпы, итераторлар жұбы арқылы тізбекті анықтау ісін тексеруді жүргізетін жақсы тәсіл жоқ, (тізбек бойынша жылжуды **first** итераторынан бастап, **last** итераторына дейін жетуді тексеру — онша жақсы идея емес).
- Тізбек мәндерінің реттеліп орналасқандығын тексеру **binary_search** функциясының өзін орындаудан гөрі көбірек жұмыс істеуді талап етеді (**binary_search** функциясын орындаудың нақты мақсаты керекті мәнді іздеп табу үшін **std::list** функция сияқты тізбек бойынша ары-бері жылжу ғана емес).

Біз енді не істей аламыз? Біз тесттен өткізу кезінде **binary_search** функциясын **b2** функциясына (бірақ оны binary_search функциясын кез келген бағытта қатынас құра алатын итераторлар арқылы шақыру үшін ғана қолдану керек) ауыстыруымызға болады. Бұған балама тәсіл ретінде біз **binary_search** функциясын жазған адамнан оның кодын алып, соған өз фрагментімізді енгізіп қоюға болар еді.

```
template<class Iter, class T>  // ескерту: жалған коды бар
bool binary_search (Iter first, Iter last, const T& value)
{
   if (тест іске қосылған болса) {
      if (Iter кез келген бағыттағы итератор) {
      // [first:last) диапазоны тізбек пе, соны тексереміз:
        if (last<first) throw Bad_sequence();
      }
         // тізбек реттелген бе, соны тексереміз:
         if (first!=last) {
               Iter prev = first;
               for (Iter p = ++first; p!=last; ++p, ++ prev)
                     if (*p<*prev) throw Not_ordered();
         }
   }

       // енді binary_search функциясын орындаймыз
}
```

Тест іске қосылған болса шартының мағынасы тесттен өткізуді ұйымдастыру тәсіліне тәуелді (нақты ұйымдағы нақты жүйе үшін) болғандықтан, оны жалған код (псевдокод) түрінде қалдыра аламыз: өз кодыңызды тесттен өткізу кезінде **test_enabled** айнымалысын пайдалануыңызға болады. Біз итератор қасиетін түсіндіргіміз келмегендіктен, **Iter кез келген бағыттағы итера-**

`тор` шартын да жалған код түрінде қалдырдық. Егер сізге осындай тест нақты түрде қажет болып жатса, C++ тіліндегі толығырақ оқулықтан *итераторлар қасиеті (iterator traits)* тақырыбын қарап шығыңыз.

26.4 Тесттен өткізуді есепке ала отырып жобалау

Программа жазуға кіріскенде, біз жұмыс нәтижесінде ол толық және дұрыс программа болып шығатынына сенімді боламыз. Біз ол үшін оны тесттен өткізу керек екендігін де білеміз. Сондықтан программа жасай отырып, біз оның тесттен өтуі керек екендігін басынан бастап ойластыруымыз қажет. Көптеген жақсы программалаушылар "Алдын ала және жиі-жиі тесттен өткіз" деген қағиданы ұстанып программа жазады және оны қалай тесттен өткізу керек екендігін білмейтін болса, ондай программаны жазбайды да. Программа жазудың бастапқы кезеңдерінде оны тесттен өткізу жайлы ойлану кейіннен оның қатесі болмайтынына (немесе олардың жылдам табылатындығына) кепілдік береді. Біз осы тәсілді дұрыс деп санаймыз. Кейбір программалаушылар өз программасының модульдеріне арналған тестті сол модульдерді жазбай тұрып құруға кіріседі.

26.3.2.1 және 26.3.3 бөлімдеріндегі мысалдар осының айқын дәлелі бола алады.

- Кодтарыңызды оларға тест жазуға болатындай етіп, айқын анықталған интерфейстер түрінде құруға тырысыңыз.
- Сақтауға, талдауға және қайта пайдалануға болатындай етіп, программаңыздағы операцияларды мәтін түрінде сипаттайтын тәсілдер қолдануды қарастырыңыз. Бұл шығару операцияларына да қатысты болып табылады.
- Қате аргументтерді жүйелік тесттен өткізуге дейін анықтап алу үшін, шақыратын код құрамына тағы тексерілмейтін ұсыныстар (assertions) үшін тесттер енгізіңіз.
- Тәуелділіктерді азайтыңыз да, оларды тікелей етіп жасаңыз.
- Ресурстарды басқару стратегиясын айқын түрде орындаңыз.

Философиялық тұрғыдан алғанда, мұны толық жүйелерді және ішкі жүйелерді тексеруге арналған модульдік тесттен өткізу әдістерін пайдалану деп қарастыруға болады.

Егер программаның жұмыс өнімділігінің маңызы аса үлкен болмаса, онда оның ішінде ұсыныстарды (талаптарды, алғышарттарды) тексеру фрагменттерін толығынан қалдырып кетуге болады, кері жағдайда олар тексерусіз қалар еді. Бірақ мұны тұрақты түрде істеуге болмайтын өз себептері бар. Мысалы, біз тізбектің реттелгенін тексеру күрделі болатынын және ол `binary_sort` функциясына қарағанда, көбірек жұмыс жасауды керек ететінін көрсеткенбіз. Сондықтан мұндай тексерулерді таңдау арқылы енгізетін немесе енгізбейтін жүйе жасаған дұрысырақ

болады. Көптеген жүйелер үшін қолданушыларға ұсынылатын программаның соңғы нұсқасында бірсыпыра қарапайым тексерулер санын қалдырған қолайлы болып саналады: кейде мүмкін емес сияқты көрінетін оқиғалар да орын алып жатады, солар туралы программаның тұрып қалғанынан гөрі, нақты қате кеткені туралы мәлімедемелерден білген абзал.

26.5 Жөндеп түзету

Программаны жөндеп түзету – бұл қағида жетекші рөл атқаратын техника және қағида мәселесі. 5-тарауды тағы бір рет оқып шығыңызшы. Программаны жөндеп түзетудің оны тесттен өткізуден қандай айырмашылығы бар екеніне назар аударыңыз. Бұл екеуінде де қателер анықталады, бірақ жөндеп түзетуде бұл жүйелі түрде атқарылмайды, мұнда көбінесе белгілі қателер табылып анықталады да, кейбір қасиеттер жүзеге асырылады.

Жөндеп түзету кезінде орындалатын жұмыстардың барлығы да тесттен өткізу кезінде де атқарылады. Аздап артық айтар болсақ, біз тесттен өткізуді ұнатамыз, ал бірақ жөндеп түзетуді жақсы көрмейміз. Модульдерді жасау мен жобалаудың алғашқы кезеңдерінде оларды жақсылап тесттен өткізу программаны жөндеп түзету жұмысын барынша азайтуға көмектеседі.

26.6 Жұмыс өнімділігі

Программа пайдалы болуы үшін оның дұрыс болуы жеткіліксіз. Тіпті оның пайдалы болатын барлық мүмкіндіктері болатын болса да, ол оның үстіне қажетті жұмыс өнімділігін де қамтамасыз етуі тиіс. Жақсы программа тиімді де болып табылады; басқаша айтқанда, ол керекті уақыт мөлшерінде қолжетімді ресурстарды пайдалану арқылы орындалады. Абсолюттік тиімділік ешкімді қызықтырмайды және де оның кодын күрделендіре отырып, сүйемелдеу құнын да қымбаттатып (басқа компьютерлерге ауыстыру мен жұмыс өнімділігін баптау істерін орындау арқылы), орындалу жылдамдығын арттыру да жалпы жүйеге зиянды (қателерінің көптігімен және түзету жұмыстарының да көлемінің артуымен) болуы мүмкін.

Программаның (немесе оның модулінің) керекті мөлшерде тиімді екенін қалай білуге болады? Абстрактілі түрде бұл сұраққа жауап беру мүмкін емес. Қазіргі аппараттық жабдықтамалар жылдам жұмыс істейтін болғандықтан, көптеген программалар үшін бұл мәселе көтерілмейді де. Біз өндіріске енгізгеннен кейін қателерді анықтау (диагностика) мүмкіндіктерін арттыру мақсатында әдейі түзету режимінде компиляциядан өткізілген (яғни, керекті деңгейден 25 есе төмен жұмыс істейтін) программаларды (бұл ең жақсы кодпен де орындалуы мүмкін, егер ол "бір жерлерде" жасалған программалармен бірге жұмыс атқаратын болса) кездестірдік.

Сондықтан, "Программа керекті деңгейде тиімді ме?" деген сұраққа мынадай жауап: "Қызықты тест орындалатын уақытты өлшеңіз" беруге болады. Әрине,

бұл үшін өзіміздің ақырғы қолданушыларымызды жақсы білуіміз қажет, яғни олардың нені қызықты деп санайтынын және сол қызықты тесттерді орындау үшін қандай уақыт аралығын тиімді деп есептейтінін білу керек. Логикалық түрде ойлана отырып, біз тесттерді орындау кезінде жай ғана секунд өлшеуішінде уақытты белгілейміз де, ол шектен тыс көп уақыт алмағанын қадағалаймыз. `clock()` функциясы арқылы (26.6.1 бөлім) тесттердің орындалу уақытын бағалай отырып, автоматты түрде салыстырамыз. Бұған балама ретінде (немесе қосымша түрінде) тесттердің орындалу уақыттарын жазып ала отырып, бұрынғы алынған нәтижелермен салыстыруға болады. Бұл тәсіл программаның жұмыс өнімділігін регрессивтік тесттен өткізу әдісін еске салады.

Жұмыс өнімділігінің нашар көрсеткіштерін берген нұсқалар әдетте алгоритмнің дұрыс таңдалмағанымен түсіндіріледі де, программаны түзету кезеңінде анықталуы мүмкін. Программаларды мәліметтердің көлемді жиындарымен тесттен өткізудің бір мақсаты тиімсіз алгоритмдерді анықтау болып табылады. Мысал ретінде, матрицаның жолдарында (26-тараудағы `Matrix` класын пайдалану арқылы) орналасқан элементтерді қосатын программаны қарастырайық.

Біреулер бұған сәйкес келетін функцияны пайдалануды да ұсынды.

```
double row_sum(Matrix<double,2>m,int n);
// m[n] жиымы элементтерін қосады
```

Сонан соң сол біреу осы функцияны қосындылар векторын туындату (генерациялау) үшін қолдана бастады, мұндағы `v[n]` – алғашқы `n` жолда орналасқан элементтер қосындысы.

```
double row_accum(Matrix<double,2> m,int n)
// m[0:n) жиымы элементтерін қосу
{
   double s = 0;
   for (int i=0; i<n; ++n) s+=row_sum(m,i);
   return s;
}

// m матрицасының жолдарындағы жинақталған қосындыларды
есептейді:
vector<double> v;
for (int i = 0; i<m.dim1(); ++i)
   v.push_back(row_accum(m,i+1));
```

Осы кодтың модульдік тест немесе жүйелік тест бөлігі ретінде орындалатынын көзге елестетіңіз. Кез келген жағдайда, матрица көлемді болып кететін болса, сіз бір қызық жайттарды байқайсыз: негізінде, программаны орындауға қажетті уақыт `m` матрицасының көлемінің квадратына тәуелді болады. Неге? Әңгіме

мынада: біз бірінші жолдағы барлық элементтерді қостық та, сонан соң оған екінші жол элементтерін қостық (қайтадан бірінші жол элементтерін біртіндеп түгел қарастырып шығып), сонан кейін үшінші жол элементтерін қосамыз (тағы да бірінші және екінші жол элементтерін біртіндеп түгел қарастырып шығып), т.с.с.

Егер бұл мысалды онша дұрыс емес деп санасаңыз, онда `row_sum()` функциясы мәліметтер базасын мәліметтер алу үшін пайдаланса, не болатынын қараңыз. Дискіден мәлімет оқу компьютердің жедел жадынан мәлімет оқудан гөрі өте көп мыңдаған есе жай орындалады.

Сіздің: "Ешкім ешқашанда мұндай ақылсыз программа жасай алмайтын шығар!", – деуіңіз мүмкін. Кешіріңіз, бірақ біз бұдан да сорақыларын көргенбіз, оның үстіне, программа кодының ішінде жасырын тұрған нашар (жұмыс өнімділігі жағынан) алгоритмді көбінесе оңай анықтай да алмайсыз. Сіз бұл кодты алғаш көргенде, жұмыс өнімділігімен мәселе барын байқадыңыз ба? Мәселені мақсатты түрде іздемесе, оны анықтау да қиын болады. Бір серверде кездескен қарапайым нақты программаны келтірейік:

```
for(int i=0;i<strlen(s);++i) {
/*  s [i] жиымымен бір әрекет жасаймыз  */ }
```

Көбінесе `s` айнымалысы көлемі шамамен 20 К болатын тіркес түрінде болады.

Программаның жұмыс өнімділігімен байланысты мәселелердің барлығы алгоритмнің әлсіздігімен түсіндірілмейді. Негізінде, өзіміз жазатын кодтың басым бөлігінің (26.3.3 бөлімінде біз көрсеткендей) алгоритмін әлсіз деп айтуға болмайды.

Мұндай "алгоритмдік емес" мәселелер әдетте жобалауды дұрыс жасамағандықтан пайда болады. Солардың бірсыпырасын келтіре кетейік:

- Ақпаратты қайталай отырып қайта есептеу (жоғарыдағы мысал тәрізді).
- Бір фактыны қайталап тексере беру (мысалы, циклге кірген сайын индекстің берілген диапазоннан тысқары шықпайтынын тексеру немесе бір функциядан екінші бір функцияға өзгеріссіз берілетін аргументтерді де қайталап тексеріп отыру).
- Дискідегі (немесе желідегі) мәліметті қайталап оқуға талпыну.

Осындағы "*қайталау*" сөзіне назар аударыңыз. Әрине, мұнда біз бұл сөзге "бос қайталану" деген мағына беріп отырмыз, өйткені жұмыс өнімділігіне өте көп қайталанатын әрекеттер ғана әсер етеді. Біз функциялар аргументтерін және цикл айнымалыларын қатаң тексеруді жақтаймыз, бірақ егер біз бір айнымалыны миллиондаған рет тексеретін болсақ, олар программаның жұмыс өнімділігін төмендетіп жібереді. Егер қадағалаулар нәтижесінде жұмыс өнімділігінің төмендегені сезілетін болса, біз қайталанатын әрекеттерді болдырмауымыз керек. Бірақ мұны программаның баяу жұмыс істейтіні анық байқалмай, орындаушы болмаңыз. Өйткені орынсыз программаға тиісу көптеген қателердің шығуына себепші болады да, уақытты да зая кетіреді.

26.6.1 Уақытты өлшеу

Код фрагментінің ойдағыдай жылдам істейтінін қалай білуге болады? Берілген операцияның жылдамдығының дұрыс екеніне қалай көз жеткіземіз? Уақытты өлшеуге байланысты көптеген жағдайларда жай ғана сағатқа (қол сағатына, қабырға сағатына, секунд өлшеуішіне, т.б.) қарау жеткілікті. Әрине, бұл ғылыми және дәл тәсіл емес, бірақ егер назар аударатындай ешнәрсе болмаса, программа дұрыс жұмыс істеп тұр деп есептеуге болады. Дегенмен, бұл тәсіл программаның жұмыс өнімділігі маңызды рөл атқаратын жағдайлар үшін сәйкес келмейді.

Сізге өте шағын уақыт аралықтарын өлшеп отыру керек болып секунд өлшеуішін қолға алғыңыз келмесе, онда сізге компьютер мүмкіндіктерін пайдалана білу қажет, өйткені ол арқылы уақыт өлшеу онша қиын емес. Мысалы, Unix жүйесінде команда алдына `time` сөзін қосып жазу оның орындалу ұзақтығын беру ісін жүзеге асырады. Оған қоса, `time` командасын да қолдану арқылы да бастапқы `x.cpp` файлының қанша уақытта компиляциядан өткенін анықтай аласыз. Әдетте компиляция мынадай:

```
g++ x.cpp
```

команда бойынша атқарылады. Команданың компиляциядан өту ұзақтығын білу үшін алдына `time` сөзін қосу жеткілікті.

```
time g++ x.cpp
```

Мұнда жүйенің өзі `x.cpp` файлын компиляциядан өткізіп, жұмсалған уақытты экранға шығарады. Бұл шағын программалардың орындалу ұзақтығын білуге арналған қарапайым және тиімді тәсіл болып табылады. Өлшеуді бірнеше рет орындауды ұмытпаңыз, өйткені программаның орындалу уақытына компьютердегі орындалып жататын басқа да әрекеттер әсер етуі мүмкін. Егер сіз үш рет бірдей жауап алсаңыз, онда нәтижеңізге сенуге болады.

Ал егер сіз ұзақтығы бірнеше миллисекунд қана болатын уақыт аралығын өлшегіңіз келсе ше? Егер сіз өз программаңыздың бір бөлігінің ғана орындалу кезеңін нақты түрде өлшегіңіз келсе ше? Енді `do_something()` функциясының орындалу мерзімін өлшейтін стандартты кітапханадағы `clock()` функциясын пайдалану мысалын көрсетейік.

```
#include <ctime>
#include <iostream>
using namespace std;

int main()
{
    int n = 10000000;  // do_something()  n рет қайталанады
```

```
    clock_t t1 = clock();              // есептеу басы
    if (t1 == clock_t(-1)) {
    // clock_t(-1)  "clock() істемейді" дегенді білдіреді
       cerr << "кешіріңіз, таймер істемейді \n";
       exit(1);
    }

    for (int i = 0; i<n; i++) do_something(); // өлшеу циклі

    clock_t t2 = clock();              // есептеу соңы
    if (t2 == clock_t(-1) {
       cerr << "кешіріңіз, таймер толып кетті \n";
       exit(2);
    }

    cout << "do_something() " << n << " рет орындалды, ол "
       << double(t2-t1)/CLOCKS_PER_SEC <<"сек уақыт алды"
       << " (өлшеу дәлдігі: "
       << CLOCKS_PER_SEC << " сек)\n";
}
```

`clock()` функциясы `clock_t` типіндегі нәтиже қайтарады. `double (t2-t1)` мәнін бөлу алдында тікелей түрлендіру қажет, өйткені `clock_t` типі бүтін сан болуы мүмкін. `clock()` функциясының дәл іске қосылу сәті оның іске асырылуына байланысты болады; `clock()` функциясы программа орындалуының бір сеансы кезіндегі уақыт интервалдарын өлшеу ісін атқарады. `clock()` функциясы қайтаратын `t1` және `t2` мәндері үшін `double(t2-t1)/CLOCKS_PER_SEC` саны `clock()` функциясын екі шақыру арасында өтетін уақыттың ең дәл жуық мәнін (секундпен) береді. `clocks _per_sec` (такт секундына) макросы `<ctime>` тақырыбында сипатталған.

Егер `clock()` функциясы процессор үшін қарастырылмаған болса немесе уақыт интервалы тым үлкен болса, `clock()` функциясы `clock_t(-1)` мәнін қайтарады.

`clock()` функциясы секунд бөліктерінен бастап бірнеше секундқа дейінгі уақыт интервалдарын өлшеуге арналған. Мысалы, егер (бұл жиі болып тұрады) `clock_t` типі 32-биттік таңбасы бар `int` типі болатын болса және де `CLOCKS_PER_SEC` параметрі 1 000 000-ға тең болса, біз 0-ден 2000 секундқа (жарты сағаттай) дейінгі уақыт интервалдарын микросекундпен өлшей аламыз.

Есіңізге саламыз: нәтижелері шамамен бірдей болғанымен, қайталауға болмайтын кез келген уақыт өлшемдеріне сенуге болмайды. "Нәтижелері шамамен бірдей" деген нені білдіреді? Шамамен 10% дегенді білдіреді. Біз бұрын айтқандай, қазіргі компьютерлер *жылдам* болып табылады, олар секундына миллиард нұсқау орындай алады. Бұл, егер бір операция ондаған мың рет қайталанбаса немесе егер

программа өте баяу орындалмаса, мысалы, мәліметтерді дискіге жаза отырып немесе веб желісін қолдану арқылы, оны сіз бір де бір рет өлшей алмайсыз дегенді білдіреді. Веб-ті пайдаланғанда, сіз әрекетті жүздеген рет қайталап орындауыңыз керек, бірақ программаның баяу орындалуынан сіз сезіктене бастауыңыз қажет.

26.7 Сілтемелер

Stone, Debbie, Caroline Jarrett, MarkWoodroffe, and Shailey Minocha. *User Interface Design and Evaluation.* Morgan Kaufmann, 2005. ISBN 0120884364.

Whittaker, James A. *How to Break Software; A Practical Guide to Testing.* Addison-Wesley, 2003. ISBN 0321194330.

ТАПСЫРМА

`binary_search` функциясын тесттен өткізіңіз.

1. 26.3.2.2 бөліміндегі `Test` класы үшін енгізу операторын іске асырыңыз.
2. 26.3 бөліміндегі тізбектер үшін тесттер файлын толтырыңыз.
   2.1 `{1,2,3,5,8,13,21} // "қарапайым тізбек"`
   2.2 `{}`
   2.3 `{1}`
   2.4 `{1,2,3,4}  // элементтердің тақ сандары`
   2.5 `{1,2,3,4,5} // элементтердің жұп сандары`
   2.6 `{1,1,1,1,1,1,1} // барлық элементтер тең`
   2.7
```
{0,1,1,1,1,1,1,1,1,1,1,1,1}
   //басында әртүрлі элементтер
```
   2.8
```
{0,0,0,0,0,0,0,0,0,0,0,0,1}
   //соңында әртүрлі элементтер
```
3. 26.3.1.3 бөлімін негізге алып, келесі нұсқаларды туындататын (генерациялайтын) программаны орындаңыз.
   3.1 Өте үлкен тізбек (үлкен тізбек деп нені айтамыз және неге?).
   3.2 Элементтерінің саны кездейсоқ болып келетін он тізбек.
   3.3 Элементтері кездейсоқ болып келетін он тізбек `0, 1, 2 ... 9` (бірақ реттелген).
4. Бұл тесттерді мынадай тіркестер тізбектері үшін `{Bohr Darwin Einstein Lavoisier Newton Turing}` қайталаңыз.

БАҚЫЛАУ СҰРАҚТАРЫ

1. Қатеге байланысты шығатын ең нашар оқиғалар түрінде құрылған қысқаша сипаттамамен сүйемелдей отырып, қосымша программалар тізімін жасаңыз; мысалы, ұшақты басқару – авиа апат: 231 адам қаза болған; жабдықтар шығыны 500 млн. доллар.
2. Неге біз программаның дұрыс жұмыс істейтінін дәлелдей алмаймыз?
3. Модульдік және жүйелік тесттен өткізу айырмашылықтары неде?
4. Регрессивтік тесттен өткізу деген не және ол неге маңызды болып саналады?
5. Тесттен өткізудің мақсаты неде?
6. `binary_search` функциясы неге өз талаптарын тексермейді?
7. Егер біз мүмкін болатын барлық қателерді тексере алмайтын болсақ, онда бірінші кезекте қандай қателерді іздеу керек?
8. Элементтер тізбегімен жұмыс істейтін кодтың қандай жерлерінен қате табуға болады?
9. Үлкен мәнді сандар болғанда, программаны неге тесттен өткізу керек болып саналады?
10. Неге тесттер көбінесе кодтар түрінде емес мәліметтер түрінде беріледі?
11. Кездейсоқ шамаларға негізделген көптеген тесттерді қашан және неге пайдаланамыз?
12. Неге графикалық қолданушы интерфейсін пайдаланатын программаларды тесттен өткізу қиын болып табылады?
13. Жеке модульді тексеру кезінде нені тесттен өткізу керек?
14. Тесттен өткізушілік пен ауысымдылық бір-бірімен қандай байланыста болады?
15. Функцияға қарағанда, класты тесттен өткізу неге қиындау болады?
16. Тесттерді қайталап орындау мүмкіндігі неге маңызды болып саналады?
17. Тесттен өткізуші модульдің тексерілмейтін ұсыныстарға (алғышарттарға) негізделгенін білгеннен кейін не істей алады?
18. Жобалаушы/конструктор тесттен өткізуді қалай жақсарта алады?
19. Программаны тесттен өткізу мен оны түзетіп жөндеудің айырмашылығы неде?
20. Программа жұмыс өнімділігінің маңыздылығы неде?
21. Программаның жұмыс өнімділігі жайлы мәселелер жеңіл пайда болатындығын көрсететін екі (немесе одан көп) мысал келтіріңіз.

ТҮЙІНДІ СӨЗДЕР

`clock()`
алғышарт
дәлелдеу
енгізу (мәлімет)
жүйелік тест
қалып-күй
модульдік тест
регрессия
соңғышарт
тармақталу
тесттен өткізу
тесттен өткізуді есепке алып жобалау
тесттен өткізу (қара жәшік тәсілі)
тесттен өткізу (мөлдір жәшік тәсілі)
тесттер байланысы
тестті қамту
уақытты өлшеу
ұсыныс
шығару (нәтиже)

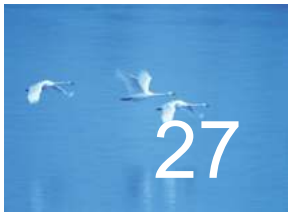
ЖАТТЫҒУЛАР

1. 26.1 бөліміндегі `binary_search` алгоритмін 26.2.1 бөліміндегі тесттермен орындап шығу керек.
2. `binary_search` функциясын кез келген типтегі элементтерді өңдеу үшін тесттен өткізу ісін баптаңыз. Сонан соң сол арқылы `string` типіндегі элементтер тізбегі мен жылжымалы нүктелі сандарды тесттен өткізіңіз.
3. 26.2.1 бөліміндегі жаттығуды аргумент ретінде салыстыру критериін алатын `binary_search` функциясы нұсқасымен қайталаңыз. Қосымша аргумент арқылы пайда болатын қателер шығуы үшін жаңа мүмкіндіктер тізімін жасаңыз.
4. Тізбекті бір рет беріп алып, сол тізбек үшін бірнеше тест өткізуді орындайтын тесттік мәліметтер форматын жасап шығыңыз.
5. `binary_search` функциясы үшін жасалған тесттер жиынына жаңа тест қосып, тізбекті өзгерту кезінде пайда болатын (мүмкіндігі төмен) қатені анықтауға тырысыңыз.
6. 7-тараудағы калькуляторды файлдан мәлімет енгізу мен файлға мәлімет шығаруды (немесе операциялық жүйенің енгізу-шығару бағытын өзгертетін мүмкіндіктерін пайдаланып) қарастыра отырып, аздап толықтырып шығыңыз. Сонан соң оған толыққанды тесттер жиынын жасаңыз.
7. 20.6 бөліміндегі қарапайым мәтіндік редакторды тесттен өткізіңіз.
8. 12-15 тараулардағы графикалық қолданушы интерфейсі кітапханасына мәтіндік интерфейс қосыңыз. Мысалы, `Circle(Point(0,1),15)` тіркесі `Circle(Point(0,1),15)` функциясын шақыруды генерациялауы тиіс. Осы мәтіндік интерфейсті "балалар жасаған сурет" – жазықтықтағы төбесі жабылған екі терезесі мен бір есігі бар шағын үйшік – салуға пайдаланыңыз.

9. Графикалық интерфейс кітапханасына мәтін шығару форматын қосыңыз. Мысалы, `Circle(Point(0,1),15)` тіркесін шығару ағымына шақыруды орындағанда, `Circle(Point(0,1),15)` тіркесі шығарылуы керек.
10. 9-жаттығудағы мәтіндік интерфейсті пайдаланып, графикалық қолданушы интерфейсі кітапханасы үшін сапасы жоғарылау тест жазып шығыңыз.
11. 26.6 бөлімінде келтірілген мысалдағы қосындылау амалын орындау уақытын бағалаңыз, ондағы `m` – көлемі 100, 10 000, 1 000 000 және 10 000 000 болатын квадраттық матрица. [-10:10) диапазонынан алынған кездейсоқ мәндерді пайдаланыңыз. Ондағы `v` шамасын есептеу процедурасын, тиімдірек (`O(n^2)` емес) алгоритмді пайдаланып, қайта жазып шығыңыз да, оның орындалу уақытын бұрынғымен салыстырыңыз.
12. Жылжымалы нүктелі кездейсоқ сандар генерациялайтын программа жазыңыз да, оларды `std::sort()` функциясы арқылы сұрыптаңыз. `double` типіндегі 500 мың санды және `double` типіндегі 5 миллион санды сұрыптауға кететін уақытты өлшеңіз.
13. Алдыңғы жаттығудағы тәжірибені ұзындығы [0:100) интервалында жататын кездейсоқ сөз тіркестерімен қайталап шығыңыз.
14. Алдыңғы жаттығуды `vector` емес, сұрыптау қажет етілмейтін `map` контейнерін пайдаланып қайталап орындаңыз.

СОҢҒЫ СӨЗ

Программалаушылар ретінде біз қарапайым түрде іске қосылып және мүмкіндігінше, бірден жұмыс істеп кететін жақсы программалар жайлы армандаймыз. Бірақ өмірде басқаша: бірден дұрыс жұмыс істеп кететін және өзіңіз (немесе әріптесіңіз) толықтыру барысында оған қателік енгізбейтін программа жазып шығу қиын. Тесттен өткізу, оны есепке ала отырып жобалауды қоса алғанда, – бұл жұмысыңыздың нәтижесінде сіздің жасаған жүйеңіздің дұрыс істеуіне кепілдік беретін басты тәсіл. Жоғары технологиялар үстемдік еткен әлемде өмір сүре отырып, біз әрбір жұмыс күнінің соңында тест жасаушыларды ризашылықпен еске алуымыз керек (олар көбінесе ұмытылып кетеді).



С программалау тілі

С – типтерді қатаң бақылауы және әлсіз тексеруі бар программалау тілі

- Деннис Ритчи

Бұл тарау С++ тілін білетін адамның көзқарасы тұрғысынан С программалау тілі мен оның стандартты кітапханасына арналған қысқаша шолу болып табылады. Мұнда С тілінде жоқ С++ тілінің қасиеттері көрсетілген және С тілінде программалаушыларға оларсыз қалай жұмыс істеуге болатынына мысалдар келтірілген. С және С++ тілдері арасындағы айырмашылықтар және де оларды тіркес пайдалану мәселелері қарастырылған. Енгізу-шығару мысалдары, операциялар тізімі, компьютер жадын басқару және де сөз тіркестері операцияларын бейнелеу мәліметтері келтірілген.

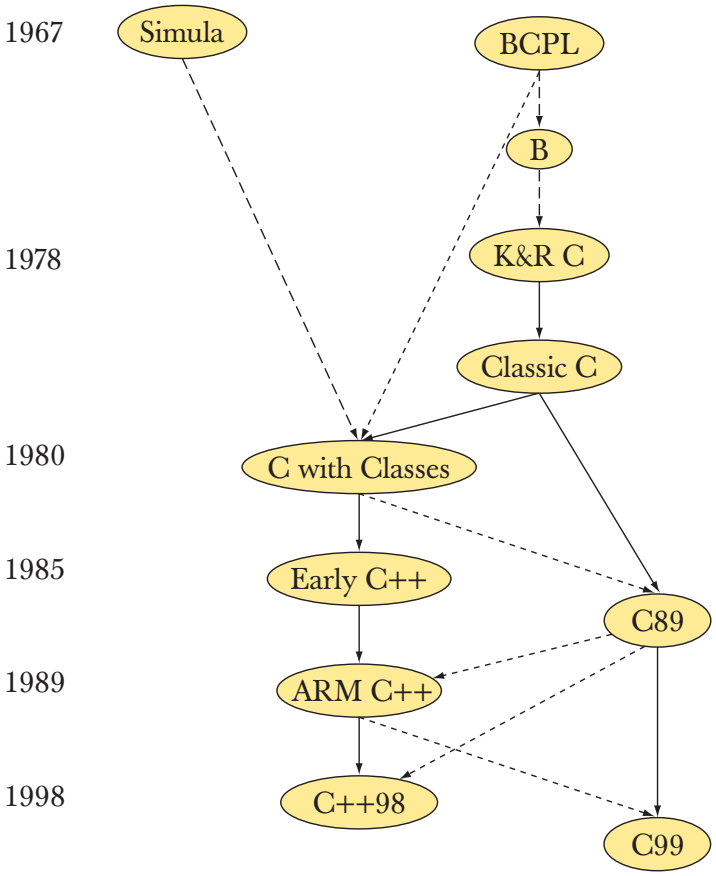
27.1 C және C++ тілдері: ағайындар

C программалау тілін Bell Labs компаниясындағы Деннис Ритчи ойлап шығарып, оны жүзеге асырды. Ол Брайан Керниган (Brian Kemighan) мен Деннис Ритчидің (Dennis Ritchie) *The C Programming Language* кітабында баяндалған. Ол (ауызекі әңгімеде "K&R" деген атпен белгілі) C тіліне ең жақсы кіріспе болды және программалаудан ең жақсы оқулық болып табылатын да шығар (22.2.5 бөлімін қ.). C++ тілінің бастапқы анықталу мәтіні 1980 жылы Деннис Ритчи жазған C тілін анықтау редакциясында болған еді. Осыдан кейін екі тіл де өздігінен жеке-жеке жетілдіріле бастады. C++ тілі сияқты C тілі де қазіргі кезде ISO стандартымен бекітілген.

Біз C тілін, негізінен, C тілінің ішкі бір нұсқасы ретінде қарастырамыз. Сондықтан, C++ тілі тұрғысынан қарғанда, C тілін сипаттау мәселелері 2 сұраққа келіп тіреледі.

- C тілі C++ тілінің ішкі бір нұсқасы болып қарастырылмайтын сәттерді сипаттау.
- C++ тілінің C тілінде жоқ қасиеттерін және де осы кемшіліктердің орнын толтыратын мүмкіншіліктері мен тәсілдерін сипаттау.

Тарихи тұрғыдан алғанда, қазіргі C++ тілі мен қазіргі C тілі "ағайынды" болып табылады. Бұл екеуі де "классикалық C" тілінің, яғни C тілінің бір диалектісінің мұрагері болып саналады. Ол Керниган мен Ритчидің *The C Programming Language* кітабының бірінші басылымында сипатталған, онда оған құрылымдар мен тізбелерді меншіктеу қосылған болатын.



Қазіргі кезде практикалық тұрғыдан қарасақ, барлық жерде дерлік C89 нұсқасы қолданылады (K&R[1] кітабының 2-ші басылымында сипатталған). Міне, осы бөлімде біз осы нұсқаны баяндаймыз. Кей жерлерде бұл нұсқадан басқа бұрынғыша классикалық C тілі қолданылады, C 99 нұсқасын қолданудың да бірнеше мысалдары бар, бұл бірақ C++ және C89 тілін білетін оқырмандарға қиын мәселе болмауы керек.

C және C++ тілдері Нью-Джерси штатындағы Bell Labs (Computer Science Research Center of Bell Labs), Мюррей-Хилл (Murray Hill, New Jersey) компанияларының компьютерлік ғылымдары зерттеу орталығының ізбасары болып табылады (менің офисім де Деннис Ритчи мен Брайан Кернигганның офистерімен тіркес тұрғанын айта кетейін).

Екі тіл де қазіргі кезде анықталған, олар ISO стандарттау комитеттерімен бақыланады. Әрбір тілге арналған көптеген нұсқалар жасалған. Көбінесе бұл нұсқалар екі тілді де сүйемелдейді, мұндағы қажетті тіл бастапқы файлдың кеңейтілуін көрсету жолымен орнатылады. Басқа тілдермен салыстырғанда, C және C++ тілдерінің екеуі де көптеген платформаларда өте кең түрде қолданыс тапқан.

Екі тіл де толық жасалып, қазіргі кезде күрделі программалық тапсырмаларды орындауда екпінді түрде қолданылып келеді. Олардың кейбіреулерін атап өтейік:

- Операциялық жүйелердің ядросы
- Құрылғылар драйверлері
- Құрамдас жүйелер
- Компиляторлар
- Байланыс жүйелері

C және C++ тілінде жазылған эквивалентті программалар арасында өнімділігі жағынан ешқандай айырмашылық жоқ.

[1] Орысша аудармасы: Керниган У., Ритчи Д. Язык программирования С. 2-изд. – М.: ИД Вильямс, 2006.

C++ тілі сияқты C тілі де кеңінен қолданылады. Екеуі бірге алынғанда, олар жер жүзіндегі программалық жабдықтамаларды жасайтын ең ірі қоғамдастықты құрайды.

27.1.1 C және C++ тілдерінің үйлесімділігі

"C/C++" атауын жие кездестіруге болады. Бірақ мұндай тіл жоқ. Бұл атауды қолдану білімсіздік белгісі болып табылады. Біз бұл атауды үйлесімділік сұрақтары ішінде және осы екі тілді тіркес қолданатын программалаушылар қоғамдастығы жайлы айтқан кезде пайдаланамыз.

C++ тілі негізінен, бірақ толығынан емес, C тілінің жоғарғы сатылық деңгейі болып табылады. Аздаған айырмашылықтары болмаса, C және C++ тілдерінің ортақ конструкцияларының мағынасы (семантикасы) бірдей болып келеді. C++ тілі "C++ тіліне мүмкіндігінше жақын болып, бірақ берілген деңгейден тыс жақын болмайтындай" етіп жасалған. Ол бірнеше мақсатты көздейді:

- Ауысу қарапайымдылығы.
- Үйлесімділік.

C++ тілінің C тілімен үйлеспейтін көптеген қасиеттері оның типтерді қатаң бақылайтындығымен түсіндіріледі.

C тілінде орындауға болатын, бірақ C++ тілінде емес, программа мысалы ретінде C++ тілінің түйінді сөздерінің идентификатор ретінде қолданылуы болып табылады (27.3.2 бөлім)

```
int class (int new,int bool);   /* C,   бірақ  C++  емес */
```

Екі тілде де конструкциялық семантикасы сәйкес келіп орындала беретін мысал табу қиындау, бірақ ондай мүмкіндіктер бар.

```
int s=sizeof('a');
/* sizeof(int)  C тілінде 4-ке, ал C++ тілінде 1-ге тең */
```

C тіліндегі `'a'` сияқты тіркестік литерал `int` типінде, ал C++ тілінде – `char` типінде болады. Бірақ `char` типіндегі `ch` айнымалысы үшін екі тілде де `sizeof(ch)==1` шарты орындалады.

Тілдер арасындағы үйлесімділік пен айырмашылыққа қатысты ақпарат онша қызықты емес. C тілінде программалаудың әдейі үйренуге болатындай жетілдірілген тәсілдері жоқ. Сізге `printf()` функциясы көмегімен мәліметтерді шығару ұнауы мүмкін (27.6 бөлім), бірақ бұл бөлімдегі осы функциядан басқа мәліметтер (кейбір әзіл-оспақтарды қоспағанда) құрғақ әрі қарапайым ғана мазмұнда берілген. Оның мақсаты қарапайым: оқырмандарға, қажеттілік туып

қалса, С тіліндегі программаны оқу және жазу мүмкіндігін беру. Ол С тілінде жұмыс істейтін тәжірибелі программалаушыларға қауіптер жайлы ескертулер жасап, ал С++ тілінде жұмыс істейтін программалаушыларға көбінесе күтілмеген жағдай жайлы мәлімет береді. Мұндай қиыншылықтардан аздаған ғана қиындықтармен құтылып кетеді деген үміттеміз.

С++ тілінде жұмыс істейтін программалаушылар ерте ме кеш пе, әйтеуір С тілінде жазылған программамен кездеседі. Керісінше, С тілінде программа жазатын программалаушылар да, көбінесе С++ тілінде жазылған программамен жұмыс істеуге мәжбүр болып жатады. Бұл бөлімде сипатталатындардың басым бөлігі С тілінде жұмыс істейтін программалаушыларға таныс, бірақ бірсыпыра мәліметтер сарапшылар деңгейіне жатқызылуы мүмкін. Мұның себебі түсінікті: барлық адамдар сарапшылар деңгейі туралы бірдей көзқараста болмайды, сондықтан да біз нақты программаларда жиі кездесетін жайттарды сипаттаймыз. Үйлесімділік жайлы туралы ойлар "С тілінің сарапшысының" үстіртін беделін (репутация) арзан жолмен алу тәсілі болуы мүмкін. Бірақ нақты тәжірибе тілдің эзотерикалық ережесін (бұл үйлесімділікке арналған бөлімдерде айтылғандар сияқты) үйренуден емес, тілді практикалық түрде пайдалануға байланысты (мұнда С тілі) жетілетінін есте сақтау керек.

БИБЛИОГРАФИЯ

ISO IEC 9899:1999. *Programming Languages* – C. Бұл кітапта C99 тілі сипатталған, көптеген компиляторлар C89 тілін жүзеге асырады (көбінесе кейбір кеңейтілулерімен қоса).

ISO/IEC 14882:2003-27-01 (2-басылым). Programming Languages – C++. Бұл кітап программалаушының көзқарасы тұрғысынан жазылған, 1997 жылғы нұсқасына ұқсас.

Kernighan, Brian W., and Dennis M. Ritchie. The CProgramming Language. Addison-Wesley, 1988. ISBN 0131103628.

Stroustrup, Bjarne. "Learning Standard C++ as a New Language". C/C++ Users Journal, May 1999.

Stroustrup, Bjarne. "C and C++; Siblings"; "C and C++: A Case for Compatibility"; and "C and C++: Case Studies in Compatibility". The C/C++ Users Journal, July, Aug., and Sept. 2002.

Страуструп мақалаларын оның жеке веб-парақтарынан таба аласыз.

27.1.2 C++ тілінің C тілінде жоқ қасиеттері

C++ тілінің тұрғысынан қарағанда, C тілінде (яғни C89 нұсқасында) көптеген қасиеттер жоқ.

- Кластар және функция-мүшелер
  - C тілінде құрылымдар және ауқымды (глобальді) функциялар қолданылады
- Туынды кластар және виртуалды функциялар
  - C тілінде құрылым, ауқымды функциялар және функцияларға нұсқауыштар қолданылады (27.2.3 бөлім)
- Шаблондар және қойылатын функциялар
  - C тілінде макростар қолданылады (27.8 бөлім)
- Аластамалар
  - C тілінде қателер кодтары, қате қайтарылған мәндер және т.б қолданылады.
- Функциялардың асыра жүктелуі
  - C тілінде әрбір функцияға жеке ат беріледі.
- `new/delete` операторлары
  - C тілінде `malloc()/free()` функциялары және инициалдау мен жоюдың жеке коды қолданылады.
- Сілтемелер
  - C тілінде нұсқауыштар қолданылады
- `const` түйінді сөзі константалық өрнекте болады
  - C тілінде макростар қолданылады
- For нұсқауындағы жариялаулар және жариялаулар нұсқаулар сияқты
  - C тілінде барлық жариялаулар блоктың басында орналасуы керек, ал әрбір анықтаулар топтамасы үшін жаңа блок ашылады
- `bool` типі
  - C тілінде `int` типі қолданылады.
- `static_cast`, `reinterpret_cast` және `const_cast` операторлары
  - C тілінде `static<int>(a)` түріндегі емес, `(int)a` түріндегі келтіру әрекеті қолданылады.
- `// комментарийлер (түсініктемелер)`
  - C тілінде `/*...*/` түсініктемелері қолданылады.

С тілінде көптеген пайдалы программалар жазылған, сондықтан бұл тізім бізге, тілдің ешқандай да қасиеті абсолютті түрде қажет болмайтынын еске салып отыруы керек. Тілдің көптеген мүмкіндіктері – тіпті С тілі қасиеттерінің басым бөлігі – тек программалаушылардың ыңғайлы жұмыс істеуіне арналып жасалған. Ақырында айтарымыз, уақыт, шеберлік және шыдамдылық жеткілікті болып жатса, кез келген программаны ассемблерде жазып шығуға болады. С және С++ тілдері модельдерінің нақты компьютерлерге жақындығының арқасында олар программалаудың көптеген стильдерін пайдаланып жүзеге асыруға мүмкіндік беретініне назар аударыңыз.

Бұл бөлімнің басқа бөліктері осы қасиеттерсіз қалай пайдалы программалар жазуға болатынын түсіндіруге арналады. Біздің С++ тілін қолдану бағытында беретін негізгі кеңестеріміз мыналарға келіп тіреледі:

- С++ тілінің қасиеттері ескерілген программалау стильдеріне ұқсас тәсілдерді С тілінде қарастырылған мүмкіндіктер арқылы жүзеге асыруға тырысыңыз.
- С тілінде программа жазу кезінде оны С++ тілінің ішкі жиыны деп есептеңіз.
- Функция аргументтерін тексеру үшін компилятор ескертулерін пайдаланыңыз.
- Үлкен программалар жазған кезде программалау стилінің стандартқа сәйкестігін бақылап отырыңыз (27.2.2 бөлім).

С және С++ тілдерінің сәйкес еместігіне қатысты көптеген нақтылықтар көнерген, олар техникалық сипатты ғана бейнелейді. Бірақ С тілінде оқып жазу үшін, сіз мұны есте сақтауыңыз міндетті емес.

- С тілінде жоқ С тілінің құралдарын қолданатын болсаңыз, компилятордың өзі оны сіздің есіңізге салады.
- Егер сіз жоғарыда көрсетілген ережелерді ұстанатын болсаңыз, С++ тілімен салыстырғанда, С тілінде басқа мағынасы бір нәрсеге ұрынуыңыз мүмкін емес.

С++ тілінің барлық мүмкіншіліктері болмаған күннің өзінде, С тілінің кейбір құралдары ерекше мәнге ие болады:

- Жиымдар мен нұсқауыштар.
- Макростар.
- `typedef` операторы.
- `sizeof` операторы.
- типтерді келтіру операторлары.

Бұл тарауда сондай құралдарды қолданудың бірнеше мысалдары келтіріледі.

Мені `/*...*/` түріндегі комментарийді теру жалықтырған кезде, мен C++ тіліне одан бұрын шыққан BCPL тілінен мұраланған `//` түсініктемесін енгіздім. `//` түсініктемесі, C99 нұсқасын қосқанда, тілдің көптеген диалектілерінде қабылданған болатын, сондықтан оны ойланбастан, қауіпсіз түрде қолдануға болады. Біздің мысалдарымызда `/*...*/` түріндегі түсініктемені біз C тілінде программа жазып отырғанымызды көрсету үшін ғана қолданамыз. C99 тілінде C++ тілінің кейбір мүмкіндіктері (және де C++ тіліне сәйкес келмейтін кейбір мүмкіндіктері) жүзеге асырылған, бірақ біз C89 нұсқасын ұстанамыз, өйткені ол кеңірек қолданылады.

27.1.3 C тілінің стандартты кітапханасы

Әрине, C++ тілі кітапханасының кластар мен шаблондарға байланысты мүмкіндіктерін C тілінде қолдана алмаймыз. Солардың бірнешеуін көсете кетейік:

- `vector` класы.
- map класы.
- `set` класы.
- `string` класы.
- STL кітапханасының алгоритмдері: мысалы, `sort()`,`find()` және `copy()`.
- `iostream` енгізу-шығару ағымы.
- `regex` класы.

Осыған байланысты C тілінің кітапханасы жиымдарға, нұсқауыштарға және функцияларға негізделген. C тілі стандартты кітапханасының негізгі бөлігіне келесі тақырыптық файлдар жатады.

- `<stdlib.h>`. Ортақ утилиттер (мысалы, `malloc()` және `free()`; 27.4 бөлімді қ.).
- `<stdio.h>`. Енгізу/шығарудың стандартты механизмі; 27.6 бөлімді қ..
- `<string.h>`. C тілі стиліндегі тіркестер және жадымен жұмыс істеу ; 27.5 бөлімді қ..
- `<math.h>`. Жылжымалы нүктелі сандарға арналған операциялар үшін стандартты математикалық функциялар; 24.8 бөлім.
- `<errno.h>`. `<math.h>` тақырыптық файлынан алынған математикалық функциялар қателерінің кодтары; 24.8 бөлім.

- **<limits.h>**. Бүтінсандық типтердің мөлшерлері.24 2 бөлімді қ.
- **<time. h>**. Күнтізбе мен уақыт функциялары; 26.6.1 бөлімді қ.
- **<assert .h>**. Жөндеп түзетуге арналған шарттар (debug assertions); 27.9 бөлімді қ.
- **<ctype.h>**. Символдарды жіктеу; 11.6 бөлімді қ.
- **<stdbool.h>**. Бульдік макростар.

С тілінің стандартты кітапханасының толық сипаттамасын осыған сәйкес оқулықтан, мысалы, K&R кітабынан табуға болады. Мұндағы барлық кітапханаларға (және тақырыптық файлдарға) С++ тілінен де қол жеткізуге болады.

27.2 Функциялар

С тіліндегі функциялармен жұмыс істеудің бірнеше ерекшеліктері бар:

- Берілген аты бар бір ғана функция бола алады.
- Функциялар аргументтерінің типтерін тексеру міндетті емес.
- Сілтемелер болмайды (ендеше, аргументтерді сілтемелермен беру механизмі де жоқ).
- Функция-мүшелері жоқ.
- Орнына қойылатын функциялары жоқ (С99 нұсқасын қоспағанда).
- Функцияларды жариялаудың баламалы синтаксисі бар.
- Бұдан басқаларының бәрінің де С++ тілінен ешқандай айырмасы жоқ. Көрсетілген ерекшеліктерді жекелеп жазып үйренейік.

27.2.1 Функциялар аттарының асыра жүктелуінің болмауы

Келесі мысалды қарастырайық:

```
void print(int);        /* бүтін санды баспаға шығару  */
void print(const char*); /* тіркесті баспаға шығару  */
/* қате! */
```

Екінші жариялау қате болып табылады, өйткені С тілінде жазылған программада бір атпен берілген екі функция болмайды. Сонымен бізге сәйкес келетін екі ат ойлап табу керек.

```
void print_int (int);   /* int бүтін санын баспаға шығару */
void print_string(const char*); /* тіркесті баспаға шығару */
```

Кейде бұл қасиетті артықшылық деп атайды; өйткен сіз енді дұрыс жазылмаған функцияны бүтін санды шығаруда кездейсоқ түрде пайдалана аласыз! Әрине, мұндай аргументтің бізді сендіре алмайтыны айқын, ал асыра жүктелген функцияның болмауы, жалпыланған программалау идеясын іске асыруды қиындатады, өйткені олар аттары бірдей болып келетін семантикалық түрде ұқсас функцияларға қолдануға негізделген.

27.2.2 Функциялар аргументтері типтерін тексеру

Келесі мысалды қарастырайық:

```
int main()
{
    f(2);
}
```

C тілінің компиляторы осындай кодты пайдалануға рұқсат етеді: сіз функцияны қолданғанға дейін оны хабарлауға міндетті емессіз (олай жасауға болады және ол міндетті болса да). `f()` функцияның анықталуы басқа бір жерде болуы да мүмкін. Сонымен тіркес, `f()` функциясы компиляцияның басқа модулінде орналасуы да мүмкін, керісінше жағдайда байланыс редакторы қате кеткені жөнінде хабарлайды.

Өкініштісі, бұл анықтау басқа бір бастапқы файлда келесідей түрде былай бейнеленуі мүмкін:

```
/* other_file.c: */

int f(char* p)
{
    int r = 0;
    while (*p++) r++;
    return r;
}
```

Байланыс редакторы бұл қате жөнінде хабарламайды. Оның орнына, сіз программаны орындау сатысында ғана қате кеткені жайлы хабарлама немесе кездейсоқ бір нәтиже аласыз.

Бұл мәселені қалай шешуге болады? Практикада программалаушылар тақырыптық файлды келісіп отырып қолдануды ұстанады. Егер, сіз шақырған немесе анықтаған барлық функциялар, `#include` директивасының көмегімен программаның сәйкес орнына қойылған тақырыпта хабарланған болса, онда типтерді тексеру механизмі іске қосылады. Бірақ та, көлемді программаларда бұған сенуге болмайды. Осының салдарынан C тілі пайда болған алғашқы

күндерден бастап, көптеген компиляторларында жарияланбаған функциялардың шақырылуы туралы ескертетін опциялар пайда болады: соларды пайдаланыңыз. Бұған қосарымыз, С тілі пайда болған алғашқы күндерден бастап-ақ, типтер қайшылығына байланысты барлық мүмкін болатын мәселелерді анықтай алатын программалар шыға бастады. Әдетте, олар *lint* деп аталады. Соларды С тіліндегі кез келген күрделірек программаларға пайдаланыңыз. Сонда *lint* программаларының С тілін С++ тілінің ішкі жиыны ретінде қолдануға итермелейтінін аңғарасыз. С++ тілін жасауға мұрындық болған байқаулардың бірі *lint* программасы тексере алатын нәрселердің басым бөлігін компилятор да оңай тексере алатындығы (бірақ бәрін емес) болған еді.

Сіз, С тіліндегі функция аргументтерін тексеруді іске қосуды сұрай аласыз. Ол үшін аргументтерінің типтері берілген функцияны хабарлау жеткілікті (дәл С++ тіліндегідей) болып саналады. Мұндай жариялау түрі функция прототипі (function prototype) деп аталады. Дегенмен де, аргументтерді бермейтін жариялаулардан аулақ болу керек; олар функция протипі болып табылмайды және де типтерді тексеру механизмін іске қоспайды.

```
int g(double);     /*  прототип –  С++ тіліндегідей   */
int h();          /* прототип емес — аргументтер
                                  типтері көрсетілмеген */

void my_fct()
{
   g();           /* қате: аргумент жіберілген */
   g("asdf");     /* қате: аргумент типі дұрыс емес */
   g(2);          /* ОК:  2 саны 2.0-ге айналады */
   g(2,3);        /* қате: бір аргумент артық */
   h() ;          /* Компилятор рұқсат етеді!
                                  Нәтижені айта алмаймыз */
   h("asdf");     /* Компилятор рұқсат етеді!
                                  Нәтижені айта алмаймыз */
   h(2);          /* Компилятор рұқсат етеді!
                                  Нәтижені айта алмаймыз */
   h(2,3);        /* Компилятор рұқсат етеді!
                                  Нәтижені айта алмаймыз */
}
```

Функция хабарламасында аргумент типі көрсетілмеген. Бұл функция `g()` ешқандай да аргумент алмайды деген емес; бұл кез келген аргумент тобын қабылдай алады және бұл топ шақырғанда дұрыс болып табылады деген. Тағы да жақсы компилятор бұл мәселе туралы ескертеді, ал *lint* программасы оны ұстап алады дейміз.

C++	C тіліндегі эквиваленті
`void f();  // осы дұрысырақ`	`void f(void);`
`void f(void);`	`void f(void);`
`void f(...);`	`void f();`
`// кез келген аргументті алады`	`/* кез келген аргументті алады */`

Егер көріну аймағында функция прототипі болмаса, аргументтерді түрлендіруді шектейтін арнайы ережелер жиыны бар. Мысалы, **char** және **short** типіндегі айнымалылар int типіндегі айнымалыға түрленеді, ал **float** типіндегі айнымалылар **double** типіндегі айнымалыларға түрленеді. Егер сіз **long** типіндегі айнымалымен не болатынын білгіңіз келсе, C тілінің жақсы бір оқулығын қарап шығыңыз. Біздің ұсынысымыз өте қарапайым: прототиптері жоқ функцияларды шақырмаңыз.

Компилятор қате типтегі аргументті беруді орындайтын болса да, мысалы, **int** типінің параметрінің орнына **char*** типінің параметрі, мұндай аргументтерді пайдалану қатеге әкеліп соқтыратынына назар аударыңыз. Деннис Ритчи айтқандай: "C – бұл типтерді қатаң бақылауы және әлсіз тексеруі бар программалау тілі".

27.2.3 Функцияларды анықтау

Сіз функцияны , нақ C++ тіліндегідей анықтай аласыз. Бұл анықтамалар функция прототипі болып табылады.

```
double square(double d)
{
    return d*d;
}

void ff()
{
    double x = square (2);
    /* OK: 2-ні 2.0-ге ауыстырамыз және шақырамыз  */
    double y = square();    /* аргумент жазылмай кеткен  */
    double y = square("Hello");
    /* қате: аргументтердің дұрыс емес типі */
    double y = square(2,3);  /* қате: аргументтер өте көп */
}
```

Аргументсіз функцияның анықталуы функция прототипі болып табылмайды.

```
void f() { /* бір нәрсе істейді */ }
```

```
void g()
{
    f(2);  /* C тілінде OK; C++ тілінде қате */
}
```

`f()` функциясының аргументтердің кез келген типтегі кез келген санын қабылдай алатынын білдіретін келесі код:

```
void f();   /* аргумент типі көрсетілмеген */
```

бір түрлі түсініксіз болып көрінеді. Осыған жауап ретінде мен жаңа белгілеу енгіздім, онда "ештеңе" деген түсінік тікелей `void` түйінді сөзі (void – "ештеңе" дегенді білдіретін төрт әріптен тұратын сөз) арқылы көрсетіледі.

```
void f(void);  /* ешқандай аргументтерді қабылдамайды */
```

Дегенмен, кейінірек мен өкініп қалдым, өйткені бұл құрылым біртүрлі болып көрінеді және тізбекті түрде аргументтер типін тексергенде, ол артық болып табылады. Мұнан да жаманы, Деннис Ритчи (C тілінің авторы) және Дуг Макилрой (Doug Mcllroy) Bell Labs (Bell Labs Computer Science Research Center; 22.2.5 бөлімін қ.) компаниясындағы компьютерлік ғылымдар зерттеу орталығының модалар заңқойы) бұл шешімді "келеңсіз" деп атады. Өкінішке орай, бұл C тілінде жұмыс істейтін программалаушылар арасында кеңінен тарап кеткен. Дегенмен, оны C++ тіліндегі программаларда қолданбаңыз, онда ол ұсқынсыз ғана емес артық та болып табылады.

C тілінде функцияның Algol-60 тілінің стиліндегі сияқты баламалы анықтамасы бар, онда параметрлер типтері олардың аттарынан бөлек көрсетіледі.

```
int old_style(p,b,x) char* p; char b;
{
   /* . . . */
}
```

Бұл "көне стильдегі" анықтама C++ тілі конструкциясына енгізілген және прототип болып табылмайды. Келісім бойынша, типі жарияланбаған аргумент `int` типіндегі аргумент болып есептеледі. Сонымен, `x` параметрі `oldstyle()` функциясының `int` типіндегі аргументі болып табылады. Біз `old_styl()` функциясын келесідей түрде шақыра аламыз:

```
old_style();  /* OK: аргументтердің барлығы да жазылмаған */
old_etyle{"hello",'a', 17);
/* OK:   барлық аргументтердің типтері дұрыс */
old_style(12,13,14);   /* OK:   12 – дұрыс емес  тип */
/* бірақ old_style() р-ны қолданбауына болады */
```

Компилятор мұндай шақыруларды өткізіп жіберуі тиіс (бірақ біз ол бірінші және екінші аргумент туралы ескертеді деп үміттенеміз).

Біз функциялар аргументтері типтерін тексерудің келесідей ережелерін ұстануды ұсынамыз:

- Функциялар прототиптерін кезегімен қолданыңыз (тақырыптық файлды пайдаланыңыз).
- Компилятордың ескертулер деңгейін аргументтер типтерімен байланысты қателерді ұстап алатындай етіп орнатыңыз.
- Lint (кез келген бір) программасын қолданыңыз.

Нәтижесінде сіз бір мезетте C++ тілінің коды да болатын кодты аласыз.

27.2.4 C++ тіліндегі программадан C тілінде жазылған программаны шақыру және керісінше

Сіз C компиляторының көмегімен компиляциядан өткізілген файлдар мен C++ тілінің компиляторы арқылы компиляциядан өткен файлдар арасында, егер олар осындай мүмкіндікті қарастырған болса, байланыс орната аласыз.

Мысалы, GNU C және GCC компиляторларын пайдалана отырып, C және C++ тілдеріндегі кодтардан туындаған объектілік файлдарды байланыстыра аласыз. Сонымен тіркес, Microsoft C және C++ (MSC++) компиляторларын қолдана отырып, C және C++ тілінің кодынан туындаған объектілік файлдарды да байланыстыра аласыз. Бұл тілдердің тек біреуін ғана қолданғаннан гөрі көбірек кітапхананы пайдалануға мүмкіндік беретін кәдімгі, әрі пайдалы тәжірибе.

C++ тілінде, C тіліне қарағанда, типтерді өте қатаң тексеру қарастырылған. Жекелегенде, C++ тіліне арналған компилятор мен байланыс редакторы `f(int)` және `f(double)` функцияларының үйлесімді анықталғанын және дұрыс қолданатындығын, олар әртүрлі бастапқы файлдарда анықталған болса да, тексереді. C тілінің байланыс редакторы мұндай тексеруді жүргізбейді. C тілінде анықталған функцияны C++ тілінде жазылған программадан шақыру үшін және осы әрекетті керісінше орындау үшін, сіз не істейін деп тұрғаныңызды компиляторға хабарлауыңыз қажет.

```
// C++ тілінің кодынан C тіліндегі функцияны шақыру:
extern "C" double sqrt(double);
// C тілі функциясымен байланыс

void my_c_plus_plus_fct ()
{
     double sr = sqrt(2);
}
```

Негізінде, **extern "C"** өрнегі компиляторға сіздің С тілінің компиляторында қабылданған келісімді қолданатыныңызды хабарлайды. Мұнан басқасы, С++ тілінің тұрғысынан қарағанда, бұл программада бәрі дұрыс көрсетілген. Нақты айтар болсақ, С++ тіліндегі **sqrt(double)** стандартты функциясы, әдетте С тілінің стандартты кітапханасына кіреді.

С++ тілінде жазылған программадан С тілінің кітапханасының функциясын шақыру үшін басқа ештеңе қажет емес. С++ тілі С тілінің байланыс редакторында қабылданған келісімдерге бейімделген.

Сонымен тіркес, біз С тілінде жазылған программадан С++ тілінің функциясын шақыру үшін **extern "C"** өрнегі қолдана аламыз

```
// С тілінің кодынан С++ тіліндегі функцияны шақыру :

extern "C" int call_f(S* p, int i)
{
      return p->f(i);
}
```

Енді С тіліндегі программадан **f()** функция-мүшесін жанамалы түрде шақыруға болады.

```
/* С тіліндегі функциядан С++ тіліндегі функцияны шақыру: */

int call_f(S* p, int i);
struct S* make_S(int,const char*);
void my_c_fct(int i)
{
     /* . . . */
     struct S* p = make_S(x,"foo");
     int x = call_f(p,i);
}
```

Бұл конструкция жұмыс істеуі үшін мұнан былай С++ тілін еске алу міндетті емес.

Мұндай өзара байланыстың пайдасы көрініп тұр: кодты С және С++ тілдерінің араласуымен де да жазуға болады. Жекелеп айтсақ, С++ тіліндегі программалар С тілінде жазылған кітапхананы, ал С тіліндегі программалар С++ тілінде жазылған кітапхананы қолдануына болады. Бұдан басқа көптеген тілдер (әсіресе Fortran) С тілінде жазылған функцияны шақыру интерфейсін қабылдаған және С тілінде жазылған программалардан олардың өз функцияларын шақыруды жүзеге асыруға болады.

Жоғарыда келтірілген мысалдарда біз С және С++ тілдерінде жазылған программалардағы **p** нұсқауышы сілтеме жасап тұрған объектіні олар бірігіп пайдаланады деп санадық. Бұл шарт көптеген объектілер үшін орындалады. Жекелеп айтар болсақ, бізде келесі класс болсын делік:

```
//  C++ тілінде:
class complex {
      double re, im;
public:
      // кәдімгі барлық операциялар
};
```

Сонда объектіге нұсқауышты C тілінде жазылған программаға және керісінше, жібермеуге болады. Тіпті, C тілінде жазылған программадан re және im мүшелеріне келесідей жариялау арқылы қол жеткізе алуға болады:

```
/* C++ тілінде:  */
struct complex {
      double re, im;
      /* ешқандай операция жоқ */
};
```

Кез келген тілде программаларды біріктіру ережелері күрделі болып келеді, ал бірнеше тілде жазылған модульдерді біріктіру ережелерін, тіпті кейде сипаттау да оңай емес. Дегенмен, C және C++ тілінде жазылған функциялар бір-бірімен виртуалдық функцияларсыз-ақ құрамдас типтер объектілерімен және кластармен алмаса алады. Егер кластың виртуалдық функциялары болса, онда оларға оның объектілеріне нұсқауыштарды жіберіп, олармен жұмыс істеуді C++ тілінде жазылған кодқа тапсыруға болады. Бұл ереженің мысалы болып `call_f()` функциясы есептеледі: `f()` функциясы `virtual` бола алады. Сонымен, бұл мысал C тілінде жазылған программадан виртуалдық функцияны шақыру әрекетін көрсетеді.

Құрамдас типтерден басқа, типтерді бірігіп пайдаланудың өте қарапайым және ең қауіпсіз тәсілі болып C және C++ тілдерінің ортақ тақырыптық файлында анықталған `struct` конструкциясы саналады. Бірақ бұл стратегия C++ тілін пайдалану мүмкіндіктерін әжептеуір шектейді, сондықтан біз оны қолдануды ұсынбаймыз.

27.2.5 Функцияларға нұсқауыштар

Егер біз объектіге бағытталған технологияны қолданғымыз келсе, C тілінде не істеуге болады (14.2-14.4 бөлімдерді қ.)? Негізінде, бізге виртуалдық функцияларға қандайда бір балама керек. Көптеген адамдардың ойына, бірінші кезекте, берілген объект қандай фигура түрін көрсететінін суреттейтін "тип өрісі" ("type field") бар құрылымды пайдалану келеді. Мысал қарастырайық:

```
struct Shapel {
    enum Kind { circle, rectangle } kind;
    /* . . . */
};

void draw(struct Shape1* p)
{
    switch (p->kind) {
    case circle:
       /* шеңбер саламыз */
       break;
    case rectangle:
       /* тіктөртбұрыш саламыз */
       break;
    }
}

int f(struct Shape1* pp)
{
    draw(pp);
    /* . . . */
}
```

Бұл тәсіл жұмыс істейді. Бірақ бұған екі кедергі бар.

- Әрбір жалған виртуалдық функция (`draw()` функциясы сияқты) үшін біз `switch` операторының жаңа нұсқаулығын жазуымыз керек.
- Жаңа фигураны қосатын әрбір сәт сайын, біз әрбір жалған виртуалдық функцияны `switch` нұсқауына жаңа `case` бөлімін қоса отырып, түрлендіруіміз керек.

Екінші мәселе аздап жағымсыз сипат алып отыр, өйткені біздің қолданушыларымыз бұл функцияларды өте жиі толықтырып отыруы тиіс болғандықтан, біз жалған виртуалдық функцияларды ешқандай кітапханаға қоса алмаймыз. Мұның ең тиімді баламасы болып функцияларға нұсқауыштарды пайдалану болып табылады.

```
typedef void (*Pfct0)(structShape2*);
typedef void (*Pfctlint)(struct Shape2*,int);

struct Shape2  {
    Pfct0 draw;
    Pfctlint rotate;
    /* . . . */
};
```

```
void draw(struct Shape2* p)
{
    (p->draw)(p);
}
void rotate(struct Shape2* p, int d)
{
    (p->rotate)(p,d);
}
```

Shape2 құрылымын Shape1 құрылымы сияқты қолдануға болады:

```
int f(struct Shape2* pp)
{
    draw(pp);
    /* . . . */
}
```

Аз ғана қосымша жұмыс істей отырып, әрбір объектінің әрбір жалған виртуалдық функцияға нұсқауышты сақтау міндетті болмауына қол жеткізе аламыз. Бұның орнына, функцияға нұсқауыштар жиымына нұсқауышты сақтай аламыз (бұл C++ тіліндегі виртуалдық функцияларды іске асыруға ұсқас). Нақты программаларда осындай сұлбаларды қолдану кезінде негізгі мәселе функцияларға деген барлық нұсқауыштарды дұрыс инициалдау болып табылады.

27.3 Қосалқы тілдік айырмашылықтар

Бұл бөлімде C және C++ тілдерінің арасындағы бірсыпыра айырмашылықтарға мысалдар келтірілген, егер оқырман бірінші рет естіп отырған болса, олар аздап қиыншылық туғызады. Олардың кейбіреуі программалауға әжептеуір әсер етеді, сондықтан оларды тікелей түрде ескеру керек.

27.3.1. `struct` атаулар кеңістігінің дескрипторы

C тілінде құрылым атаулары (онда `class` түйінді сөзі жоқ, тек `struct` сөзі бар) қалған идентификаторлардан бөлек атаулар кеңістігінде орналасады. Сондықтан әрбір құрылым атауы (құрылым дескрипторы – structure tag деп аталатын) алдына `struct` түйінді сөзі жазылуы керек. Мысал қарастырайық.

```
structure tag { int x,y;}
pair p1;/*қате: pair идентификаторы көріну аймағында емес*/
struct pair p2;     /* OK */
int pair = 7;
/* OK: pair құрылымы дескрипторы көріну аймағында жоқ */
```

```
struct pair p3;
/* OK: pair құрылымы дескрипторы int типімен перделенбейді */
pair = 8;
/* OK:  pair идентификаторы int типіндегі санға сілтеме */
/* жасайды  */
```

Бір қызығы, айналмалы әрекет жасап, бұл тәсілді C++ тілінде де жұмыс істетуге болады. Құрылымдарға берілген атауларды айнымалыларға (функцияларға) да беру – C тіліндегі программаларда кеңінен қолданылатын амал, бірақ бұлай істеуге біз қарсымыз.

Егер сіз әрбір құрылым атауы алдында **struct** түйінді сөзін жазғыңыз келмесе, онда **typedef** операторын қолданыңыз (20.5 бөлімді қ.). Келесі идиома кеңінен тараған:

```
typedef struct { intx,y; } pair;
pair p1 = { 1, 2 };
```

Жалпы **typedef** операторы жиі қолданылады және ол программалаушының жаңа типті және оған байланысты операцияларды анықтау мүмкіндігі жоқ. C тіліндегі программаларда өте тиімді болып табылады.

C тілінде қабаттасқан құрылымдар аттары сол өздері кіріп тұрған құрылым атауы орналасқан кеңістік атауында болады. Мысал қарастырайық:

```
struct S {
    struct T {   /*   .   .   .   */  };
    /* ... */
};

struct T x;   /* C тілінде  ОК  (бірақ C+ + тілінде емес )  */
```

C++ тіліндегі программада бұл фрагмент былай жазылар еді:

```
S::T x;         // C++ тілінде   ОК (бірақ  C тілінде емес)
```

C тіліндегі программада, аз ғана мүмкіншілік болса, қабаттасқан құрылымды қолданбауға тырысыңыз: олардың көріну аймағындағы рұқсат беру ережелері көптеген адамдардың ойларына келе қоймайтын деңгейде болады.

27.3.2 Түйінді сөздер

C++ тіліндегі көптеген түйінді сөздер C тілінде де түйінді сөздер болып табылмайды (өйткені C тілі бұларға сәйкес функционалдық мүмкіндіктерді қамтамасыз ете алмайды), сондықтан олар C тіліндегі программаларда идентификатор ретінде қолданыла алады.

C++ тілінің C тілінде жоқ түйінді сөздері					
and	and_eq	asm	bitand	bitor	bool
catch	class	compl	const_cast	delete	dynamic_cast
explicit	export	false	friend	inline	mutable
name-space	new	not	not_eq	operator	or
or_eq	private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try	typeid
typename	using	virtual	wchar_t	xor	xor_eq

Бұл атауларды C тіліндегі программаларда идентификатор ретінде қолданбаңыз, әйтпесе сіздің кодыңыз C++ тілімен үйлеспейтін болады. Егер сіз осы атаулардың бірін тақырыптық файлда қолдансаңыз, онда сіз оны C++ тіліндегі программада қолдана алмайсыз.

C++ тіліндегі кейбір түйінді сөздер C тілінде макростар болып табылады.

C++ тілінің C макростары болатын түйінді сөздері							
and	and_eq	bitand	bitor	bool	compl	false	
not	not_eq	or	or_eq	true	wchar_t	xor	xor_eq

C тілінде бұлар `<iso646.h>` және `<stdbool.h>(bool, true, false)` тақырыптық файлдарында анықталған. Олардың C тіліндегі макрос болып табылатындығын пайдаланбаңыз.

27.3.3 Анықтамалар

C++ тілі C тілімен салыстырғанда, программада анықтамаларға көп көлемде орын береді. Мысал қарастырайық:

```
for(int i=0; i<max; ++i) x[i] = y[i];
//  C тілінде мүмкін емес i айнымалысы анықтамасы

while(struct S* p = next(q)) {
//  C тілінде мүмкін емес p нұсқауышының анықтамасы
    /* ... */
}
```

```
void f(int i)
{
    if (i<0 || max<=i) error("диапазон қатесі");
    int a[max];     // қате:  C тілінде нұсқаудан кейін
                    // жариялауға рұқсат етілмеген
    /* ... */
}
```

C (C89) тілі **for** циклі санауышының инициалдау бөлімінде, шарттарда және блоктағы нұсқаулардан кейін жариялауға жол бермейді. Біз алдыңғы фрагментті былай етіп қайта жазып шығуымыз керек:

```
int i;
for(i = 0; i<max; ++i) x[i] = y[i];

struct S* p;
while(p = next(q)) {
    /* ... */
}

void f(int i)
{
    if (i<0 || max<=i) error("диапазон қатесі");
    {
      int a[max];
      /* ... */
    }
}
```

C тілінде инициалданбаған хабарлама анықтама болып есептеледі; ал C тілінде ол жай хабарлама болып есептеледі, сондықтан оны қосарлап жазуға болады.

```
int x;
int x;
/* C тіліндегі программада x атауымен бір бүтінсандық айнымалыны
   анықтайды немесе хабарлайды; C++ тілінде ол қате болады */
```

C++ тілінде болмыс тек бір рет қана анықталуы керек. Егер осы int типіндегі бірдей атаумен белгіленген екі айнымалы компиляцияның әртүрлі модульдерінде орналасқан болса, жағдай қызықты болар еді.

```
/* x.c  файлында:  */
int x;
/* y.c  файлында:  */
int x;
```

С тіліндегі компилятор да, C++ тіліндегі компилятор да **x.c** немесе **y.c** файлдарынан ешқандай қате таба алмайды. Бірақ егер **x.c** және **y.c** файлдарын C++ тіліндегі файлдар ретінде компиляциядан өтсе, онда байланыс редакторы екі рет анықтауға байланысты қате туралы хабарлама береді. Ал егер **x.c** және **y.c** файлдары С тілінде компиляциядан өтсе, онда байланыс редакторы қате туралы хабарлама бермейді. Және әңгіме **x.c**, **y.c** файлдарында ортақ қолданылатын бір ғана (С тіліндегі ережеге толық сәйкестікке орай) **x** айнымалы туралы екен деп есептейді. Егер программадағы барлық модульдерде ортақ бір ғана ауқымды **x** айнымалысы қолданылатын болса, онда төменде көрсетілгенді істеңіз.

```
/* x.c файлында:  */
int x = 0;    /* анықтау */

/* y.c  файлында:  */
extern int x; /* анықтау емес, хабарлау */
```

Бұдан гөрі тақырыптық файлды қолданған жақсы.

```
/* в файле x.h: */
extern int x; /*  хабарлау, анықтау емес  */

/* x.c файлында: */
#include "x.h"
int x = 0;    /* анықтау */

/* y.c  файлында: */
#include "x.h"
/* x айнымалысының хабарлануы тақырыптық файлда орналасқан */
```

Бәрінен де ең жақсысы: ауқымды айнымалыдан аулақ болыңыз.

27.3.4 С тілі стилінде типтерді келтіру

С тілінде (C++ тілінде де) шағын ғана белгілеуді қолдана отырып, **v** айнымалысын тікелей түрде **T** типіне келтіруге болады.

```
(T)v
```

Бұл "С тілі стиліндегі келтіру" немесе "көне стильдегі келтіру" деп айтылады. Оны мәтін тере алмайтын адамдар (икемділігі үшін) мен еріншектер (өйткені олар **v** айнымалысынан **T** типіндегі айнымалы шығуы үшін не істеу керек екендігін білулері міндетті емес) жақсы көреді. Екінші жағынан, бұл стильге программаны сүйемелдеумен айналысып жүрген программалаушылар өте қарсы, өйткені мұндай

түрлендірулер, практикалық тұрғыдан алғанда, байқалмайды және өзіне назар аудартқызбайды. C++ тіліндегі келтірулер (*жаңа стильдегі келтірулер* – new-style casts немесе *шаблондық стильдегі келтірулер* – template-style casts; A.5.7 бөлімін қ.) жеңіл байқауға болатын типтерді тікелей түрлендіру ісін жүзеге асырады. Сізге оны C тілінде таңдауға болмайды.

```
int* p = (int*)7;    /* reinterpret_cast<int*>(0)
                        биттік комбинациясын көрсетеді */
int x = (int)7.5>    /* static_cast<int>(7.5)
                       типіндегі айнымалыны қысқартады */

typedef struct S1 { /*...*/ } S1;
typedef   struct  S2 { /*...*/ } S2;
S2 a;
const S2 b;
/*  C тілінде инициалданбаған константалар болады */
S1* p = (S1*)&a;  /* reinterpret_cast<S1*>(&a)
                     биттік комбинациясын көрсетеді */
S2* q=(S2*)&b;    /* const: const_cast<S2*>(&b)
                     спецификаторын алып тастайды */
S1* r=(S1*)&a;    /* const спецификаторын алып тастайды
                    және типті өзгертеді: қатеге ұқсас */
```

Біз C тіліндегі программаларда макростарды қолдануды ұсынбаймыз (27.8 бөлім), бірақ жоғарыда сипатталған идеяларды келесідей түрде өрнектеуге болады:

```
#define REINTERPRET_CAST(T,v) ((T)(v))
#define CONST_CAST(T,v) ((T)(v))

S1* p = REINTERPRET_CAST (S1*,&a);
S2* q = CONST_CAST(S2*,&b);
```

Бұл `reinter-pret_cast` және `const_cast` операторларын орындау кезінде типтерді тексеруді қамтамасыз етпейді , бірақ бұл келеңсіз операцияларды көзге түсетін қылады да, программалаушының назарын өзіне аударады.

27.3.5 `void*` типіндегі нұсқауышты түрлендіру

`void*` типіндегі нұсқауышты C тілінде меншіктеу операторының оң бөлігінде және де кез келген типтегі нұсқауышты инициалдау үшін қолдануға болады; C++ тілінде бұлай қолдану мүмкін емес. Мысал қарастырайық:

```
void* alloc(size_t x);   /*  x байттарды ерекшелейді  */

void f(int a)
{
    int* p = alloc(n*sizeof(int));
    /* OK C тілінде;  C++ тілінде қате */
    /* . . . */
}
```

Мұнда **void** типіндегі нұсқауыш **alloc** функциясының нәтижесі ретінде қайтарылады да, жанамалы түрде **int*** типіндегі нұсқауышқа түрленеді. C++ тілінде біз бұл жолды келесідей түрде қайта жаза аламыз:

```
int* p =(int*)alloc(n*sizeof(int));
/* C тілінде және C++ тілінде де OK */
```

Біз мұны C тіліндегі программада да және C++ тіліндегі программада да қолдануға болатын болуы үшін, C тілі стиліндегі типті келтіруді қолдандық (27.3.4 бөлім).

Неге **void*** типін **T*** типіне жанамалы түрде түрлендіруді C++ тілінде жасауға болмайды? Өйткені, мұндай түрлендіру қауіпсіз болып саналмайды.

```
void f()
{
    char i = 0;
    char j = 0;
    char* p = &i;
    void* q = p;
    int* pp = q;
    /* қауіпсіз емес; C тілінде болады, C++ тілінде қате */
     *pp = -1;
    /* компьютер жадына &i адресінен бастап қайта жазамыз */
}
```

Мұнда жадтың қай фрагменті қайта жазылатынын да анық айта алмаймыз: **j** айнымалысы ма, әлде **p** нұсқауышы сілтеме жасап тұрған жады бөлігі ме? **f** функциясын шақыруды басқару үшін пайдаланылған жады бөлігі ме (**f** функциясының стегі)? Мәліметтің қандай түрі қайта жазылса да, **f** функциясын шақыру жағымсыз жағдайға әкеледі.

Назар айдарыңыз, **T*** типіндегі нұсқауышты **void*** типіндегі нұсқауышқа түрлендіру ешқандай қауіпсіз болып табылады, – сіз алдыңғыға ұқсас келеңсіз мысалдар ойлап та таба алмайсыз, – олар C тілінде де және C++ тілінде де рұқсат етіледі.

Өкініштісі, `void*` типін `T*` типіне жанамалы түрде түрлендіру C++ тілінде кеңінен тараған және нақты программаларда C және C++ тілдерінің үйлесімділігінің негізгі мәселесі болып табылады.

27.3.6 Тізбе

C тілінде `int` типін enum типіне келтірмей-ақ, бүтін санды тізбеге меншіктеуге болады. Мысал қарастырайық.

```
enum color { red, blue, green};
int x = green;      /* C және  C++ тілдерінде   OK */
enum color col = 7;  /* C тілінде OK ;  C++ тілінде қате  */
```

Мұның салдарларының бірі болып, C тіліндегі программаларда инкрементация (`++`) және декрементация (`--`) операцияларын тізбе болып саналатын айнымалыларға қолдана алатынымыз болып табылады. Бұл ыңғайлы болуы мүмкін, бірақ сонымен тіркес қауіпсіз де емес.

```
enum color x = blue;
++x;              /*  x айнымалысы green мәніне тең болады;
                                  C++ тіліндегі қате  */
++x;              /*  x айнымалысы 3-ке тең болады,
                                 C++ тілінде қате  */
```

Тізбенің шегінен шығып кету біздің жоспарға кіруі мүмкін, күтпеген жағдайда кездейсоқ түрде болуы да мүмкін.

Назар аударыңыз, құрылым дескрипторларындағы сияқты тізбелер аттары өздерінің жеке атаулар кеңістігінде болады, сондықтан әрбір тізбе атауларын көрсеткен кезде олардың алдына `enum` түйінді сөзін қойып отыру керек.

```
color c2 = blue;   /* C тілінде қате: color айнымалысы көріну
           аймағы ішінде орналаспаған; C++ тілінде OK */
enum color c3 = red; /*  OK */
```

27.3.7 Атаулар кеңістігі

C тілінде атаулар кеңістігі жоқ (C++ тіліндегі сияқты түрде). Сонымен, C тілінде жазылған үлкен программаларда атаулар қайшылығынан (коллизиясынан) аулақ болу үшін не істеу керек? Көбінесе мұндайда префикстер мен суффикстер қолданылады. Мысал қарастырайық:

```
/* bs.h файлында: */
typedef struct bs_string {/*...*/} bs_string;
/* Бьярне тіркесі */
typedef int bs_bool ;       /* Бьярненің бульдік типі */

/* pete.h файлында: */
typedef char* pete_string;          /* Пит тіркесі */.
typedef  char pete_bool    ;        /* Пит бульдік типі */
```

Бұл тәсіл өте кеңінен қолданылады, сондықтан бір немесе екі әріптік префиксті қолдану әдетте жеткіліксіз болады.

27.4 Бос жады аймағы

C тілінде объектілермен жұмыс істейтін **new** және **delete** операторлары жоқ. Бос жады аймағын пайдалану үшін мұнда компьютер жадымен жұмыс істейтін функциялар қолданылады. Ең маңызды функциялар жалпы утилиттердің стандартты **<stdlib.h>** тақырыптық файлында анықталған.

```
void* malloc (size_t sz); /*  sz  байтты бөлу  */
void free(void* p);
/*  p нұсқауышы сілтеме жасап тұрған жады аймағын босату  */
void* calloc(size_t n,size_t sz);
/* нөлдермен инициалдап, n*sz  байтты бөлу  */
void* realloc(void* p,size_t sz);
/* p нұсқауышы сілтеме жасап тұрған жады аймағынан sz байт бөлу */
```

typedef size_t типі – бұл да **<stdlib.h>** тақырыптық файлында анықталған таңбасыз тип.

malloc() функциясы неге **void*** нұсқауышын қайтарады? Себебі ол сіздің компьютер жадында қандай типтегі мәлімет сақтайтыныңызды білмейді. Инициалдау – бұл сіздің проблемаңыз. Мысал қарастырайық:

```
struct Pair {
    const char* p;
    int val;
};

struct Pair p2 = {"apple",78};
struct Pair* pp = (struct Pair*) malloc(sizeof(Pair));
                  /* жады бөлу */
pp->p = "pear";     /* инициалдау */
pp->val = 42;
```

Біз енді мынадай нұсқауды:

```
*pp = {"pear", 42}; /* қате:   C немесе C++98 емес */
```

C тіліндегі программада да, C++ тіліндегі программада да жаза алмаймыз. Бірақ C++ тілінде біз **Pair** құрылымы үшін конструктор анықтап, келесідей нұсқау жаза аламыз:

```
Pair* pp = new Pair("pear", 42);
```

C тілінде (бірақ C++ тілінде емес; 27.3.4 бөлімін қ.) **malloc()** функциясын шақыру алдында типті келтіруді көрсетпеуге болады, бірақ біз мұны істеуді ұсынбаймыз.

```
int* p = malloc(sizeof(int) *n);    /* бұдан аулақ болыңыз */
```

Программаларда келтіруді ескермеу жиі кездеседі, өйткені бұл уақыт үнемдейді және программалаушы программа мәтініне **malloc()** функциясын қолдана отырып **<stdlib.h>** тақырыптық файлын қосуды ұмытқан кезде сирек кететін қатені табуға мүмкіндік береді. Бірақ та, мұнымен бірге жады көлемінің дұрыс есептелгенін көрсететін визуалды маркер жоқ болып кетеді.

```
p = malloc(sizeof(char)*m);
/*  қате болуы мүмкін – m бүтін санға орын жоқ */
```

C++ тілінде жазылған программаларда **malloc()/free()** функцияларын қолданбаңыз; **new/delete** операторлары типті келтіруді талап етпейді, инициалдауды атқарады (конструкторды шақыра отырып), жады аймағын тазартады (деструкторды шақыра отырып), компьютер жадын бөлуге (аластамалар арқылы) байланысты қателерді хабарлайды және жылдамырақ жұмыс істейді. **delete** операторын орындай отырып **malloc()** функциясының көмегімен компьютер жадында орналасқан объектіні өшірмеңіз және де **free()** функциясын шақыра отырып, **new** операторының көмегімен құрылған объектіні өшірмеңіз. Мысал қарастырайық:

```
int* p = new int[200];
// . . .
free(p);     //  қате

X* q = (X*)malloc(n*sizeof(X));
//. . .
delete q;    //  қате
```

Бұл код жұмысқа қабілетті болуы мүмкін, бірақ ол тасымалданатын болып есептелмейді. Бұдан басқа, конструкторы мен деструкторы бар объект үшін, бос жады аймағын басқару кезіндегі C және C++ тілдері стилінің араласуы апатты жағдайға (қатастрофаға) алып келуі мүмкін.

Буферлерді кеңейту үшін көбінесе `realloc()` функциясы қолданылады.

```
int max = 1000;
int count = 0;
int c;
char* p = (char*)malloc(max);
while ((c=getchar())!=EOF) {
/* read: ignore chars on eof line */
    if (count==max- 1) {      /* need to expand buffer */
       max += max;            /* double the buffer size */
       p = (char*)realloc(p,max);
       if (p==0) quit();
    }
    p[count++] = c;
}
```

C тіліндегі енгізу операторлар туралы түсініктемелер 27.6.2 және Б.10.2 бөлімдерінде келтірілген.

`realloc()` функциясы жады көлемін бұрынғы учаскеден бөліп бере алады немесе оның мәліметін жады көлемінің жаңадан бөлінген аймағына ауыстыра алады. `new` операторының көмегімен бөлінген жады аймағында `realloc()` функциясын қолдану жайлы ойламаңыз.

C++ тілінің стандартты кітапханасын қолдана отырып, бұл кодты келесідей түрде қайта жазуға болады:

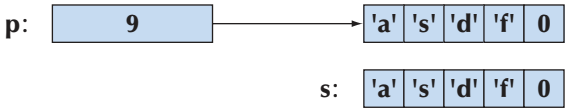
```
vector<char> buf;
char c;
while (cin.get(c)) buf.push_back(c);
```

Енгізу стратегиясы мен жады аймағының бөлінуін толығырақ талқылауды "Learning Standard C++ as a New Language" мақаласынан табуға болады (библиографиялық сілтемелер тізімін 27.1 бөлімнің соңынан қ.)

27.5 C тілі стиліндегі тіркестер

C тіліндегі, символдар тіркесі немесе тіркес (C++ тіліне арналған әдебиетте оны көбінесе *C-тіркес* (C-string) немесе *C тілі стиліндегі тіркес* (C-style)) – бұл нөлмен аяқталатын символдар жиымы. Мысал қарастырайық:

```
char* p = "asdf";
char s[] = "asdf";
```



С тілінде функция-мүшелер жоқ, функцияларды асыра жүктеу мүмкін емес және құрылымға арналған операторды (== сияқты) анықтауға болмайды. Осының арқасында, С тілі стиліндегі тіркестерді басқаруға арналған функциялар тобы (класс мүшелері емес) қажет. Мұндай функциялар С және С++ тілдерінің стандартты кітапханаларындағы <string.h> тақырыптық файлында анықталған.

```
size_t strlen(const char* s); /* символдар санын анықтайды */
char* strcat(char* s1,const char* s2);
/* s2-ні s1-дің соңына көшіреді */
int strcmp(const char* s1,const char* s2);
/* лексикографиялық салыстыру */
char* strcpy(char* s1,const char* s2);
/* s2-ні s1-ге көшіреді */

char* strchr(const char *s,int c);
/* c-ді s-тен іздеп табады */
char* strstr(const char *s1,const char *s2);
/* s2-ні s1-ден іздеп табады */

char* strncpy(char*, const char*, size_t n);
/* n символдарды салыстырады */
char* strncat(char*, const char, size_t n);
/* n символдармен strcat */
int strncmp(const char*, const char*, size_t n);
/* n символдармен strcmp */
```

Бұл тіркестермен жұмыс істеуге арналған функциялардың толық тізімі емес, бірақ ол ең пайдалы және кеңінен қолданылатын функцияларды қамтып отыр. Олардың қолданылуын қысқаша көрсетіп кетейік.

Біз тіркестерді салыстыра аламыз. Теңдікті (==) тексеру операторы нұсқауыш мәндерін салыстырады; strcmp() стандартты кітапхана функциясы С-тіркестерінің мәндерін салыстырады.

```
const char* s1 = "asdf";
const char* s2 = "asdf";
```

```
if (s1==s2){
/* s2 және s2 нұсқауыштары бір ғана жиымға сілтеме жасай ма? */
/*(қалыпты жағдайда бұл қажетті емес) */
}
if (strcmp(s1,s2)==0) {
/* s1 және s2 тіркестері бірдей символдарды сақтай ма? */

}
```

strcmp() функциясы үш түрлі жауап бере алады. Жоғарыда көрсетілген **s1** және **s2** мәндерінде **strcmp(s1,s2)** функциясы (**s1** және **s2**) нөлді қайтарады, бұл толық сәйкестікті (бірдей) білдіреді. Егер **s1** тіркесі **s2** тіркесінің алдында орналасса, онда ол теріс сан қайтарады, ал егер **s1** тіркесі лексикографикалық тәртіпке сәйкес **s2** тіркесінен кейін тұрса, онда ол оң сан қайтарады. *Лексикографикалық* (lexicographical) термині "сөздіктегідей" дегенді білдіреді. Мысал қарастырайық :

```
strcmp("dog","dog")==0
strcmp("ape","dodo")<0
/* "ape" сөздікте "dodo" сөзі алдында тұрады */
strcmp("pig","cow")>0
/* "pig" сөздікте  "cow" сөзінен кейін тұрады */
```

s1==s2 нұсқауышын салыстыру нәтижесі **0**-ге тең болуы міндетті емес (**false**). Барлық тіркестік литералдарды сақтау үшін тілдің жүзеге асырылу механизмі жады аймағының берілген бір ғана көлемін қолдана алады, сондықтан біз **1** (**true**) жауабын ала аламыз. Қалыпты жағдайда **strlen()** функциясы С-тіркестерін салыстыруды жақсы атқарады.

С-тіркесінің ұзындығы **strlen()** функциясы арқылы анықталады:

```
int lgt = strlen(s1);
```

strlen() функциясы символдар ұзындығын, тіркес соңындағы ақырғы нөлді ескермей, есептейтініне назар аударыңыз. Мұнда **strlen(s1) == 4**, ал **"asdf"** тіркесі компьютер жадында бес байт орын алады. Осы аз ғана айырмашылық есептеулердегі көптеген қателердің бастауы болып табылады.

Біз бір ғана С-тіркесін (соңғы нөлді қосып есептегенде) басқа орынға көшіре аламыз.

```
strcpy(s1,s2);   /* символдарды s2-ден  s1-ге көшіреміз */
```

Нәтижелік тіркеске берілген орын бастапқы тіркес символдарының орналасуы үшін жеткілікті көлемде екеніне программалаушының өзі кепілдік беруі керек.

`strcpy()`, `strcat()` және `strcmp()` функцияларының үшінші аргумент болып берілетін параметрі мәні `n` символдан аспайтын мән болатынын ескеретін `strncpy()`, `strncat()` және `strncmp()` функцияларының нұсқасы болып табылады. Мұнда бастапқы тіркестегі символдар саны `n` мәнінен артық болса, онда `strncpy()` функциясы соңғы нөлді көшіре алмайды да, соның салдарынан көшіру нәтижесі С-тіркесімен сәйкес келмейтін болып шығатынына назар аудару керек.

`strchr()` және `strstr()` функциялары өздерінің екінші аргументтерін бірінші аргумент болып табылатын тіркестен іздеп табады да, нұсқауышты соның бірінші символына қайтарады. `find()` функциясы сияқты, олар да символдарды іздеуді сол жақтан оңға қарай орындайды.

Бір қызығы, бұл қарапайым функциялармен көптеген әрекеттер орындауға болады, бірақ соның үстіне олар жай көзге көрінбейтін қателер көзі болатынына назар аудару керек. Қарапайым тапсырма қарастырайық: қолданушының аты мен адресінің араларына `@` символын орналастырып, біріктіріп (конкатенация) жазу керек болсын. `std:string` класының көмегімен мұны былай істеуге болады :

```
string s = id + '@' + addr;
```

С-тіркестерімен жұмыс істейтін стандартты функциялар көмегімен бұл кодты келесідей түрде жазуға болады:

```
char* cat(const char* id, const char* addr)
{
    int sz = strlen(id)+strlen(addr)+2;
    char* res = (char*) malloc(sz);
    strcpy(res,id);
    res[strlen(id)+1] = '@';
    strcpy(res+strlen(id)+2,addr);
    res[sz- 1]=0;
    return res;
}
```

Біз дұрыс жауап алдық па? `cat()` функциясы қайтарған тіркеске `free()` функциясын кім шақырады?

МЫНАНЫ ЖАСАП КӨРІҢІЗ

`cat()` функциясын тесттен өткізіңіз. Біз неге бірінші нұсқауда `2` санын қосамыз? Біз ойланбай, `cat()` функциясында қате жібердік, соны тауып түзетіңіз. Біз кодқа түсініктеме беруді "ұмытып кеттіппіз". Оқырман С-тіркесімен жұмыс істейтін стандартты функцияларды біледі деп есептеп, қажетті түсініктемелер қосыңыз.

27.5.1 C тілі стиліндегі тіркестер және const түйінді сөзі

Келесі мысалды қарастырайық:

```
char* p = "asdf";
p[2] = 'x';
```

C тілінде осылай жазуға болады, ал C++ тілінде – болмайды. C++ тілінде тіркестік литерал константа болып табылады, яғни өзгермейтін шама, сондықтан `p[2] = 'x'` операторын (яғни, бастапқы тіркесті `"asxf"` тіркесіне айналдыруға талпынады) орындау мүмкін емес болып табылады. Өкініштісі, кейбір компиляторлар `p` нұсқауышына мән меншіктеуді жіберіп алады да, ол соңынан мәселеге айналады. Егер сіздің жолыңыз болса, онда программаны орындау сатысында қате көрсетіледі. Мұның орнына былай жазу керек:

```
const char* p = "asdf";  // енді сіз p нұсқауышы арқылы
                         // "asdf" тіркесіне символ жаза алмайсыз
```

Бұл ұсыныс C тіліне де және C++ тіліне де қатысты болып саналады.

C тіліндегі `strchr()` функциясы осы сияқты, бірақ ол одан да қиын анықталатын проблема туындатады. Мысал қарастырайық.

```
char* strchr(const char* s,int c);
/* s константалық тіркесінде c-ді табу (C++ тілінде емес) */

const char aa[] = "asdf";    /* aa – константалар жиымы */
char* q = strchr(aa, 'd');    /*  'd' символын табады   */
*q = 'x';
/* a тіркесіндегі 'd' символын 'x' символына алмастырады */
```

Бұл кодты да C тілінде де, C++ тілінде де пайдалануға болмайды, бірақ C тілінің компиляторлары мұнан қате таба алмайды. Кейде бұл құбылысты *трансмутация* (transmutation) деп атайды: функция константаларды константа емес етіп жібереді де, код туралы дұрыс көзқарасты бұзады.

C++ тілінде бұл проблема `strchr()` стандартты кітапханалық функциясын аздап өзгерту арқылы жариялау жолымен шешіледі.

```
char const* strchr(const char* s,int c);
//  найти символ c в константной строке  s
char* strchr(char* s,int c);  // найти символ c в строке s
```

`strstr()` функциясы да осындай тәсілмен жарияланады.

27.5.2 Байттармен орындалатын операциялар

Баяғы орта ғасырда (1980-ж. басында), `void*` нұсқауышын ойлап тапқанға дейін C (және C++ тілінде де) тілінде жұмыс істейтін программалаушылар байттармен жұмыс істеу үшін тіркестерді пайдаланатын еді. Қазіргі кезде типтерді бақыламай жадымен тікелей жұмыс істеліп жатқаны жайлы қолданушыларды ескерту үшін, компьютер жадымен жұмыс істейтін негізгі стандартты кітапханалық функциялардың `void*` типіндегі параметрлері бар және олар `void*` типіндегі нұсқауыштарды қайтара да алады.

```
/* n байтты s2 тіркесінен s1 тіркесіне
(strcpy функциясы ретінде) көшіреді: */
void* memcpy(void* s1, const void* s2, size_t n);

/* n байтты s2 тіркесінен s1 тіркесіне көшіреді ([sl:sl+n)
диапазоны [s2:s2+n) диапазонының ішіне кіруі мүмкін)*/
void* memmove(void* s1, const void* s2, size_t n);

/* s2 тіркесіндегі n байтты s1 тіркесімен салыстырады
(strcmp функциясы ретінде): */
int memcmp(const void* s1, const void* s2, size_t n);

/* s тіркесінің алғашқы n байты ішінен c (unsigned  char
типіне түрлендірілген) символын табады: */
void* memchr(const void* s, int c, size_t n);

/* c (unsigned char типіне түрлендірілген)
символын көшіреді: */
void* memset(void* s, int c, size_t n);
```

C++ тілінде бұл функцияларды пайдаланбаңыз. Жекелеп айтар болсақ, `memset()` функциясы конструкторлар берген кепілдіктерге әсер етеді.

27.5.3. Мысал: `strcpy()` функциясы

`strcpy()` функциясының анықталуы C (және C++) тілінде орындауға болатын, бірақ келеңсіздігімен белгілі қысқаша жазылатын стиль болып табылады.

```
char* strcpy(char* p,const char* q)
{
    while (*p++ = *q++);
    return p;
}
```

Бұл кодтың неге `q` С-тіркесін `p` С-тіркесіне көшіретінін түсіндіруді жаттығу ретінде оқырмандардың өздерінің орындауына қалдырамыз.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

`strcpy()` функциясын жүзеге асыру дұрыс болып табыла ма? Неге екенін түсіндіріңіз.

Егер сіз өз жауабыңызды дәлелдеп айта алмасаңыз, өзіңізді С тілінде (бірақ сіз басқа программалау тілінде құзыретті болуыңыз мүмкін) программалаушымын деп санауыңызға болмайды. Әрбір тілдің өз ерекшеліктері (идиомалары) бар, ол С тіліне де қатысты болып табылады.

27.5.4 Стиль сұрақтары

Біз біртіндеп стильдің ұзақ әрі жиі келіспеушілік туғызатын сұрақтарына кірісе бастаймыз, бірақ бұлар көбінесе аса маңызды да емес.Біз нұсқауышты былай жариялаймыз:

```
char* p;      //  p  – char типті айнымалыға нұсқауыш
```

Біз төменде көрсетілген стильді қабылдай алмаймыз.

```
char *p;      /* p   —   бұл бір нәрсе,
                 символ алу үшін оны атаусыз етуге болады */
```

Босорынды компилятор тіпті есепке алмайды, бірақ оның программалаушы үшін маңызы бар. Біздің стиль (C++ тілінде программалаушылар арасында жалпы қабылданған) жарияланатын айнымалының типіне көңіл бөледі, ал баламалы стильде (С тілінде программалаушылар арасында жалпы қабылданған) айнымалыны пайдалануға басты назар аударады. Біз бір жолда бірнеше айнымалыны жариялауды ұсынбаймыз.

```
char c,*p,a[177],*f();
/* рұқсат етілген, бірақ бұл шатастыруы мүмкін */
```

Мұндай жариялауларды көбінесе ескі программаларда жиі кездестіруге болады. Бұл жариялаудың орнына оны бір-бірден бірнеше жолдарға орналастырып, қасындағы бос орындарға комментарилер мен инициалдауды жазып қойған дұрыс.

```
char c = 'a'; /*f() функциясы үшін енгізуді аяқтау символы*/
char* p = 0;  /*f() функциясы арқылы оқылған соңғы символ */
```

```
char a[177];        /* енгізу буфері */
char* f();          /* мәліметтерді a буферіне оқу,
        нұсқауышты бірінші оқылған символға қайтарады */
```

Бұған қосарымыз, мағынасы бар аттарды таңдаңыз.

27.6. Енгізу-шығару: stdio тақырыбы

С тілінде **iostream** енгізу-шығару ағымы жоқ, сондықтан біз **<stdio.h>** файлында анықталған С тілінің стандартты енгізу-шығару механизмін пайдаланамыз. С++ тіліндегі **cin** және **cout** енгізу және шығару ағымдарының эквиваленттері болып С тіліндегі **stdin** және **stdout** ағымдары саналады. С тілінің стандартты енгізу-шығару құралдары мен **iostream** ағымдары бір программада қатар қолданыла береді (бір ғана енгізу-шығару ағымдары үшін), бірақ біз мұны қолдануды ұсынбаймыз. Егер сізге бұл механизмдерді қатар қолдану қажет болса, онда жақсы оқулықты пайдаланып, оларды жақсылап талдап шығыңыз (**iosjbase::sync_with_stdio()** функциясына ерекше назар аударыңыз). Б.10 бөлімін де қ.

27.6.1. Шығару

stdio кітапханасының ең танымал әрі пайдалы функциясы болып **printf()** функциясы есептеледі. **printf ()** функциясының негізгі атқаратын қызметі С-тіркесін шығару болып табылады.

```
#include<stdio.h>
void f(const char* p)
{
    printf("Hello, World !\n");
    printf(p);
}
```

Бұл онша қызықты емес. Одан қызығырағы **printf()** функциясы аргументтердің кез келген санын қабылдай алады және қосымша аргументтерді қалай шығаруға болатынын анықтайтын бастапқы басқару тіркесін де ала алады. С тілінде **printf()** функциясын жариялау мынадай түрде болады:

```
int printf(const char* format, ... );
```

Мұндағы көп нүкте (**...**) "және қалған аргументтер де болуы мүмкін" дегенді білдіреді. Біз **printf()** функциясын былай шақыра аламыз:

```
void fl(double d,char* s,int i,char ch)
{
    printf("double %g string %s int %d char %c\n",d,s,i,ch);
}
```

Мұндағы `%g` символдары "Әмбебап форматты пайдаланып, жылжымалы нүктелі санды баспаға шығару" дегенді білдіреді; `%s` символдары "С-тіркесті баспаға шығару" дегенді білдіреді; `%d` символдары "Ондық цифрларды пайдаланып, бүтін санды баспаға шығару" дегенді, ал `%c` символдары "Символды баспаға шығару" дегенді білдіреді. Әрбір осындай формат спецификаторы келесі, бірақ әзірге пайдаланылмай тұрған аргументпен байланысқан, мұнда `%g` экранға `d` айнымалысы мәнін шығарады; `%s` – `s` айнымалысы мәнін, `%d` – `i` айнымалысы мәнін, ал `%c` – `ch` айнымалысы мәнін шығарады. `printf()` функциясы форматтарының толық тізімі Б.10.2 бөлімінде келтірілген.

Өкінішке орай, `printf()` функциясы типтер тұрғысынан қарағанда, қауіпсіз емес. Мысал қарастырайық.

```
char a[] = { 'a', 'b' };                /* соңғы нөл жоқ */
void f2(char* s, int i)
{
    printf("goof %s\n", i);             /* ұсталмаған қате */
    printf("goof %d: %s\n",i);          /* ұсталмаған қате */
    printf("goof %s\n", a);             /* ұсталмаған қате */
}
```

`printf()` функциясының соңғы шақырылу әсері қызықты: ол экранға `a[1]` элементінен кейінгі компьютер жадындағы әрбір байтты нөл кездескенше экранға шығарады. Мұндай экранға шығару әрекеті көптеген символдар санынан тұратыны белгілі шығар.

С және C++ тілдерінің екеуінде де `stdio` кітапханасында сипатталған енгізу-шығару стандартты механизмі бірдей жұмыс істейтініне қарамастан, мұндағы типтерді тексеру кемшілігі біздің `iostream` ағымын ұнатуымыздың негізгі себебі болып табылады. Оның тағы бір себебі `stdio` кітапханасы функцияларының кеңейтілу мүмкіндігі жоқ: `printf()` функциясын біз өзіміз жасаған типіміздегі айнымалылар мәнін шығару үшін қолдана алмаймыз. Ол үшін `iostream` ағымын пайдалануға болады. Мысалы, біздің өзіміз жасаған `%Y` форматы спецификаторын анықтап, сол арқылы `struct Y` құрылымын шығарудың ешқандай да мүмкіндігі жоқ.

`printf()` функциясының бірінші аргументі ретінде файл дескрипторын қабылдайтын пайдалы бір нұсқасы бар:

```
int fprintf(FILE* stream, const char* format,...);
```

Мысал қарастырайық.

```
fprintf(stdout,"Hello, World!\n");
// мынаған ұқсас:  printf("Hello, World!\n");
FILE* ff = fopen("My_file","w");
// My_file файлын мәлімет жазу үшін ашады
fprintf(ff,"Hello, World!\n");
// My_file файлына "Hello, World!\n" сөзін жазу
```

Файлдар дескрипторлары 27.6.3 бөлімінде сипатталған.

27.6.2 Енгізу

Төменде `stdio` кітапханасының ең кең таралған функциялары көрсетілген.

```
int scanf(const char* format, . . . );
/* stdin ағымынан форматты енгізу */
int getchar(void);      /* stdin ағымынан символ енгізу */
int getc(FILE* stream);  /* stream ағымынан символ енгізу */
char* gets(char* s);     /* stdin ағымынан символ енгізу */
```

Символдар тіркесін енгізудің ең оңай жолы — `gets()` функциясын пайдалану. Мысал қарастырайық.

```
char a[12];
gets (a);           /* a символдар жиымына мәліметтерді
                       '\n' символына дейін енгізу */
```

Ешқашанда мұны істемеңіз! `gets()` функциясы уланған деп есептеңіз. `gets()` функциясы өзінің ең жақын "туысы" – `scanf("%s")` функциясымен **бірге** – хакер шабуылдарының шамамен ширегіне нысана (мишень) болып табылады. Ол қауіпсіздікке байланысты көптеген мәселелер туғызады. Жоғарыда көрсетілген ең қарапайым мысалдағыдай, мұнда келесі жаңа тіркеске дейін 11 символдан аспайтын тіркес енгізілетінін біле аласыз ба? Сіз оны біле алмайсыз. Сондықтан `gets()` функциясы компьютер жадын бүлдіруге (буферден кейін орналасқан байттарды) алып келетін шығар, ал жадыны бүлдіру хакер шабуылдарының негізгі құралы болып табылады. Өмірдегі барлық жағдайларға сәйес келетін ең үлкен буфер көлемін анықтай аламын деп есептемеңіз. Мүмкін, енгізу ағымының басқа шетіндегі "субъект" – ол сіз ойлағандай дұрыс программа емес шығар.

`scanf()` функциясы да мәліметтерді оқуды формат арқылы `printf()` функциясы сияқты етіп орындайды. `printf()` функциясы сияқты ол да өте ыңғайлы бола алады.

```
void f()
{
    int i;
    char c;
    double d;
    char* s = (char*)malloc(100);
    /* нұсқауыш тәрізді етіп жіберілген
    айнымалыларға мәлімет оқимыз: */
    scanf("%i %c %g %s", &i, &c, &d, s);
    /* %s спецификаторы алғашқы босорынды өткізіп жібереді
     де, келесі босорында әрекетті тоқтатады */
}
```

scanf() функциясы да **printf()** функциясы сияқты типтер тұрғысынан қарағанда, қауіпсіз болып табылмайды. Форматтық символдар мен аргументтер (барлығы да нұсқауыштар) бір-біріне дәлме-дәл сәйкес келуі керек, әйтпесе программаның орындалуы барысында қызық заттар орын алады. Тағы да мәліметтерді s тіркесіне **%s** спецификаторы арқылы оқу аса толып кетуге де әкелуі мүмкін екендігіне назар аударыңыз. Ешқашанда **gets()** немесе **scanf("%s")** шақыруларын пайдаланбаңыз!

Сонымен, символдарды қалай қауіпсіз түрде енгізу керек? Біз форматтың оқылатын символдар санының шектелуін орнататын **%s** түрін қолдана аламыз. Мысал қарастырайық.

```
char buf[20];
scanf("%19s",buf);
```

Бізге нөлмен аяқталатын жады бөлігі керек болады, сондықтан 19 – бұл **buf** жиымына оқуға болатын символдардың максимал саны. Бірақ бұл тәсіл егер біреу 19 символдан артық мәлімет енгізсе, не істеу керек деген сауалға жауап бере алмайды. Артық символдар енгізу ағымында қалады да, олар тек келесі енгізу кезінде анықталады.

scanf() функциясымен туындайтын мәселелер көбінесе **getchar()** функциясын пайдаланған пайдалы әрі қолайлы екенін көрсетеді. Қалыпты жағдайдағы **getchar()** функциясы арқылы символдарды енгізу әрекеті былай болып көрсетіледі:

```
while((x=getchar())!=EOF) {
      /* . . . */
}
```

stdio кітапханасында сипатталған **EOF** макросы "файл соңы" дегенді білдіреді; 27.4 бөлімді де қ.

C++ тілінің стандартты кітапханасындағы **scanf("%s")** және **gets()** функцияларының баламалары мұндай мәселелерден зардап шекпейді.

```
string s;
cin >> s;              // бір сөз оқимыз
getline(cin,s);        // бір жол оқимыз
```

27.6.3 Файлдар

С (және C++) тілінде файлдарды `fopen()` функциясы арқылы ашып, `fclose()` функциясы арқылы жабуға болады. Бұл функциялар файлдар дескрипторы `FILE` мен `EOF` (файл соңы) макросын көрсету арқылы `<stdio.h>` тақырыптық файлында сипатталған.

```
FILE *fopen(const char* filename, const char* mode);
int fclose(FILE *stream);
```

Негізінде, біз файлды шамамен былай пайдаланамыз:

```
void f(const char* fn, const char* fn2)
{
    FILE* fi = fopen(fn, "r");   /* fn файлын оқу үшін ашу */
    FILE* fo = fopen(fn2, "w");   /* fn файлын жазу үшін ашу */

    if(fi == 0)error("енгізу үшін файл ашу мүмкін емес");
    if(fo == 0)error("шығару үшін файл ашу мүмкін емес");

    /* файлдан stdio кітапханасының енгізу функциясы
    арқылы мәлімет оқу, мысалы, getc() */
    /* файлға stdio кітапханасының шығару функциясы арқылы
    мәлімет жазу, мысалы, fprintf() */

    fclose(fo);
    fclose(fi);
}
```

Есіңізде болсын: С тілінде аластамалар жоқ, сондықтан қате шыққандықтан, файлдың жабылғанын біле алмайсыз.

27.7. Константалар мен макростар

С тілінде константалар статикалық болып саналмайды.

```
const int max = 30;
const int x;
/* инициалданбаған константа: С тілінде ОК (C++ -те қате) */
```

```
void f(int v)
{
    int a1[max];
    /* қате: жиым шекарасы константа емес (C++ тілінде ОК) */
    /* (max сөзінің константалық өрнекте болуы тиіс емес!) */
    int a2[x];  /* қате: жиым шекарасы константа емес */

    switch (v) {
    case 1:
        /* . . . */
        break;
    case max:           /* қате: case бөлімінің белгісі
                           константа емес (C++ тілінде ОК */
        /* . . . */
        break;
    }
}
```

Техникалық себептерге байланысты C тілінде (бірақ C++ емес) константалардың жанамалы түрде компиляцияның басқа модульдерінен алынуына рұқсат етілген.

```
/* file x.c: */
const int x;              /* басқа жерде инициалданған */

/* file xx.c: */
const int x = 7;          /* нағыз анықтау */
```

C++ тілінде әртүрлі файлдарда екі түрлі объект **x** тәрізді бір атаумен аталып тұра береді. **const** түйінді сөзін қолданудың орнына символдық константаларды көрсету үшін программалаушылар C тілінде әдетте макростарды пайдаланады. Мысал қарастырайық.

```
#define MAX 30
void f(int v)
{
    int a1[MAX];       /* ОК */
    switch (v) {
    case 1:
        /* . . . */
        break;
    case MAX:          /* ОК */
        /* . . . */
        break;
    }
}
```

`MAX` макрос аты осы макростың мәні болып саналатын `30` символдарымен алмастырылады; басқаша айтқанда, `a1` жиымы элементтерінің саны `30`-ға тең, ал case операторының екінші бөлімінің белгісі – `30` саны. Жалпы қабылданған келісім бойынша `MAX` макросының аты тек бас әріптерден тұрады. Бұл макростар арқылы шығатын қателер санын азайтуға мүмкіндік береді.

27.8. Макростар

Макростардан сақтаныңыз: С тілінде макростарды қолданбай кетуге тұрарлық нағыз әрі тиімді тәсілдер жоқ, бірақ оларды қолданудың қосалқы әсерлері бар, өйткені олар С және С++ тілдерінде қабылданған типтер мен көріну аймағын шешудің қалыпты ережелеріне бағынбайды. Макростар — бұл мәтін алмастыру түрі. Бұған қосымша тағы А.17.2 бөлімді қараңыз.

Макростардан толығынан бас тартпай, оларға байланысты әлеуеттік мәселелерден қалай қорғануға болады (және С++ тілінде қарастырылған баламаларға қарамай)?

- Барлық макростарға бас әріптерден тұратын атаулар беріңіз: `ALL_CAPS`.
- Макрос емес объектілерге тек кіші әріптерден тұратын аттар мешіктеңіз.
- Макростарға ешқашанда қысқа әрі "нұсқа" аттар бермеңіз, мысалы, `max` немесе `min` сияқты.
- Басқа программалаушылар да осы қарапайым және жалпыға белгілі ережелерді сақтайды деген сенімде болыңыз.

Негізінен макростар келесідей жағдайларда қолданылады:

- "константаларды" анықтау;
- функцияларды еске түсіретін конструкцияларды анықтау;
- синтаксисті жақсарту;
- шартты компиляцияны басқару.

Мұнан басқа, макростар қолданылатын көптеген мысалдар бар, бірақ олар практикада аз кездесетін жағдайларды қамтиды.

Біз макростар өте жиі қолданылады деп санаймыз, бірақ С тіліндегі программаларда оларға сәйкес келетін толыққанды баламалар жоқ. Оларды, тіпті С++ тіліндегі программаларда да айналып өту өте қиын (әсіресе, егер сізге өте ескі компиляторларды қолданатын немесе қалыптан тыс шектеулері бар платформаларда орындалатын программалар жазу қажет болғанда).

Біз төменде көрсетілетін тәсілдерді "лас трюктер" деп санайтын оқырмандардан кешірім сұраймыз, олар туралы дені дұрыс қоғамда әңгіме айту қажет емес

деп саналады. Бірақ біз программалау нақты жағдайларды есепке алуы тиіс деп ойлаймыз және макростарды осындағы есептерде (өте қарапайым) дұрыс қолдану мен дұрыс емес қолдану мысалдары жаңа үйреніп жүргендер үшін көптеген қиналатын сәттерін үнемдейді деп санаймыз. Макростарды білмеу жақсылық әкелмейді.

27.8.1 Функцияларға ұқсас макростар

Функцияны еске түсіретін қалыпты жағдайда кездесетін макросты қарастырайық.

```
#define MAX(x,y)((x)>=(y)?(x):(y))
```

Біз **MAX** атауында бас әріптерді оны көптеген **max** (әртүрлі программаларда) атты функциялардан ажыратып алу үшін пайдаланамыз. Әрине, бұл макростың функциядан айырмашылықтары көп: оның аргументтер типі жоқ, ішкі тұлғасы жоқ, **return** нұсқауы жоқ, т.с.с. Неге мұнда жақшалар көп? Келесі кодты талдап көрейік:

```
int aa = MAX(1,2);
double dd = MAX(aa++,2);
char cc = MAX(dd,aa)+2;
```

Ол мынадай программа фрагментіне айналады:

```
int aa = ((1)>=( 2)?(1):(2));
double dd = ((aa++)>=(2)?( aa++):(2));
char cc = ((dd)>=(aa)?(dd):(aa))+2;
```

Егер осындағы жақшаларды алып тастасақ, соңғы жол мынадай болар еді:

```
char cc = dd>=aa?dd:aa+2;
```

Басқаша айтқанда, **cc** айнымалысы макрос анықталуына сәйкес, сіз күтпеген басқа мәнді жеңіл алуына болатын еді. Макросты анықтай отырып, өрнекке кіретін әрбір аргументті жақшаға алуды ұмытпаңыз.

Басқа жағынан, жақшалар барлық кездерде бізді алмастырудың екінші нұсқасынан құтқара алмайды. **x** макросының параметріне **aa++** мәні меншіктелген болатын, ал **x** айнымалысы **max** макросында екі рет қолданылатындықтан, **a** айнымалысы екі рет инкременттелуі мүмкін. Макросқа қосалқы әсерлері бар аргументтер жібермеңіз.

Бір "данышпан" макросты келесідей түрде анықтап, оны кең қолданылатын

тақырыптық файлға орналастырған. Өкінішке орай, ол оны **MAX** деп емес, **max** деп атаған, сондықтан C++ тілінің стандартты тақырыбында келесі функция жарияланғанда,

```
template<class T> inline T max(T a,T b) { return a<b?b:a; }
```

max атауы **T a** және **T b** аргументтерімен алмастырылғанда, компилятор келесі жолды көреді:

```
template<class T> inline T ((T a)>=( T b)?( T a):( T b))
                           { return a<b?b:a; }
```

Компилятор беретін қате туралы хабарламалар қызықты, бірақ олардан онша көп ақпарат ала алмайсыз. Қауіпті жағдайларда, макросты анықтаудан бас тарта аласыз.

```
#undef max
```

Қуанышқа орай, бұл макрос үлкен келеңсіздіктерге алып келген жоқ. Дегенмен, кең қолданылатын тақырыптық файлдарда он мыңдаған макростар болады; сіз хаос жасамай, олардың бәрінен бас тарта алмайсыз.

Макростардың барлық параметрлері өрнек түрінде пайдаланылмайды. Келесі мысалды қарастырайық:

```
#define ALLOC(T,n) ((T*)malloc(sizeof(T)*n))
```

Бұл бөлінетін жадының қажет етілетін типі мен **sizeof** операторын қолдану арасындағы үйлесімдіктен туындайтын қателерді болдырмауға көмектесетін өте пайдалы шынайы мысал болып табылады.

```
double* p=malloc(sizeof(int)*10); /* қателеге ұқсас сияқты */
```

Өкінішке орай, компьютер жадында орын жоқ екенін анықтауға мүмкіндік беретін макрос жазу оңай іс емес. Егер біз программаның ішкі бір жерінде сәйкестікті бұзбай, **error_var** және **error()** функциясын анықтаған болсақ, мұны жасауға болар еді.

```
#define ALLOC(T,n) (error_var = (T*)malloc(sizeof(T)*n), \
          (error_var==0)\
          ?(error("жады бөлуге рұқсат жоқ"),0)\
          :error_var)
```

\ символымен аяқталатын жолдарда қателер жоқ; бұл тек қана макросты жариялауды бірнеше жолға бөліп жазу тәсілі. Біз C++ тілінде программалар жазған кезде, **new** операторын қолданғанды дұрыс көреміз.

27.8.2 Макростар синтаксисі

Бастапқы кодты сізге көрнекі түрге келтіретін макросты анықтауға болады. Мысал қарастырайық.

```
#define forever for(;;)
#define CASE break; case
#define begin {
#define end }
```

Біз бұған қатаң қарсылық көрсетеміз. *Көптеген* адамдар осындай заттар жасауға тырысқан болатын. Олар (және осындай программаларды сүйемелдеуге тура келген адамдар) келесідей қорытындыларға келді.

- Жақсы синтаксис деп нені есептеу керек деген сіздің көзқарасыңызды көптеген адамдар қолдамайды.
- Жақсартылған синтаксис стандартты емес және күтілмеген де болып табылады; қалған адамдар жолдан жаңылысады.
- Жақсартылған синтаксисті пайдалану түсініксіз компиляция қателерін шақыруы мүмкін.
- Сіздің көз алдыңыздағы программа мәтіні компилятор көріп тұрған мәтінмен бірдей емес, сондықтан компилятор сіздің емес, өз сөздік қорын пайдаланып, қателер туралы мәлімет береді.

Өз кодыңыздың сыртқы түрін жақсарту үшін синтаксистік макростар жазбаңыз. Сіз және сіздің жақын достарыңыз оны тамаша деп санауы мүмкін, бірақ ірі программалаушылар қауымдастығында сіз өте аз ғана қолдауға ие боласыз, сондықтан біреулерге сіздің кодыңызды (егер ол сол сәтке дейін қолдануда болса) қайта жазуға тура келеді.

27.8.3 Шартты компиляция

Сіздің тақырыптық файлыңыздың екі нұсқасы бар деп есептеңіз, мысалы, біреуі – Linux операциялық жүйесі үшін, ал екіншісі – Windows операциялық жүйесі үшін болсын. Сіздің программыңызда дұрыс нұсқаны қалай таңдау керек? Бұл есептің жалпы қолданыстағы шешімі мынадай болады:

```
#ifdef WINDOWS
     #include "my_windows_header.h"
#else
     #include "my_linux_header.h"
#endif
```

Енді егер біреу, компилятор бұл кодты көргенше, **WINDOWS** нұсқасын анықтаған болса, онда келесі код орындалады:

```
#include "my_windows_header.h"
```

Қарсы жағдайда басқа тақырыптық файл іске қосылады

```
#include "my_linux_header.h"
```

#ifdef WINDOWS директивасын **WINDOWS** макросының қандай екені қызықтырмайды; ол тек жай ғана ол бұрын анықталды ма, соны тексереді.

Ірі жүйелердің көбісінде (операциялық жүйелердің барлық нұсқаларын қоса есептегенде) макростар бар, сондықтан сіз оларды тексере аласыз. Мысалы, сіздің программаңыз компиляциядан қалай өткенін: ол C++ тіліндегі программа ретінде компиляциядан өтті ме немесе C тіліндегі программа ретінде өтті ме, соны тексере аласыз.

```
#ifdef__cplusplus
    // C++ тілінде
#else
    /* C тілінде */
#endif
```

Іске қосу сақшысы (include guard) деп жиі аталатын осыған ұқсас конструкция тақырыптық файлды қайта іске қосуды болдырмас үшін қолданылады.

```
/* my_windows_header.h: */
#ifndef MY_WINDOWS_HEADER
#define MY_WINDOWS_HEADER
       /* тақырыптық файл туралы ақпарат */
#endif
```

#ifndef директивасы бір нәрсе бұрын анықталды ма, соны тексереді; мысалы, **#ifndef** **#ifdef** директивасына қарама-қарсы болады. Логикалық тұрғыдан қарағанда, бастапқы файлды бақылау үшін қолданылатын бұл макростардың бастапқы кодты толықтыру үшін қолданылатын макростардан әжептеуір айырмашылықтары бар. Олар тек өз функцияларын орындау үшін бірдей базалық механизмдерді қолданады.

27.9. Мысал: интрузивтік контейнерлер

C++ тілінің стандартты кітапханасындағы **vector** және **map** сияқты контейнерлер интрузивтік емес болып табылады; басқаша айтқанда, олар өздерінің элементтері сияқты пайдаланылған мәліметтер типтері жайлы ақпаратты талап етпейді. Бұл оларды практика жүзіндегі барлық типтер үшін жалпылауға мүмкіндік береді (құрамдас, әрі қолданушы жасаған типтер үшін де), өйткені бұл типтерге көшіру операциясын қолдануға болады. Контейнерлердің басқа бір түрлері де бар, олар C және C++ тілдерінде кең таралған *интрузивтік контейнерлер* (intrusive container). Құрылымдарды, нұсқауыштарды және бос жады аймағын пайдалануды көрсету үшін интрузивті емес тізімді қолданамыз.

Тоғыз операциясы бар қос байланысты тізімді анықтайық.

```
void init(struct List* lst); /* lst-ті бос етіп инициалдаймыз */
struct List* create();
/* бос жады аймағында жаңа бос тізім жасайды */
void clear(struct List* lst);
/* lst тізімінің барлық элементтерін жояды */
void destroy(struct List* lst);
/* lst тізімінің барлық элементтерін жояды,
                сонан соң lst-тің өзін жояды */

void push_back(struct List* lst, struct Link* p);
/* lst тізімінің соңына p элементін қосады */
void push_front(struct List*, struct Link* p);
/* lst тізімінің басына p элементін қосады */

/* lst тізімінің p элементі алдына q элементін кірістіреді: */
void insert(struct List* lst, struct Link* p, struct Link* q);
struct Link* erase(struct List* lst, struct Link* p);
/* lst тізімінен p элементін жояды  */

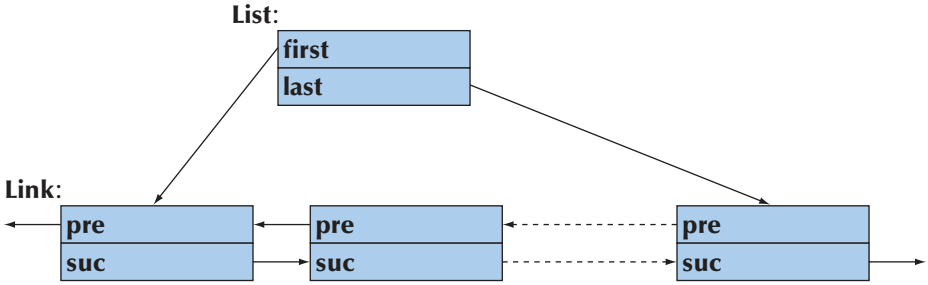
/* p түйініне n түйін жетпей немесе одан кейін n түйін
   өткен соң орналасқан элементті қайтарады: */
struct Link* advance(struct Link* p, int n);
```

Біз бұл операцияларды оларды қолданушыларға тек **List*** және **Link*s** нұсқауыштарын қолдану жеткілікті болатындай етіп анықтағымыз келеді. Бұл осы функцияларды жүзеге асыруды, оны қолданушылардың жұмысына әсер етпей, толығынан өзгертуге болады дегенді білдіреді. Әрине, мұндағы аттарды таңдау STL кітапханасының әсерімен болғаны білініп тұр. **List** және **Link** құрылымдарын күтілгендей қарапайым түрде анықтауға болады.

```
struct List {
      struct Link* first;
      struct Link* last;
};

struct Link {        /* қос байланысты тізімнің түйіні */
      struct Link* pre;
      struct Link* suc;
};
```

`List` контейнерінің графикалық бейнесін келтірейік:



Біздің мақсатымызға қиын-қыстау тәсілдер немесе алгоритмдерді көрсету кірмейді, сондықтан олардың бір де бірі суретте көрсетілмеген. Дегенмен, біздің түйіндерде (тізімдер элементтерінде) сақталған мәліметтер жайлы айтпай отырғанымызға назар аударыңыз.

Бұл құрылымның функция-мүшесіне көз сала отырып, біз соларға ұқсас бір нәрсе жасағанымызды `Link` және `List` абстрактылық кластарын анықтай отырып, көріп тұрмыз. Түйіндерде сақталуға тиіс мәліметтер кейінірек беріледі. `Link*` және `List*` нұсқауыштарын кейде мөлдір емес типтер (opaque types) деп атайды; басқаша айтқанда, `Link*` және `List*` нұсқауыштарын өз функцияларымызға бере отырып, біз `Link*` және `List*` құрылымдарының ішкі құрылуы жайлы ешнәрсе білмей отырып-ақ, `List` контейнерінің элементтерімен әрекеттер жасауға мүмкіндік аламыз.

`List` құрылымының функцияларын жүзеге асыру үшін алдымен кейбір стандартты кітапханалық тақырыптарды іске қосамыз.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
```

С тілінде атаулар кеңістігі жоқ, сондықтан декларациялар туралы немесе `using` директивалары жайлы тынышсызданудың қажеті жоқ. Екінші жағынан, біз өте

қысқа және өте танымал атаулар (**Link**, **insert**, **init** және т.б.) жайлы ойлануымыз керек, сондықтан мұндай функциялар жиынын ойын программаларынан тыс пайдалануға болмайды.

Инициалдау қарапайым орындалады, бірақ **assert()** функциясының пайдаланылуына назар аударыңыз.

```
void init(struct List* lst)
/* *p-ны бос тізіммен инициалдаймыз */
{
      assert(lst);
      lst->first = lst->last = 0;
}
```

Біз программаны орындау барысында тізімдерге дұрыс жасалмаған нұсқауыштарға байланысты қателерді өңдеумен байланыспайтын болып шештік. Егер тізімге нұсқауыш нөлге тең болатын болса, **assert()** макросын пайдалана отырып, біз жай ғана жүйелік қате шыққаны жайлы хабарлама аламыз (программа орындалуы барысында). Егер **assert()** макросының аргументі ретінде берілген шарт бұзылатын болса, бұл жүйелік қате бізге жай ғана файл аты мен жол нөмірін береді; **assert()** — бұл **<assert.h>** тақырыптық файлында анықталған макрос**,** ал оны тексеру тек түзетіп жөндеу (**debug**) режимінде орындалады. Аластамалар болмаған кезде дұрыс жазылмаған нұсқауыштармен не істеу керек екенін түсіну оңай емес.

create() функциясы бос жады аймағының **List** тізімін жасайды. Ол конструктор (**init()** функциясы инициалдауды орындайды) мен **new** операторының (**malloc()** функциясы жады бөледі) комбинациясын еске салады.

```
struct List* create()           /* бос тізім жасайды */
{
   struct List* lst =
              (struct List*)malloc(sizeof(struct List*));
   init(lst);
   return lst;
}
```

clear() функциясы барлық түйіндер құрылған және бос жады аймағында орналасқан деп есептейді де, оларды сол жақтан **free()** функциясы арқылы өшіреді.

```
void clear(struct List* lst)
/* lst тізімінің барлық элементтерін жояды */
{
    assert(lst);
    {
      struct Link* curr = lst->first;
      while(curr) {
          struct Link* next = curr->suc;
          free(curr);
          curr = next;
      }
      lst->first = lst->last = 0;
    }
}
```

Link класының **sue** мүшесін пайдаланатын тәсілге назар аударыңыз, біз солар арқылы тізімді айналып өтеміз. Біз объект мүшесін **free()** функциясы арқылы өшірген соң, оған қауіпсіз түрде қол жеткізе алмаймыз, сондықтан **next** айнымалысын енгіздік, сол арқылы **List** контейнерінде өз орнымыз (позициямыз) туралы ақпаратты сақтаймыз да, сонымен қатар **free()** функциясының көмегімен **Link** класының объектілерін өшіреміз.

Егер **Link** құрылымының барлық объектілері бос жады аймағында орналаспаса, **clear()** функциясын шақырмаған дұрыс, әйтпесе ол жады аймағындағы мәліметтерді бұза бастайды.

destroy() функциясы негізінде, **create()** функциясына қарама-қарсы жұмыс істейді, яғни ол деструктор мен **delete** операторының араласуынан шыққан функция.

```
void destroy(struct List* lst) /* lst тізімінің барлық
элементтерін жояды; сонан соң lst тізімінің өзін де жояды */
{
      assert(lst);
      clear(lst);
      free(lst);
}
```

Жады аймағын тазалау функциясын (деструкторды) шақыру алдында біз тізім түйіндері түрінде көрсетілген элементтер жайлы ешқандай пікір білдірмейтінімізге назар аударыңыз. Бұл сұлба C++ тілі әдістерінің толыққанды имитациясы болып саналамайды – ол бұған арналмаған.

push_back() функциясы — **Link** түйінін тізім соңына қосу – айқын орындалатын әрекет:

```
void push_back(struct List* lst,struct Link* p)
/* p элементін lst тізімінің соңына қосады */
{
    assert(lst);
       {

     struct Link* last = lst->last;
     if (last) {
        last->suc = p;
        /* p түйінін last түйінінен кейін қосады */
        p->pre = last;
     }
     else {
        lst->first = p;  /* p – бірінші элемент */
        p->pre = 0;
     }
     lst->last = p;      /* p – жаңа соңғы элемент */
     p->suc = 0;
    }
}
```

Бірнеше тіктөртбұрыштар мен бағыттауыш тілсызықтардан тұратын сұлбаны сызбай тұрып, бұл кодты толығымен жазу қиын болар еді. Біздің **p** аргументі нөлге тең болатын нұсқаны қарастыруды ұмытып кеткенімізге назар салыңыз. Түйінге нұсқауыш орнына нөлді берсеңіз, сіздің программаңыз ақау (қате) береді. Бұл кодты тіпті дұрыс емес деп санауға болмайды, бірақ ол өндірістік стандарттарға сәйкес келмейді. Оның мақсаты – жалпы қабылданған пайдалы әдістерді (және де кәдімгі кемшіліктер мен қателерді) көрсету.

Erase() функциясын келесідей түрде жазуға болады:

```
struct Link* erase(struct List* lst, struct Link* p)
/*
   lst тізімінен p түйінін өшіру;
   нұсқауышты p түйінінен кейін орналасқан түйінге қайтарады
*/
{
    assert(lst);
    if (p==0) return 0;   /* erase(0)-ді шақыру үшін ОК */

    if (p == lst->first) {
        if (p->suc) {
            lst->first = p->suc;
            /* келесісі бірінші болып шығады */
            p->suc->pre = 0;
            return p->suc;
        }
```

```
        else  {
            lst->first = lst->last = 0;
            /* тізім бос болып шығады */
            return 0;
      }
    }
    else if (p == lst->last) {
        if (p->pre) {
            lst->last = p->pre;
            /* алдыңғысы соңғысы болып шығады */
            p->pre->suc = 0;
        }
        else {
            lst->first = lst->last = 0;
            /* тізім бос болып шығады */
            return 0;
        }
    }
    else {
        p->suc->pre = p->pre;
        p->pre->suc = p->suc;
        return p->suc;
    }
}
```

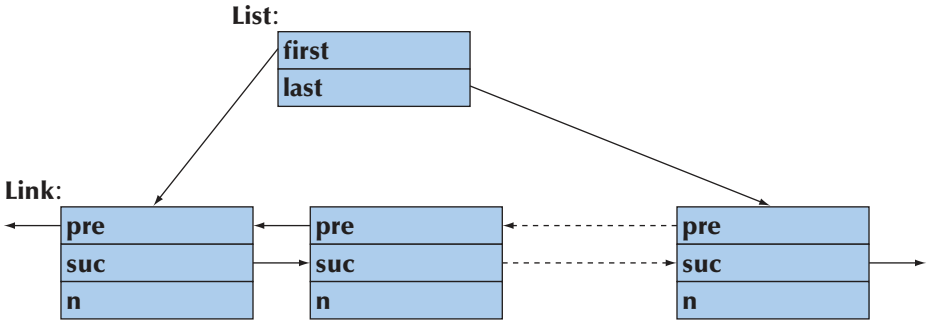
Қалған функцияларды оқырмандар жаттығу ретінде жаза алады, өйткені біздің (өте қарапайым) тест үшін олар керек емес. Дегенмен, біз енді бұл жобаның негізгі жұмбағын шешуіміз керек: тізім элементтеріндегі мәліметтер қайда орналасқан? C-тіркесі түрінде бейнеленген атаулардың қарапайым тізімін қалай жүзеге асыруға болар екен. Келесі мысалды қарастырайық:

```
struct Name {
    struct Link lnk;
    /* Link құрылымы оның операцияларын орындау үшін керек */
    char* p;    /* атаулар тіркесі  */
};
```

Бұған дейін бәрі де жақсы болған, дегенмен **Link** мүшесін біздің қалай пайдалана алатынымыз жұмбақ болып қалып отыр. Бірақ List құрылымының **Link** түйіндерін бос жады аймағында сақтайтынын білгендіктен, біз **Name** құрылымының объектілерін бос жады аймағында құратын функция жазып шықтық:

```
struct Name* make_name(char* n)
{
   structName*p=(structName*)malloc(sizeof(structName));
   p->p = n;
   return p;
}
```

Мұндағы жағдайды келесідей түрде бейнелеуге болады:



Осы құрылымдарды пайдаланып көрейік:

```
int main()
{
   struct List names;          /* тізім құрады */
   struct List* curr;
   init(&names);

   /* бірнеше Names объектісін құрып, оларды тізімге қосамыз:*/
   push_back(&names,(struct Link*)make_name("Norah"));
   push_back(&names,(struct Link*)
                            make_name("Annemarie"));
   push_back(&names,(struct Link*)make_name("Kris"));

   /* екінші атты өшіреміз (1 индексімен): */
   erase(&names,advance(names.first,1));
   curr = names.first;      /* барлық аттарды жазамыз */
   int count = 0;
   for (; curr!=0; curr=curr->suc) {
      count++;
      printf("element %d: %s\n", count,
                                  ((struct Name*)curr)->p);
   }
}
```

Сонымен, біз құлық жасадық. Біз `Name*` типіндегі нұсқауышпен `Link*` типіндегі нұсқауыш сияқты жұмыс істеу үшін типтерді келтіру әрекетін қолдандық. Осының салдарынан қолданушы Link кітапханалық құрылымы туралы білетін болады. Дегенмен кітапхана `Name` қолданбалы типі жайлы білмейді. Бұлай жасауға бола ма? Иә, болады: С тілінде (С++ тілінде де) құрылымға нұсқауышты оның бірінші элементіне нұсқауыш ретінде және бұған керісінше әрекетті де жасауға болады.

Әрине, бұл мысалды С++ тілінің компиляторы арқылы компиляциядан өткізуге болады.

МЫНАНЫ ЖАСАП КӨРІҢІЗ

С++ тілінде жұмыс істейтін программалаушылар, С тілінде жұмыс істейтін программалаушылармен сөйлескенде былай деп қайталайды: "Сенің жасағаныңның барлығын мен жақсырақ етіп жасай аламын!" Сонымен, С++ тіліндегі `List` интрузивтік контейнерінің мысалын көшіріп алыңыз, мұны қысқа және қарапайым түрде программа жұмысын аялдатпай немесе объектілерді ұлғайтпай жасауға болатынын көрсетіңіз.

ТАПСЫРМА

1. С тілінде `"Hello, World!"` программасын жазып, оны компиляциядан өткізіп, орындап шығыңыз.
2. `"Hello"` және `"World!"` тіркестерін сәйкесінше сақтайтын екі айнымалыны анықтаңыз; олардың орталарына босорын таңбасын қосып біріктіріңіз де (конкатенация операциясын), `Hello, World!` тіркесі түрінде шығарыңыз.
3. С тілінде `char*` типіндегі `p` параметрін және `int` типіндегі `x` параметрін алатын функцияны анықтаңыз, баспаға солардың мәндерін келесі форматта: `p is "foo" and x is 7` шығарыңыз. Осы фунцияны бірнеше аргументтер үшін шақырып орындаңыз.

БАҚЫЛАУ СҰРАҚТАРЫ

Келесі сұрақтарда ISO С89 стандарты орындалады деп есептеледі.

1. С++ тілі С тілінің ішкі жиыны болып табыла ма?
2. С тілін кім ойлап шығарды?
3. С тілінен жоғары беделді оқулықты атаңыз.
4. С және С++ тілдері қай мекемеде жасалып шыққан еді?
5. С++ тілі неге С тілімен үйлесімді (өте жақын) болып келеді?

6. C++ тілі неге C тілімен өте жақын үйлеседі?
7. C++ тілінің C тілінде жоқ он ерекшелігін атап өтіңіз.
8. C және C++ тілдері қандай ұйымға "жатады"?
9. 9. C++ тілі стандартты кітапханасының C тілінде қолданылмайтын алты компонентін тізіп көрсетіңіз.
10. C тілінің стандартты кітапханасының қандай компоненттерін C++ тілінде қолдануға болады?
11. C тілінде функциялар аргументтерінің типтерін тексеруді қалай қамтамасыз етуге болады?
12. C++ тілінің функциямен байланысты қандай қасиеттері C тілінде жоқ? Кем дегенде, үш қасиетін көрсетіңіз. Мысалдар келтіріңіз.
13. C++ тіліндегі программадан C тілінде жазылған функцияны қалай шақыруға болады?
14. C тіліндегі программадан C++ тілінде жазылған функцияны қалай шақыруға болады?
15. C және C++ тілдерінде қандай типтер үйлеседі (бірдей)? Мысалдар келтіріңіз.
16. Құрылым дескрипторы деген не?
17. C тілінде қолданылмайтын C++ тілінің жиырма түйінді сөзін атап көрсетіңіз.
18. C++ тілінде `int x`; нұсқауы анықтау болып санала ма? Ал C тілінде ше?
19. C тілі стилінде типтерді келтіру әрекеті не істейді және ол несімен қауіпті?
20. `void*` типі не істейді және оның C және C++ тілдеріндегі нұсқаларының қандай айырмашылықтары бар?
21. C және C++ тілдеріндегі тізбелердің бір-бірінен қандай айырмашылықтары бар?
22. C тіліндегі программаларда кең таралған атаулар бірдей болып жатса, не істеу керек?
23. C тіліндегі бос жады аймағымен жұмыс істейтін кең таралған үш функцияны атаңыз.
24. C тілі стилінде анықтау қалай көрсетіледі?
25. C-тіркестері үшін `strcmp()` функциясы мен `==` операторының айырмашылықтары неде?
26. C-тіркестерін қалай көшіруге болады?
27. C-тіркестерінің ұзындығын қалай анықтауға болады?
28. `int` типіндегі бүтін сандардың үлкен жиымын қалай көшіруге болады?
29. `printf()` функциясының артықшылықтары мен кемшіліктерін атап көрсетіңіз.

30. `gets()` функциясын неге ешқашанда қолданбауға тырысу керек? Оның орнына нені қолануға болады?
31. С тіліндегі программада оқу үшін файлды қалай ашу керек?
32. С және С++ тілдеріндегі константалар (`const`) арасындағы айырмашылық неде?
33. Біз неге макростарды жақсы көрмейміз?
34. Әдетте макростар қалай қолданылады?
35. "Қосу сақшысы" дегеніміз не?

ТЕРМИНДЕР

`#define`	`malloc()`	интрузивті
`#ifdef`	`printf()`	интрузивті емес
`#ifndef`	`strcpy()`	құрылым дескрипторы
`Bell Labs`	`void`	лексикографикалық
С стилінде типті келтіру	`void*`	макрос
С/С++	айқын емес тип	үйлесімділік
С-тіркестері	асыра жүктеу	үшжақты салыстыру
`FILE`	байланыс	форматтық тіркес
`fopen()`	Брайан Керниган	шартты компиляция
`K&R`	Деннис Ритчи	

ЖАТТЫҒУЛАР

Бұл жаттығулар үшін барлық программаларды да С тілінің де және С++ тілінің де компиляторлары арқылы компиляциядан өткізу пайдалы болуы мүмкін. Егер тек С++ тілінің компиляторы пайдаланылса, онда кездейсоқ түрде С тілінде жоқ қасиеттер қолданылып кетуі мүмкін. Егер сіз тек С тілінің компиляторын пайдалансаңыз, онда типтерге байланысты қателер байқалмай қалуы мүмкін.

1. `strlen()`, `strcmp()` және `strcpy()` функцияларының нұсқаларын жүзеге асырыңыз
2. 27.9 бөліміндегі `List` интрузивтік контейнерімен орындалатын мысалды аяқтап, оның әрбір функциясын тесттен өткізіңіз.
3. 27.9 бөліміндегі `List` интрузивтік контейнерімен орындалатын мысалды өз қалауыңызша жетілдіріңіз. Мүмкіндігінше қателер санының барынша көп санын айқындап ұстап, өңдеуді қарастырыңыз. Мұнда құрылым анықтауларын өзгертуге, макростарды пайдалануға, т.с.с. әрекеттерді қолдануға болады.

4. Егер сіз 27.9 бөліміндегі `List` интрузивтік контейнерімен орындалатын мысалды C++ тілінде қайта жазбаған болсаңыз, онда соны жасаңыз да, әрбір функцияны тесттен өткізіңіз.
5. 3 және 4 жаттығулар нәтижелерін салыстырыңыз.
6. 27.9 бөліміндегі `Link` және `List` құрылымдарының функциялармен қамтамасыз етілген қолданушы интерфейсін өзгертпей, бірақ көрсетілу бейнелерін өзгертіңіз. Түйіндерді жиымдарда орналастырыңыз да, олардың `int` типіндегі (жиым индекстері) `first`, `last`, `pre` және `suc` мүшелерін қарастырыңыз.
7. C++ тілінің стандартты кітапханасындағы интрузивтік емес контейнерлерге қарағанда, интрузивтік контейнерлердің артықшылықтары мен кемшіліктерін атап көрсетіңіз. Осы контейнерлердің дұрыс және бұрыс жақтарын қамтитын аргументтері тізімін құрыңыз.
8. Сіздің компьютеріңізде қандай лексикографикалық тәртіп қабылданған? Өз пернетақтаңыздың әрбір символын және оның бүтінсандық кодын баспаға шығарыңыз; сонан соң символдарды олардың бүтінсандық кодтарына сәйкес тәртіппен баспаға шығарыңыз.
9. C тілінің құралдарын ғана пайдаланып, оның стандартты кітапханасын да қоса отырып, `stdin` ағымынан сөздер тізбегін оқып шығыңыз да, оны `stdout` ағымына лексикографикалық тәртіппен шығарыңыз. Көмек: C тіліндегі сұрыптау функциясы `qsort()` деп аталады; соның сипаттамасын тауып алыңыз. Бұған балама (альтернатива) ретінде сөздерді оқылу тәртібіне қарай реттелген тізімге кірістіріңіз. C тілінің стандартты кітапханасында тізімдер жоқ.
10. C++ тілінен немесе C with Classes (27.1 бөлімі) тілінен алынған C тілінің қасиеттері тізімін құрастырыңыз.
11. C++ тілінен *алынбаған* C тілінің қасиеттері тізімін құрастырыңыз.
12. `find(struct table*, const char*)`, `insert(struct table*, const char*, int)` және `remove(struct table*, const char*)` операциялары бар іздеу кестесін жүзеге асырыңыз (C-тіркестері арқылы немесе `int` типі арқылы). Бұл кестені құрылымдардың қос жиымы немесе қос жиым (`const char* []` және `int*`) түрінде бейнелеуге болады; керектісін өзіңіз таңдаңыз. Өз функцияларыңыз үшін қайтарылатын мәндер типтерін таңдап алыңыз. Жобалық шешіміңізді құжаттап шығыңыз.
13. C тілінде `string s; cin>>s;` нұсқауларының баламасы болып табылатын программа жазыңыз. Басқаша айтқанда, нөлмен аяқталатын символдарды және босорындармен бөлінген символдардың кез келген ұзын тізбегін жиымға оқитын енгізу операциясын анықтаңыз.
14. Кірісіне `int` типіндегі бүтін сандарды қабылдап алып, оның ең кіші және ең үлкен элементтерін анықтайтын функция жазыңыз. Ол және де медиана

мен орташа мәнді де есептеуі тиіс. Қайтарылатын мән ретінде нәтижелерді сақтайтын құрылымды пайдаланыңыз.

15. С тілінде жалқы мұралау әрекетін бейнелейтін (имитациялайтын) әрекет жасаңыз. Әрбір базалық класта жиымдар нұсқауышындағы функцияларға нұсқауыш (өзінің алғашқы аргументі ретінде базалық класс объектісіне нұсқауыш алатын жеке қолданылатын функциялар ретінде виртуалдық функцияларды модельдеу үшін) болатын болсын; 27.2.3 бөлімін қ. Базалық класты туынды кластың алғашқы мүшесінің типі етіп туынды класты шығаруды жүзеге асырыңыз. Әрбір класс үшін сәйкестікті сақтай отырып, виртуалдық функциялар жиымын инициалдаңыз. Оны тексеру үшін базалық және туынды `draw()` кластары бар `Shape` класымен орындалатын ескі мысал нұсқасын жүзеге асырыңыз. Мұнда тек С тілінің стандартында бар құралдар мен кітапхананы пайдаланыңыз.
16. Алдыңғы мысалды жүзеге асыруды күрделендіру үшін (белгілеулерді қарапайым ету арқылы) макростарды пайдаланыңыз.

СОҢҒЫ СӨЗ

Біз жоғарыда үйлесімділіктің барлық сауалдары жақсы түрде шешілген жоқ деп айтқан болатынбыз. Дегенмен, біреулер, бір кездерде, бір жерлерде С тілінде жазып қалдырған көптеген программалар (миллиардтаған жолдар) бар. Егер сізге соларды оқып және сондай программалар жазуға тура келсе, онда осы тарау сізді сондай жұмысқа дайындайды. Бірақ біз өзіміз С++ тілін дұрыс көреміз және осы тарауда оның аздаған себептерін де айтып кеттік. Бір сұрайтынымыз – `List` интрузивтік тізімінің мысалын дұрыс бағаламай жүрмеңіз, `List` интрузивтік тізімдері мен айқындалмаған типтер маңызды және қуатты технология (С тілінде де және С++ тілінде де) болып табылады.

V-БӨЛІМ

ҚОСЫМШАЛАР

Тілге қысқаша шолу

"Өз тілектеріңізге абай болыңыз – олар жүзеге асуы мүмкін"

- Мақал

Бұл қосымшада C++ тілінің элементтері жайлы негізгі мәліметтер қысқаша баяндалған. Ол сарапталынып алынған сипатта жазылып, кітапта жазылғандардан гөрі көбірек білгісі келетін үйренушілерге арналған. Бұл қосымшаның мақсаты – толықтық емес, қысқалық.

A.1 Жалпы мәліметтер

Бұл қосымша анықтамалық болып табылады. Оны жай тарау сияқты басынан аяғына дейін оқу міндетті емес. Мұнда жүйелі түрде (толығынан немесе жартылай) C++ тілінің түйінді элементтері сипатталған. Дегенмен, бұл толық анықтамалық емес, тек оның конспектісі ғана. Қосымша студенттер жиі қоятын сұрақтарға арналған. Көбінесе оқырмандарға толығырақ жауап алу үшін керекті

тақырыпқа сәйкес тарауларды оқып шығу керек. Бұл қосымшаны баяндалу дәлдігі мен терминологиясы бойынша стандарт эквиваленті деп айтуға болмайды. Оның орнына біз баяндаудың түсініктілігіне назар аударды қ. Оқырмандар бұдан толығырақ ақпаратты Stroustrup, *The C++ Programming Language* кітабынан таба алады. C++ тілінің анықталуы ISO C++ стандартында баяндалған, бірақ бұл құжат жаңа үйреніп жүргендерге сәйкес келе қоймайды. Өйткені бұл құжат оларға арналмаған. Желідегі құжаттаманы пайдалану мүмкіндігін де ұмытуға болмайды. Егер сіз қосымшаның алғашқы тарауларын оқып, оған көз салсаңыз, оның басым бөлігі түсінікті болып көріне қоймас. Бірақ басқа тарауларды да оқығаннан кейін барып, сіз барлығын да түсінетін боласыз.

Стандартты кітапхананың мүмкіндіктері Б қосымшасында сипатталған.

C++ тілінің стандарты INCITS (АҚШ), BSI (Ұлыбритания) және AFNOR (Франция) сияқты ұлттық стандарттау комитеттерінің ISO (International Organization for Standards – Халықаралық стандарттау ұйымы) басшылығымен жұмыс істейтін комитетпен бірігіп ынтымақтасуы арқасында анықталған. Оның қазіргі ресми жұмыс стандарты болып ISO/IEC 14882:2003 Standard for Programming Language C++ құжаты есептеледі. Оны электрондық түрде және Wiley (ISBN 2870846747) баспаханасы жариялаған қарапайым *The C++ Standard* кітабы түрінде де пайдалануға болады.

А.1.1 Терминология

C++ тілінің стандартында осы тілдің программасына және оның әртүрлі конструкцияларына келесідей анықтамалар берілген.

- *Стандартқа сәйкестігі*. C++ тілінде жазылған программа стандартқа сәйкестігі бойынша стандартқа сәйкес (conforming), немесе легальді (legal), немесе дұрыс (valid) деп аталады.
- *Жүзеге асырылуына тәуелділігі*. Программа қасиеттерге (int типінің мөлшері немесе 'a' символының сандық мәні сияқты) тәуелді болуы мүмкін (тәуелді болады да), олар тек берілген компиляторға ғана, операциялық жүйеге, машина архитектурасына, т.б. байланысты дәл анықталған. Жүзеге асырылуына байланысты болатын тіл қасиеттері стандартта көрсетіледі және ол компилятордың құрамындағы құжаттамасында көрсетілуі тиіс және де <limits> сияқты стандартты тақырыптарда (Б.1.1 бөлімін қ.) да болады. Сонымен, стандартқа сәйкестік программаның C++ тіліндегі әртүрлі нұсқаларына ауысымдылығына эквивалентті емес.
- *Анықталмағандығы.* Кейбір конструкциялардың *мағынасы дәл орнатылмаған* (unspecified), немесе *анықталмаған* (undefined), немесе *стандартқа сәйкес емес, бірақ диагностика қойылмайтын* (not conforming but requiring a diagnostic) болып табылады. Әрине, мұндай қасиеттерді

пайдаланбаған дұрыс. Бұл кітапта олар жоқ. Қолдануға болмайтын анықталмаған қасиеттерді тізіп көрсетейік.

- Әртүрлі бастапқы файлдардағы бір-бірімен сәйкестендірілмеген анықтаулар (тақырыптық файлдарды сәйкестендірілген түрде пайдаланыңыз; 8.3 бөлімді қ.).
- Өрнектегі бір айнымалыны қайталап оқу және қайталап жазу (негізгі мысал болып `a[i] = ++i;` нұсқауы саналады).
- Типтерді көптеген рет тікелей түрлендірулер (келтірулер), ерекше `reinterpret_cast`.

A.1.2 Программаны бастау және аяқтау

C++ тіліндегі программада жеке ауқымды (глобальді) `main()` атты функция болуы тиіс. Программа осы функцияны орындаудан басталады. `main()` функциясының қайтаратын типі `int` болады (бұған альтернативті `void` типі стандартқа сәйкес келмейді). `main()` функциясының қайтаратын мәні жүйеге беріледі. Кейбір жүйелер бұл мәнді есепке алмайды, бірақ программаның дұрыс аяқталуы нөлді, ал қате боп аяқталғаны – нөлге тең емес мәнді немесе әсерсіз қалған аластаманы (мұндай аластама жаман стильдің белгісі болып саналады) береді.

`main()` функциясының аргументтері жалпы тіл нұсқасына (жүзеге асырылуы) байланысты болады, бірақ кез келген нұсқа екі вариантты (бірақ нақты программа үшін тек біреуін) қарастыруы керек.

```
int main();     // аргументтері жоқ
int main(int argc, char* argv[]);
// argv[] жиымында argc С-тіркес бар
```

`main()` функциясының анықталуында қайтарылатын мәннің типін тікелей көрсету міндетті емес. Мұндай жағдайда программа орындалып соңына дейін жеткен соң, нөл мәнін қайтарады. C++ тіліндегі ең шағын программа мынадай болады:

```
int main()
```

Егер сіз конструкторы мен деструкторы бар ауқымды (атаулар кеңістігінде) объект анықтаған болсаңыз, онда конструктордың `main()` функциясына дейін, ал деструктордың `main()` функциясынан кейін орындалуы қисынды болып табылады. Былайша айтқанда, мұндай конструкторлардың орындалуы `main()` функциясын шақырудың бір бөлігі болып табылады, ал деструкторларды орындау

– **main()** функциясынан оралудың бір бөлігі болып саналады. Аз ғана мүмкіндік болғанның өзінде, егер олар қарапайым құру мен өшіруді талап етпесе, ауқымды объектілерді пайдаланбауға тырысыңыз.

A.1.3 Комментарийлер

Программада айтуға болатынның бәрі де айтылуы тиіс. Дегенмен C++ тілінде программалаушының кодпен өрнектей алмағанын жазып көрсететін комментарийлердің екі стилі бар.

```
// бұл бір жолдық комментарий
/*
    Бұл көп жолдық
    комментарийлер блогы
*/
```

Әрине, комментарийлер блогы көбінесе көп жолдық комментарийлер түрінде беріледі, әйтсе де кейбір адамдар оларды бірнеше бір жолдық комментарийлерге бөлгенді дұрыс көреді.

```
// Бұл үш бір жолдық
// комментарийлер түрінде берілген
// көп жолдық комментарий

/* ал мынау комментарийлер блогы ретінде
   берілген бір жолдық комментарий */
```

Комментарийлер кодтың атқаратын қызметі жайлы құжаттама беруде үлкен рөл атқарады; 7.6.4 бөлімді қ.

A.2 Литералдар

Литералдар әртүрлі мәндерді көрсетеді. Мысалы, 12 литералы он екі бүтін санын көрсетсе, **"Morning"** литералы – *Morning* сиволдық тіркесін, ал true литералы *true* бульдік мәнін бейнелейді.

A.2.1 Бүтінсандық литералдар

Бүтінсандық литералдардың (integer literals) үш түрі бар.

- Ондық: ондық цифрлар тізбегі.
  ондық цифрлар: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

- Сегіздік: нөлден басталатын сегіздік цифрлар тізбегі.
  Сегіздік цифрлар: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Он алтылық: 0x немесе 0X таңбаларынан басталатын он алтылық цифрлар тізбегі.
  Он алтылық цифрлар: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

`U` немесе `u` жалғауы (суффиксі) бүтінсандық литералдың таңбасы жоқтығын білдіреді, яғни `unsigned` спецификаторы бар (25.5.3 бөлімін қ.), ал `L` немесе `l` жалғауы `long` типіне қатысты, мысалы, `10u` немесе `123456UL`.

A.2.1.1 Сандық жүйелер

Әдетте біз сандарды ондық жүйеде жазамыз. `123` санындағы `1` жүздікке, `2` ондыққа, `3` бірлікке сәйкес келеді де, олардың қосындысы былай `1*100+2*10+3*1` немесе (дәрежелеу үшін `^` таңбасын пайдалансақ) `1*10^2+2*10^1+3*10^0` болады. Кейде ондық сөзі орнына "Санау базасы онға тең" (base-10) деп айтады. Мұндағы жағдайда 10 саны өрнектегі `1*base^2+2* base^1+3* base^0` дегенді білдіреді, яғни `base==10` шарты орындалады.Біздің неге ондық санау жүйесін пайдаланатынымызды түсіндіретін теория көп. Солардың бірі табиғи тілге жүгінеді: біздің қолымызда он саусағымыз бар, ал позициялық санау жүйесіндегі мынадай 0, 1 және 2 сияқты әрбір символ **digit** деп аталады. *Digit* сөзі латын тілінде саусақ дегенді білдіреді.

Ал, кейде басқа санау жүйелері де қолданылады. Көбінесе оң бүтін сандар компьютер жадында екілік санау жүйесінде бейнеленеді, яғни санау базасы 2 санына тең (0 мен 1 мәндері физикалық түрде жеңіл бейнеленеді). Аппараттық жабдықтаманың төменгі деңгейінде жиі жұмыс істеуге тиіс адамдар кейде сегіздік санау жүйесін (базасы 8-ге тең) пайдаланады, ал компьютер жадын адрестеу кезінде көбінесе оналтылық жүйе (базасы – 16) қолданылады.

Оналтылық жүйені қарастырайық. Біз 0-ден 15-ке дейінгі оналты мәнді атап шығуымыз керек.Әдетте ол үшін мынадай символдар: 0, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, мұндағы A – ондық жүйедегі 10 мәніне, B – 11 мәніне, т.с.с. мәндерге сәйкес келеді:

```
A==10, B==11, C==12, D==13, E==14, F==15
```

Енді `123` санын оналтылық санау жүйесінде `7B` деп жаза аламыз. Бұған сенімді болу үшін оналтылық санау жүйесінде `7B` саны `7*16+11` мәніне тең болатынына назар салыңыз, ол ондық жүйедегі – `123` саны. Және, керісінше, оналтылық `123` саны `1*16^2+2*16+3,` яғни `1*256+2*16+3` мәніне, ондық санау жүйесіндегі `291`-ге тең. Егер сіз ондық емес жүйелердегі бүтін сандар, онда біз сандарды ондық жүйеден оналтылық жүйеге түрлендіруге және керісінше, жаттығулар орындауыңызды ұсынамыз. Оналтылық цифрлардың өздерінің екілік мәндерімен өте қарапайым сәйкестігі бар екеніне назар аударыңыз.

Hexadecimal and binary								
hex	0	1	2	3	4	5	6	7
binary	0000	0001	0010	0011	0100	0101	0110	0111
hex	8	9	A	B	C	D	E	F
binary	1000	1001	1010	1011	1100	1101	1110	1111

Бұл оналтылық жүйенің кең таралғандығын түсіндіреді. Ашып айтар болсақ, байт мәні екі оналтылық цифрмен өрнектеледі.

C++ тілінде (қуанышқа орай) басқа жүйе тікелей көрсетілмесе, барлық сандар ондық болып табылады. Бір санды оналтылық деп айту үшін, оның алдына `0x` таңбалары (префикс) қойылуы тиіс, мысалы, `123 == 0x7B` және `0x123==291`. Осылайша біз оналтылық цифрларды a, b, c, d, e және f төменгі регистрде де пайдалана аламыз. Мысалы, `123 == 0x7b`.

Сегіздік жүйе сегізге тең санау базасына негізделген. Мұнда біз тек сегіз сегіздік цифрларды қолдана аламыз: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`. C++ тілінде сегіздік санау жүйесіндегі санда `0` таңбасынан басталады, яғни `0123` саны – ондық сан емес, ол `1*8^2+2*8+3`, яғни `1*64+2*8+3` немесе (ондық түрде) `83`. Және керісінше, `83` сегіздік саны, яғни `083`, ол `8*8+3`, яғни `67` ондық санына тең. C++ тілінің белгілеу жүйесін пайдаланып, мынадай теңдік аламыз `0123==83` және `083==67`.

Екілік жүйе екіге тең санау базасына негізделген. Мұнда тек екі цифр ғана бар: `0` және `1`. C++ тілінде екілік сандарды тікелей литералдар түрінде бейнелей алмаймыз. C++ тілінде литералдар және енгізу-шығару форматы ретінде тікелей тек сегіздік, ондық және оналтылық сандар ғана сүйемеледенеді. Бірақ екілік сандарды тіпті біз программа мәтінінде тікелей бейнелей алмасақ та, оларды білу пайдалы болып саналады. Мысалы, `123` ондық саны мынадай өрнекке тең:

`1*2^6+1*2^5+1*2^4+1*2^3+0*2^2+1*2+1`,

яғни ол `1*64+1*32+1*16+1*8+0*4+1*2+1`, яғни (екілік түрде) `1111011`.

A.2.2 Жылжымалы нүктелі литералдар

Жылжымалы нүктелі (floating-point-literal) литералдарда ондық нүкте (.), дәреже көрсеткіші (мысалы, e3) немесе жылжымалы нүктелі санды белгілейтін жалғау (d немесе f). Мысалдар қарастырайық.

```
123      // int (ондық нүкте, жалғау (суффикс)
             немесе дәреже көрсеткіші жоқ)
123.     // double:      123.0
```

```
123.0    // double
.123     // double:        0.123
0.123    // double
1.23e3   // double:        1230.0
1.23e-3  // double:        0.00123
1.23e+3  // double:        1230.0
```

Жылжымалы нүктелі литералдар, егер соңында басқа типті көрсететін жалғауы болмаса, **double** типінде болады. Мысалдар қарастырайық.

```
1.23      // double
1.23f     // float
1.23L     // long double
```

A.2.3 Бульдік литералдар

bool типіндегі литералдар болып **true** және **false** мәндері есептеледі. **true** литералының бүтінсандық мәні **1**, ал **false** литералынікі – **0**.

A.2.4 Символдық литералдар

Символдық литерал – бұл жалқы тырнақшаға алынған символ, мысалы, '**a**' немесе '**@**'. Оған қоса, бірсыпыра арнайы символдар бар:

Аты	**ASCII** кодындағы аты	C++ тіліндегі аты
Жаңа жол	**NL**	**\n**
Горизонталь табуляция	**HT**	**\t**
Вертикаль табуляция	**VT**	**\v**
Каретканы қайтару	**BS**	**\b**
Каретканы ауыстыру	**CR**	**\r**
Бетті ауыстыру	**FF**	**\f**
Дыбыстық сигнал	**BEL**	**\a**
Кері қиғаш сызық	****	****
Сұрақ белгісі	**?**	**\?**
Жалқы тырнақша	**'**	**\'**
Қос тырнақша	**"**	**\"**
Сегіздік сан	**ooo**	**\ooo**
Оналтылық сан	**hhh**	**\xhhh**

Арнайы символдар C++ тіліндегі жалқы тырнақшаға алынған аттары арқылы, мысалы, `'\n'` (жаңа жол) және `'\t'` (табуляция), бейнеленеді.

Символдар жиынында келесідей көрінетін символдар бар:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!@#$%^&*()_+|~`{}[]:";'<>?,./
```

Ауысымды кодта қосымша көрінетін символдар болмайды. Мысалы, `a` әрпі үшін `'\a'` символының мәні оның жүзеге асырылуына тәуелді болады (бірақ оны анықтау үшін, мысалы, `cout << int('a')`) нұсқауын орындау жеткілікті болып табылады).

A.2.5 Тіркестік литералдар

Тіркестік литерал (string literal) – қостырнақшаға алынған символдар тізбегі, мысалы, `"Knuth"` және `"King Canute"`. Тіркестік литералды кез келген жерден бірнеше жолға бөлуге болмайды; жаңа жолға көшу үшін арнайы `'\n'` символы қолданылуы тиіс.

```
"King
Canute "      // қате: тіркестік литералда жаңа жолға көшу
"King\nCanute"      // OK: жаңа жолға көшу дұрыс орындалған
```

Бір босорын таңбасымен ғана бөлініп жазылған екі сөз тіркесі бір тіркестік литерал болып саналады. Мысал қарастырайық.

```
"King" "Canute"
// "KingCanute" тіркесімен (бос орынсыз) бірдей
```

`\n` сияқты арнайы символдар тіркестік литералдарға кіретініне назар салыңыз.

A.2.6 Нұсқауыштық литералдар

Тек бір ғана *нұсқауыштық литерал* (pointer literal) бар, ол нөлдік нұсқауыш (`0`). Нөлдік нұсқауыш ретінде 0-ге тең кез келген константалық өрнекті пайдалануға болады. Мысалы:

```
t* p1 = 0;                  // OK: нөлдік нұсқауыш
int* p2 = 2-2;              // OK: нөлдік нұсқауыш
int* p3 = 1;          // қате: 1 – int, нөлдік нұсқауыш емес
```

```
int z = 0;
int* p4 = z;                    // қате: z – константа емес
```

Мұнда 0 мәні жанамалы түрде нөлдік нұсқауышқа айналып тұр. Көбінесе (бірақ тұрақты емес) нөлдік нұсқауыш `0` саны секілді тек қана нөлдерден тұратын биттік перде (маска) түрінде болады.

C++ тілінде (бірақ C тілінде емес, сондықтан C тілінің тақырыптарын қолданудан сақ болыңыз) `NULL` литералы анықтама бойынша 0-ге тең, сондықтан келесі кодты жазуға болады:

```
int* p4 = NULL;
// (NULL литералын дұрыс анықтағанда) нөлдік нұсқауыш
```

C++0x тілінде нөлдік нұсқауыш `nullptr` түйінді сөзімен белгіленетін болады, ал әзірше бұл үшін `0` санын пайдалануды ұсынамыз.

A.3 Идентификаторлар

Идентификатор (identifier) – бұл әріптен немесе астын сызу таңбасынан басталып, ары қарай әріптермен, цифрлармен немесе астын сызу таңбаларымен жалғасатын (немесе жалғаспайтын) символдар тізбегі (жоғарғы немесе төменгі регистрлерде).

```
int foo_bar;        // OK
int FooBar;         // OK
int foo bar;        // қате: идентификаторда босорын
                       таңбасы қолданылмайды
int foo$bar;        // қате: $ символы да идентификаторда
                       қолданылмайды
```

Астын сызу таңбасынан басталатын немесе қатарынан екі астын сызу таңбасы бар идентификаторлар компилятордың қолдануы үшін қалдырылған болып табылады; оларды пайдаланбаңыз. Мысалы:

```
int _foo;           // ұсынылмайды
int foo_bar;        // OK
int foo_ _bar;      // ұсынылмайды
int foo_;           // OK
```

A.3.1 Түйінді сөздер

Түйінді сөздер (keywords) – бұл тілдің өзінің тілдік конструкцияларын бейнелеу үшін пайдаланылатын идентификаторлары.

Түйінді сөздер (қордағы идентификаторлар)					
and	and_eq	asm	auto	bitand	bitor
bool	break	case	catch	char	class
compl	const	const_cast	continue	default	delete
do	double	dynamic_cast	else	enum	explicit
export	extern	false	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	not	not_eq	operator	or
or_eq	private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static	static_cast
struct	switch	template	this	throw	true
try	typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t	while
xor	xor_eq				

A.4 Көріну аймағы, жады класы және өмірлік мерзімі

C++ тіліндегі әрбір атаудың (препроцессор атынан басқа; A.17 бөлімін қ.) белгілі бір көріну аймағы (scope) бар; басқаша айтқанда, оны пайдалануға болатын мәтін аймағы болады. Мәліметтер (объектілер) компьютер жадында сақталады; объектіні сақтауға арналған жады түрі *жады класы* (storage class) деп аталады. Объектінің өмірлік мерзімі (lifetime) оны инициалдау сәтінен бастап, сол объектіні жойғанға дейінгі уақыт кезеңін құрайды.

A.4.1 Көріну аймағы

Көріну аймағының бес түрі (8.4 бөлімді қ.) бар:

- *Ауқымды (глобальді) көріну аймағы* (global scope). Егер ол тілдік конструкциядан тыс жарияланған (мысалы, кластан немесе функциядан тысқары) болса, онда атау ауқымды аймақта орналасқан болады.
- *Атаулар кеңістігінің көріну аймағы* (namespace scope). Егер ол атаулар кеңістігінде және белгілі бір тілдік конструкциядан тыс жарияланған

(мысалы, кластан немесе функциядан тысқары) болса, онда атау атаулар кеңістігінің көріну аймағында орналасқан болады. Формальды түрде айтқанда, ауқымды көріну аймағы – ол "бос атауы" бар атаулар кеңістігінің көріну аймағы.

- *Жергілікті (локальді) көріну аймағы* (local scope). Егер ол функция ішінде жарияланған (функция параметрлерін қоса алғанда) болса, онда атау жергілікті көріну аймағында орналасқан болады.
- *Кластың көріну аймағы* (class scope). Егер ол осы кластың мүшесінің аты болып табылатын болса, онда атау кластың көріну аймағында орналасқан болады.
- *Нұсқаудың көріну аймағы* (statement scope). Егер ол **for**, **while**, **switch** немесе **if** нұсқауларының ішкі бөлігінің **(...)** көріну аймағында жарияланған болса, онда атау нұсқаудың көріну аймағында орналасқан болады.

Айнымалының көріну аймағы ол өзі жарияланған нұсқаудың аяғына дейін (тек қана) жалғасады. Мысал қарастырайық.

```
for (int i = 0; i<v.size(); ++i) {
     //  i айнымалысы осында пайдаланыла алады
}
if (i < 27)
// for нұсқауындағы i айнымалысы көріну аймағынан шықты
```

Кластың және атаулар кеңістігінің көріну аймақтарының өз аттары болады, сондықтан олардың мүшелеріне сырттан сілтеме жасай аламыз. Мысал қарастырайық.

```
void f();     //  ауқымды көріну аймағында

namespace N {
   void f()      //  N көріну аймағы кеңістігінде
   {
       int v;       //  жергілікті көріну аймағында
       ::f();       // f() ауқымды функциясын шақыру
   }
}

void f()
{
    N::f();     // N көріну аймағынан f() функциясын шақыру
}
```

Егер біз `N::f()` немесе `::f()` функцияларын шақырсақ, не болар еді? A.15 бөлімін қ.

A.4.2 Жады класы

Үш *жады класы* (17.4 бөлімді қ.) бар:

- *Автоматты жады* (automatic storage). Егер айнымалылар `static` түйінді сөзі арқылы тікелей жарияланбаған болса, онда функциялар ішінде (функция параметрлерін қоса алғанда) анықталған айнымалылар автоматты жады аймағында (яғни, стекте) орналасатын болады. Функция шақырылғанда автоматты жады аймағы бөлініп беріледі де, басқару функцияны шақырған модульге қайтарылып берілгенде бөлінген жады босатылады. Сонымен, егер функция (тікелей немесе жанамалы түрде) өзін-өзі шақыратын болса, автоматты мәліметтердің бірнеше көшірмелері болуы мүмкін: мұндайда әрбір шақыруға бір-бір көшірмеден (8.5.8 бөлімін қ.) болады.
- *Статикалық жады* (static storage). Ауқымды көріну аймағында және атаулар кеңістігінің көріну аймағында жарияланған айнымалылар функциялар мен кластардағы `static` түйінді сөзі арқылы тікелей жарияланған айнымалылар тәрізді статикалық жады аймағында сақталатын болады. Байланыс редакторы статикалық жады аймағын программа іске қосылғанша бөліп береді.
- *Бос жады, яғни үйінді* (free store – heap). `new` операторы арқылы құрылған объектілер бос жады аймағында орналасады.

Мысал қарастырайық.

```
vector<int> vg(10);    // программа іске қосылғанда бір рет
                       // құрылады ("main() функциясына дейін")

vector<int>* f(int x)
{
  static vector<int> vs(x);//f() функциясын тек алғашқы рет
                           // шақырғанда ғана құрылады
   vector<int> vf(x+x);
   // f() функциясын әрбір шақырған сайын құрылады

   for (int i=1; i<10; ++i) {
        vector<int> v1(i);
        // әрбір итерация сайын құрылады
        // . . .
}// v1 айнымалысы осы жерде жойылады (әрбір итерация сайын)
```

```
    return new vector<int>(vf);
    // vf айнымалысы көшірмесі ретінде
    // бос жады аймағында құрылады
} // vf айнымалысы осы жерде жойылады

void ff()
{
    vector<int>* p = f(10); // f() функциясынан вектор алады
    // . . .
    delete p; // f функциясынан алынған векторды жояды
}
```

Статикалық жады аймағында орналасқан **vg** және **vs** айнымалылары, егер олар құрылған болса, программа аяқталған соң (**main()** функциясынан соң) жойылады.

Класс мүшелері үші жады жеке бөлінбейді. Егер сіз объектіні бір жерде орналастырсаңыз, онда статикалық емес мүшелер де сол жерде орналасады (объектінің өзі орналасқан және өзі құрамына кіретін жады класында).

Код мәліметтерден бөлек орналасады. Мысалы, функция-мүше өз класының әрбір объектісінде сақталмайды; оның бір көшірмесі программа кодының басқа бөлігімен бірге сақталады.

14.3 және 17.4 бөлімдерін де қ.

А.4.3 Өмірлік мерзімі

Объектіні қолдану (ашық түрде) алдында, ол инициалдануы тиіс. Мұндай инициалдауды тікелей инициализатор арқылы немесе жанамалы түрде конструкторды пайдалану жолымен немесе құрамдас типтерді келісім бойынша инициалдау ережесіне сәйкес атқаруға болады. Объектінің өмірлік кезеңі оның көріну аймағымен және жады класымен анықталған нүктеде аяқталады (мысалы, 17.4 және Б4.2 бөлімдерін қ.).

- *Жергілікті, яғни локальді (автоматты) объектілер* орындалу ағымы олардың анықталуына жеткен кезде құрылады да, көріну аймағынан шыққан кезде жойылады.
- *Уақытша объектілер* нақты ішкі өрнек арқылы құрылады да, толық өрнек аяқталған кезде жойылады. Толық өрнек – бұл басқа толық өрнектің құрамына кірмейтін өрнек болып табылады.
- *Атаулар кеңістігіндегі объектілер және кластардың статикалық мүшелері* программа басында (**main()** функциясы алдында) құрылады да, программа аяқталған (**main()** функциясынан соң) кезде жойылады.
- *Жергілікті статикалық объектілер* орындалу ағымы олардың анықта-

луына жеткен кезде құрылады да, программа аяқталған кезде (егер олар құрылған болса) жойылады.

- *Бос жады аймағындағы объектілер* **new** операторы арқылы құрылады да, **delete** операторы арқылы (міндетті емес) жойылады.

Жергілікті сілтемеге байланысты болатын уақытша айнымалы сілтеме қанша уақыт істейтін болса, ол да сонша уақыт жұмыс істейді. Мысал қарастырайық.

```
const char* string_tbl[]=
              { "Mozart", "Grieg", "Haydn", "Chopin" };
const char* f(int i) { return string_tbl[i]; }
void g(string s){}

void h()
{
     const string& r = f(0);
     // уақытша тіркесті r сілтемесімен байланыстырамыз
     g(f(1));        // уақытша тіркес құрып, оны жібереміз
     string s = f(2); // s-ті уақытша тіркеспен инициалдаймыз
     cout << "f(3): " << f(3)
     // уақытша тіркес құрамыз да, оны жібереміз
          <<" s: " << s
          << " r: " << r << '\n';
}
```

Нәтиже төмендегідей болады:

```
f(3): Chopin s: Haydn r: Mozart
```

f(1), **f(2)** және **f(3)** функцияларын шақыру кезінде пайда болған уақытша тіркестер оларды құрған өрнек соңында жойылады. Бірақ **f(0)** функциясын шақыру кезінде пайда болған уақытша тіркес **r** айнымалысымен байланысты болады және ол **h()** функциясының соңына дейін "өмір сүреді".

A.5 Өрнектер

Бұл бөлімде C++ тілінің операторлары сипатталады. Біз мұнда өзіміз мнемоникалық деп санайтын белгілеулерді пайдаланамыз: мысалы, **m** – мүше аты үшін; **T** – тип аты үшін; p – нұсқауыш құратын өрнек үшін; **x** – өрнек үшін; **v** – lvalue өрнегі үшін; **lst** – аргументтер тізімі үшін. Арифметикалық операциялардың нәтижелері типтері кәдімгі арифметкалық түрлендірулер бойынша анықталады (A.5.2.2 бөлімді қ.). Бұл бөлімде келтірілген сипаттамалар, программалаушының

өзі анықтай алатын операторларға емес, тек құрамдас операторларға қатысты болып саналады. Ал, өз операторларымызды анықтай отырып, құрамдас операторлар үшін орнатылған семантикалық ережелерді ұстануымыз керек (9.6 бөлімді қ.).

Көріну аймағының шешімі
`N::m` `m` `N` атаулар кеңістігінде болады; `N` – атаулар кеңістігінің немесе кластың аты.

Мұнда мүшелердің өздері қабаттаса келуі мүмкін, сондықтан мынадай `N::C::m` өрнектер алуымыз мүмкін екендігіне назар аударыңыз (8.7 бөлімін де қ.).

Постфикстік өрнектер	
`x.m`	Класс мүшесіне қол жеткізу; `x` класс объектісі болуы тиіс.
`p -> m`	Класс мүшесіне қол жеткізу; `p` класс объектісіне нұсқауыш болуы тиіс; `(*p).m` өрнегіне эквивалентті
`p[x]`	Индекстеу; `*(p+x)` өрнегіне эквивалентті
`f(lst)`	Функцияны шақыру; `lst` аргументтер тізімімен бірге `f` функциясын шақыру
`T(lst)`	Құру: `lst` аргументтер тізімімен бірге `T` объектісін құру
`v++`	(Постфикстік) инкременттеу; `v++` мәні инкременттеуге дейінгі `v` мәніне тең
`v--`	(Постфикстік) декременттеу; `v--` мәні декременттеуге дейінгі `v` мәніне тең
`typeid(x)`	Программаны орындау кезінде `x` объектісінің типін идентификациялау
`typeid(T)`	Программаны орындау кезінде `T` типін идентификациялау
`dynamic_cast<T>(x)`	Программаны орындау кезінде `x` объектісін `T` типіне келтіру және тексеру
`static_cast<T>(x)`	Программаны компиляциядан өткізу кезінде `x` объектісін `T` типіне келтіру және тексеру
`const_cast<T>(x)`	`x` объектісі типінен `T` типін алу үшін `const` спецификаторын қосуға немесе жоюға барып тірелетін тексерілмейтін түрлендіру
`reinterpret_cast<T>(x)`	`x` объектісінің биттік комбинациясын өзге бір интерпретациялау арқылы `x` объектісін `T` типіне тексерілмейтін түрде келтіру

`typeid` операторы мен оның қолданылуы жайлы бұл кітапта айтылмаған; оның нақтылықтарын бұдан күрделі оқулықтардан табуға болады. Келтіру операторларының өз аргументтерін өзгертпейтініне назар аударыңыз. Оның орнына олар аргумент мәніне белгілі бір түрде сәйкес келетін өз типтерінің нәтижесін құрады (A.5.7 бөлім).

Унарлық өрнектер	
`sizeof(T)`	`T` типінің байтпен берілген мөлшері (көлемі)
`sizeof(x)`	`x` объектісі жататын тип мөлшері (байтпен)
`++v`	(Префикстік) инкременттеу; `v+=1` өрнегімен бірдей
`--v`	(Префикстік) декременттеу; `v-=1` өрнегімен бірдей
`~x`	`x`-ке толықтырма; `~` – биттік операция
`!x`	`x`-ке терістеу; `true` немесе `false` мәнін қайтарады
`&v`	`v` айнымалысының адресі
`*p`	`p` нұсқауышы сілтеме жасап тұрған объектінің мазмұны
`new T`	Бос жады аймағында `T` типіндегі объект құрады
`new T(lst)`	Бос жады аймағында `T` типіндегі объект құрады да, оны `lst` объектісімен инициалдайды
`new(lst) T`	`lst` аргументімен берілген жады аймағында `T` типіндегі объект құрады
`new(lst) T(lst2)`	`lst` аргументімен берілген жады аймағында `T` типіндегі объект құрады да, оны `lst2` аргументімен инициалдайды
`delete p`	`p` нұсқауышы сілтеме жасап тұрған объектіні жояды
`delete[] p`	`p` нұсқауышы сілтеме жасап тұрған объектілер жиымын жояды
`(T)x`	C тілі стилінде келтіру; `x` объектісін `T` типіне келтіру

`delete p` және `delete[] p` нұсқауларында `p` нұсқауышы сілтеме жасап тұрған объектілер компьютер жадында new операторы арқылы орналасуы тиіс (A.5.6 бөлімі). `(T)x` өрнегі аздап нақты болып табылмайды, сондықтан ол бұдан нақтырақ келтіру операторларына қарағанда, қателерге бой алдыруы мүмкін (A.5.7 бөлімін қ.).

Класс мүшесін таңдау	
`x.*ptm`	`ptm` класының мүшесіне нұсқауышпен анықталған `x` объектісінің мүшесі
`p->*ptm`	`ptm` класының мүшесіне нұсқауышпен идентификацияланған `*p` мүшесі

Бұл нұсқаулар кітапта қарастырылмайды; күрделірек оқулықтарға көз салыңыз.

Мультипликативті операторлар	
`x*y`	`x`-ті `y`-ке көбейту
`x/y`	`x`-ті `y`-ке бөлу
`x%y`	`x`-ті `y`-ке модулі бойынша (бөлгендегі қалдық) бөлу (жылжымалы нүктелі типтер үшін емес)

Егер `y==0` болса, онда `x/y` және `x%y` өрнектерінің нәтижесі анықталмаған. Егер `x` немесе `y` айнымалысы теріс болса, онда `x%y` өрнегінің нәтижесі де теріс болады.

Аддитивті операторлар	
`x+y`	`x` пен `y`-ті қосу
`x-y`	`x`-тен `y`-ті азайту

Жылжыту операторлары	
`x<<y`	`x` биттерін солға қарай `y` орынға жылжыту
`x>>y`	`x` биттерін оңға қарай `y` орынға жылжыту

Құрамдас типтер үшін `>>` және `<<` операторлары биттерді жылжытуды білдіреді (25.5.4 бөлімді қ.). Егер сол жақ операнд `iostream` класының объектісі болса, онда бұл операторлар енгізу-шығару үшін қолданылады (10-11 тарауларды қ.).

Салыстыру операторлары	
`x<y`	`x` `y`-тен кіші; `bool` типіндегі объектіні қайтарады
`x<=y`	`x` `y`-тен кіші немесе тең
`x>y`	`x` `y`-тен үлкен; `bool` типіндегі объектіні қайтарады
`x>=y`	`x` `y`-тен үлкен немесе тең

Салыстыру операторының нәтижесі `bool` типіндегі мән болып табылады.

Теңдік операторлары	
`x==y`	`x` `y`-ке тең; `bool` типіндегі мәнді қайтарады
`x!=y`	`x` `y`-ке тең емес

`x!=y` және `!(x==y)` өрнектерінің бірдей екендігіне назар аударыңыз. Теңдік операторының нәтижесі `bool` типіндегі мән болып табылады.

Биттер бойынша "және" (and)	
`x&y`	`x` пен `y` айнымалылары үшін биттер бойынша "және" операторы

`&` операторы (`^, |, ~, >>` және `<<` операторлары тәрізді) биттер комбинациясын қайтарады. Мысалы, егер `a` мен `b` айнымалылары `unsigned char` типінде болса, онда `a&b` өрнегінің нәтижесі де `unsigned char` типінде болады, мұндағы әрбір бит `a` мен `b` айнымалыларының сәйкес биттеріне `&` операторын қолдану нәтижесі болып табылады (А.5.5 бөлімі).

Биттер бойынша аластамалы (арифметикалық) "немесе" (xor)	
`x^y`	`x` пен `y` айнымалылары үшін биттер бойынша аластамалы "немесе" операторы

Биттер бойынша "немесе" (or)	
`x|y`	`x` пен `y` айнымалылары үшін биттер бойынша "немесе" операторы

Логикалық "және"	
`x&&y`	Логикалық "және"; `true` және `false` мәндерін қайтарады; мұнда егер `x` мәні `true` болса ғана, `y` мәні есептеледі

Логикалық "немесе"	
`x||y`	Логикалық "немесе"; `true` және `false` мәндерін қайтарады; мұнда егер `x` мәні `false` болса ғана, `y` мәні есептеледі

А.5.5 бөлімін қ.

Шартты өрнек	
`x?y:z`	Егер `x` мәні `true` болса, онда нәтиже `y`-ке тең; әйтпесе `z`-ке тең

Мысал қарастырайық.

```
template<class T> T& max(T& a, T& b) { return (a>b)?a:b; }
```

"Сұрақ белгісі" операторы 8.4 бөлімінде сипатталған.

Меншіктеу	
`v=x`	`x` мәні `v` айнымалысына меншіктеледі; нәтижесі `v`-ға тең
`v*=x`	`v=v*(x)` баламасы
`v/=x`	`v=v/(x)` баламасы
`v%=x`	`v=v%(x)` баламасы
`v+=x`	`v=v+(x)` баламасы
`v-=x`	`v=v-(x)` баламасы
`v>>=x`	`v=v>>(x)` баламасы
`v<<=x`	`v=v<<(x)` баламасы

`v&=x`	`v=v&(x)` баламасы
`v^=x`	`v=v^(x)` баламасы
`v|=x`	`v=v|(x)` баламасы

"`v=v*(x)` баламасы" деген сөз `v*=x` өрнегінің мәні `v=v*(x)` өрнегінің мәнімен бірдей дегенді білдіреді, тек `v` мәні бір-ақ рет есептеледі. Мысалы, `v[++i]*=7+3` өрнегі `(v[++i]=v[++i]*(7+3))` дегенді емес (ол анықталмаған болуы мүмкін; 8.6.1 бөлімін қ.), `(++i, v[i]=v[i]*(7+3))` дегенді білдіреді.

throw өрнегі	
`throw x`	`x` мәнін туындатады (генерациялайды)

`throw` өрнегінің нәтижесінің типі `void`.

"үтір" өрнегі	
`x,y`	`x` нұсқауын, сонан соң `y` нұсқауын орындайды. Нәтиже болып `y` нұсқауының нәтижесі есептеледі

Әрбір кесте приоритеттері бірдей операторлардан тұрады. Жоғарырақ жақта орналасқан кестелердегі операторлар приоритеттері төменірек орналасқан операторлардан басым болып келеді. Мысалы, `a+b*c` өрнегінің мәні `(a+b)*c` емес, `a+(b*c)` болып табылады, өйткені * операторының приоритеті + операторынан жоғары екені белгілі. Осы сияқты, `*p++` өрнегі `(*p)++` емес, `*(p++)` өрнегіне сәйкес келеді. Унарлық операторлар мен меншіктеу операторлары оңассоциативті (right-associative) болып саналады; қалғандарының барлығы да – солассоциативтілерге (left-associative) жатады. Мысалы, `a=b=c` өрнегі `a=(b=c)` дегенді білдіреді, ал `a+b+c` өрнегі `(a+b)+c` тіркесіне сәйкес келеді.

`Lvalue` – бұл өзгертіп толықтыруға болатын объект. Әрине, `const` спецификаторы бар `lvalue` объектісі өзгертілуден типтер жүйесі арқылы қорғалады және оның адресі бар. `lvalue` өрнегіне қарама-қарсы `rvalue` объектісі болып табылады, яғни ол өзгертілмейтін немесе адресі жоқ бір өрнек, мысалы, функция қайтаратын мән (`&f(x)` – қате, өйткені `f(x)` функциясы қайтаратын мән `rvalue` мәні болып табылады).

A.5.1 Қолданушы анықтаған операторлар

Жоғарыда тізіп көрсетілген ережелер құрамдас типтер үшін орнатылған. Егер қолданушы анықтаған оператор пайдаланылса, онда өрнек жай ғана қолданушы анықтаған сәйкес операторлық функцияны шақыруға түрленеді де, әрекеттер тізбегі функцияны шақыру үшін орнатылған ережелермен анықталады. Мысал қарастырайық.

```
class Mine { /* . . . */ };
bool operator==(Mine, Mine);

void f(Mine a, Mine b)
{
      if (a==b) {  // a==b means operator==(a,b)
            // . . .
      }
}
```

Қолданушы анықтаған тип – бұл класс (9-тарауды, А.12 бөлімді қ.) немесе тізбе (9.5, А.11 бөлімдерін қ.).

А.5.2 Типті жанамалы түрлендіру

Бүтін сандық типтер немесе жылжымалы нүктелі типтер (А.8 бөлім) меншіктеу операторлары мен өрнектерде еркін түрде араласа алады. Мәндер алғашқы мүмкіндіктің өзінде-ақ ақпаратты жоғалтпайтындай түрде түрленеді. Өкінішке орай, мәндерді өшіретін түрлендірулер тікелей түрде орындалады.

А.5.2.1 Қозғаулар

Мәндерді сақтай отырып, жанамалы түрде орындалатын *қозғаулар* (promotions) деп аталады. Мысалы, қысқалау бүтін сандық типтерден int типін жасау үшін атқарылатын арифметикалық операцияларды орындау алдында *бүтінсандық қозғау* (integral promotion) орындалады. Бұл қозғаулардың бастапқы мақсатын бейнелейді: арифметикалық операциялардың операндтарын "табиғи мөлшерге" келтіру. Бұған қоса, `float` типіндегі мәндерді `double` типіндегі мәндерге түрлендіру де қозғау деп аталады.

Қозғаулар кәдімгі арифметикалық түрлендірулердің бір бөлігі ретінде пайдаланылады (А.5.2.2 бөлімді қ.).

А.5.2.2 Түрлендірулер

Іргелі типтегі мәндерді бірінен біріне әртүрлі тәсілдермен түрлендіруге болады. Программа жазған кезде ақпаратты өзгертіп жіберетін анықталмаған жағдайлар мен нәтижесі белгісіз түрлендірулерден алшақ болған дұрыс (3.9 және 25.5.3 бөлімдерді қ.). Компиляторлар әдетте көптеген күмәнді түрлендірулерден сақтандыра алады.

- *Бүтінсандық түрлендірулер*. Бүтін сан басқа бір бүтін типке түрлене алады. Тізбенің мәнін бүтін типке түрлендіруге болады. Нәтижелік тип таңбасыз тип (`unsigned`) болатын жағдайда, егер ол мақсаттық жады аймағына сыйып орналаса алатын болса (қажет болып жатса, жоғарғы биттер алынып тасталады), онда нәтижелік мән бастапқы мәнде қанша бит болса, сонша биттен тұратын болады. Егер мақсаттық тип таңбалы болып және оны мақсаттық тип арқылы бейнелеуге болатын болса, онда мән өзгеріссіз қалады; қарсы жағдайда, мән тілдің нұсқасымен (жүзеге асырылған) анықталады. `Bool` және `char` типтері бүтінсандық тип болатынына назар аударыңыз.

- *Жылжымалы нүктелі мәндерді түрлендіру*. Жылжымалы нүктелі мәнді жылжымалы нүктелі басқа типтегі мәнге түрлендіруге болады. Егер бастапқы мәнді мақсатты типте дәл бейнелеуге болатын болса, онда нәтиже болып бастапқы сандық мән есептеледі. Егерде бастапқы мән екі мақсаттық мәндер арасында жататын болса, онда нәтиже осы екеуінің біріне тең болады. Басқаша айтқанда, нәтиженің қандай болатынын айта алмаймыз. `float` типіндегі мәнді `double` типіне түрлендіру қозғау болып есептелетініне назар аударыңыз.

- *Нұсқауыштар мен сілтемелерді түрлендіру*. Объект типіне жасалған кез келген нұсқауышты `void*` типіндегі нұсқауышқа түрлендіруге болады (17.8 және 27.3.5 бөлімдерін қ.). Туынды класқа жасалған нұсқауышты (сілтемені) жанамалы түрде қолжетімді және бір мәнді түрде анықталған базалық класқа түрлендіруге болады (14.3 бөлімді қ.). Нөлге тең константалық өрнекті жанамалы түрде нұсқауыштың кез келген басқа бір типіне түрлендіруге болады. `T*` типіндегі нұсқауышты жанамалы түрде `const T*` нұсқауышына түрлендіруге болады. Осы сияқты `T*` нұсқауышын жанамалы түрде `const T&` типіндегі сілтемеге түрлендіруге болады.

- *Бульдік түрлендірулер*. Нұсқауыштарды, бүтін сандарды және жылжымалы нүктелі сандарды жанамалы түрде `bool` типіне түрлендіруге болады. Нөлге тең емес мәндер `true` мәніне, ал нөл – `false` мәніне түрленеді.

- *Жылжымалы нүктелі сандарды бүтін сандарға түрлендіру*. Егер жылжымалы нүктелі сан бүтін санға түрлендірілетін болса, онда оның бөлшегі алынып тасталады. Басқаша айтқанда, жылжымалы нүктелі типті бүтін типке түрлендіру санды қиып қысқарту болып табылады. Егер қиылып қысқартылған мәнді мақсаттық тип арқылы бейнелеу мүмкін болмаса, онда нәтиже туралы алдын ала ештеңе айтуға болмайды. Бүтін санды жылжымалы нүктелі санға түрлендіру математикалық тұрғыдан алғанда, аппаратық жабдықтама мүмкіндік беретін дәрежеде ғана дұрыс болып табылады. Егер бүтін санды жылжымалы нүктелі сан ретінде дәл бейнелеуге болмайтын болса, онда дәлдікті жоғалту орын алады.

- *Кәдімгі арифметикалық түрлендірулер.* Мұндай түрлендірулер бинарлық операторлар операндтарымен орындалады да, оларды ортақ бір типке келтіру мақсатында және осы типте нәтижені бейнелеу үшін қолданылады.

  1. Егер операндтардың бірі `long double` типінде болса, онда екіншісі де осы `long double` типіне түрлендіріледі. Қарсы жағдайда, егер операндтардың бірі `double` типінде болса, онда екіншісі де осы `double` типіне түрлендіріледі. Әйтпесе, егер операндтардың бірі `float` типінде болса, онда екіншісі де осы `float` типіне түрлендіріледі. Ал егер екеуі де бүтін операнд болып жатса, онда қозғау әрекеті орын алады.
  2. Егер операндтардың бірі `unsigned long` типінде болса, онда екіншісі де осы `unsigned double` типіне түрлендіріледі. Қарсы жағдайда, егер операндтардың бірі `long int` типінде болса, ал екіншісі – `unsigned int` болса, онда `unsigned int` типі `long int` типіне түрлендіріледі, мұнда `long int` типі `unsigned int` типінің барлық мәндерін де бейнелей алады деп саналады. Қарсы жағдайда, екі операнд та `unsigned long int` типіне түрлендіріледі. Қарсы жағдайда, егер операндтардың бірі `long` типінде болса, екіншісі де `long` типіне түрленеді. Қарсы жағдайда, егер операндтардың бірі `unsigned` типінде болса, екіншісі де `unsigned` типіне түрленеді. Әйтпесе екі операнд та `int` типінде болады.

Әрине, типтердің шым-шытырық араласу нәтижесіне сенім артпай, жанамалы түрде түрлендірулер қажеттілігін азайтуға тырысу керек.

A.5.2.3 Қолданушы анықтаған түрлендірулер

Стандартты түрлендірулер мен қозғаулардан басқа программалаушы қолданушы анықтаған типтерді түрлендірулерді енгізе алады. Бір аргументті қабылдайтын конструктор осы аргументті өз типіне түрлендіруді анықтайды. Егер конструктордың `explicit` спецификаторы (18.3.1 бөлімді қ.) бар болатын болса, онда программалаушы осы әрекетті тікелей орындауды талап етсе ғана түрлендіру орындалады.

A.5.3 Константалық өрнектер

Константалық өрнектер (constant expression) – бұл компиляциядан өткізу кезеңінде есептелетін және тек қана `int` типіндегі операндтардан тұратын өрнек. екіншісі де осы `double` типіне түрлендіріледі. (Бұл шамалы қарапайым түрде берілген анықтама, бірақ ол көптеген мақсаттар үшін қолданыла алады.) Мысал қарастырайық.

```
const int a = 2*3;
const int b = a+3;
```

Константалық өрнектер аздаған жағдайларда ғана, мысалы, жиымдар шекарасын, **case** бөлімдерінің белгілерін, тізбелер инициализаторларын және **int** типіндегі шаблондық аргументтерді есептегенде талап етіледі. Мысал қарстырайық.

```
int var = 7;
switch (x) {
case 77:     // OK
case a+2:    // OK
case var:    // қате (var – константалық өрнек емес)
      // . . .
};
```

A.5.4 sizeof операторы

sizeof(x) өрнегінде **x** аргументі тип немесе өрнек болуы мүмкін. Егер **x** – өрнек болса, онда **sizeof(x)** мәні болып нәтижелік объектінің мөлшері саналады. Егер **x** – тип болса, онда **sizeof(x)** мәні болып **x** типіндегі объектінің мөлшері есептеледі. Мөлшерлер байтпен өлшенеді. Анықтама бойынша **sizeof(char)==1**.

A.5.5 Логикалық өрнектер

C++ тілінде бүтінсандық типтер үшін логикалық операторлар қарастырылған.

Биттер бойынша логикалық операциялар	
x&y	**x** пен **y** үшін биттер бойынша "және" операциясы
x|y	**x** пен **y** үшін биттер бойынша "немесе" операциясы
x^y	**x** пен **y** үшін биттер бойынша аластамалы "немесе" операциясы

Логикалық операциялар	
x&&y	Логикалық "және"; **true** немесе **false** мәнін қайтарады; **x** тек **true** мәніне тең болса, **y** есептеледі
x||y	Логикалық "немесе"; **true** немесе **false** мәнін қайтарады; **x** тек **false** мәніне тең болса, **y** есептеледі
x^y	**x** пен **y** үшін биттер бойынша аластамалы "немесе" операциясы

Бұл операторлар өздерінің операндтарының әрбір битіне қолданылады, ал **(&&** және **||)** логикалық операторлары 0 санын **false** мәні деп, ал қалғандарының барлығын да **true** деп есептейді. Бұл операторлардың анықтаулары төменде келтірілген.

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

A.5.6 new және delete операторлары

Бос жады аймағы (динамикалық жады немесе үйінді) **new** операторы арқылы бөлінеді де, **delete** (жеке объектілер үшін) немесе **delete[]** (жиым үшін) операторлары арқылы босатылады.

Егер жады аймағы болмай қалса, онда **new** операторы **bad_alloc** аластамасын туындатады. Егер **new** операторы табысты орындалса, ол аз дегенде бір байт орын бөледі де, нұсқауышты компьютер жадында орналасқан объектіге қайтарады. Бұл объектінің типі **new** операторы орындалғаннан кейін анықталады. Мысал қарастырайық.

```
int* p1 = new int;
// int типіндегі (инициалданбаған) санды орналастырады
int* p2 = new int(7);
// int типіндегі 7 санымен инициалданған санды орналастырады
int* p3 = new int[100];
// int типіндегі (инициалданбаған) 100 санды орналастырады
// . . .
delete p1;        // жеке (индивидуалды) объектіні жояды
delete p2;
delete[] p3;      // жиымды жояды
```

Егер **new** операторы арқылы сіз компьютер жадына құрамдас типтегі объектілер орналастырсаңыз, егер инициализатор көрсетілмесе, олар инициалданбайды. Егер **new** операторы арқылы сіз компьютер жадына конструкторы бар класс объектілерін орналастырып, инициализаторды көрсетпесеңіз, онда сол конструктор шақырылады (17.4.4 бөлімді қ.).

Delete операторы, егер бар болатын болса, әрбір операндтың деструкторын шақырады. Деструктордың виртуалды (A.12.3.1 бөлім) бола алатынына назар аударыңыз.

A.5.7 Келтіру операторлары

Типтерге келтіретін төрт оператор бар.

Типтерге келтіру операторлары	
`x=dynamic_cast<D*>(p)`	`p` нұсқауышын `D*` типіне келтіруге тырысады (0 қайтара алады)
`x=dynamic_cast<D&>(*p)`	`*p` нұсқауышын `D&` типіне келтіруге тырысады (`bad_cast` аластамасын туындата алады)
`x=static_cast<T>(v)`	Егер `T` типін `v` операндының типіне келтіруге болатын болса, `v` операндының типін `T` типіне келтіреді
`x=reinterpret_cast<T>(v)`	`v` операндының типін сол биттер комбинациясымен бейнеленген `T` типіне келтіреді
`x=const_cast<T>(v)`	`const` спецификаторын жоя отырып, `v` операндының типін `T` типіне келтіреді
`x=(T)v`	C тілі стилінде келтіру: кез келген ескі келтіруді орындайды
`x=T(v)`	Функционалдық келтіру: кез келген ескі келтіруді орындайды

Егер `p` нұсқауышы базалық класқа нұсқауыш болса, ал `D` класы базалық кластан туынды класс болатын болса, онда динамикалық келтіру кластар иерархиясы бойынша жылжу үшін керек. Егер `v` операнды `D*` типіне жатпайтын болса, онда бұл операция `0` қайтарады. Егер қате болған жағдайда, `dynamic_cast` операциясы 0 қайтармай, `bad_cast` аластамасын туындатуы қажет болса, оны нұсқауыштарға емес, сілтемелерге қолдану керек. Динамикалық келтіру – программаның орындалуы кезінде типтерді тексеруге негізделген жалғыз келтіру түрі.

Статикалық келтіру "ойлы түрлендірулер" үшін қолданылады, яғни `v` операнды `T` типінің жанамалы түрде түрлендірілу нәтижесі болуы мүмкін (17.8 бөлімді қ.).

`reinterpret_cast` операторы биттер комбинациясын қайта интерпретациялау үшін қолданылады. Оның ауысымдылығына кепілдік берілмейді. Негізінде оны, тіпті ауыспайтын деп есептеу керек. Программада машиналық адресті алудағы қайта интерпретациялаудың дәстүрлі мысалы болып бүтін санды нұсқауышқа түрлендіру болып табылады (17.8 және 25.4.1 бөлімдерін қ.).

C тілі стиліндегі келтірулер мен функционалдық келтірулер `static_cast` немесе `reinterpret_cast` операторларын `const_cast` операторымен біріктіру арқылы орындауға болатын кез келген типті түрлендіруді атқара алады.

Келтірулерден алшақ болған дұрыс. Көбінесе оларды пайдалану программалаудың нашар стилін қолдану белгісі болып саналады. Бұл ереженің ерекшеліктері 17.8 және 25.4.1 бөлімдерінде көрсетілген. C тілі стиліндегі келтірулер мен функционалдық келтірулердің келеңсіз қасиеті бар: олар өздерінің не жасап жатқанын сізге білдірмейтін мүмкіндікті таңдайды (27.3.4 бөлімді

қ.). Егер сіз типті тікелей түрлендіруден кете алмасаңыз, атаулы келтірулерді пайдаланыңыз.

A.6 Нұсқаулар

нұсқау:
    жариялау
    { *нұсқау_тізімі*$_{opt}$ **}**
    try { *нұсқау_тізімі*$_{opt}$ **}** *өңдеуіштер_тізімі*
    өрнек$_{opt}$ **;**
    таңдау_нұсқауы
    итерация_нұсқауы
    белгілері_бар_нұсқау
    басқару_нұсқауы

таңдау_нұсқауы:
    if *(шарт)* *нұсқау*
    if (*шарт***)** *нұсқау* **else** *нұсқау*
    switch (*шарт***)** *нұсқау*

итерация_нұсқауы:
    while (*шарт***)** *нұсқау*
    do *нұсқау* **while (***шарт***);**
    for (*инициалдау_нұсқауы; шарт*$_{opt}$**;** *өрнек*$_{opt}$**)** *нұсқау*

белгілері_бар_нұсқау:
    case *(константалық_өрнек* **:** *нұсқау*
    default : *нұсқау*
    identifier : *нұсқау*

басқару_нұсқауы:
    break ;
    continue ;
    return *өрнек*$_{opt}$ **;**
    goto *идентификатор* **;**

нұсқау_тізімі :
    нұсқау_тізімі$_{opt}$ *нұсқауы:*

шарт:
    өрнек
    тип_спецификаторы жарияланатын_объект **=** *өрнек*

инициалдау_нұсқауы_for :
    өрнек$_{opt}$ **;**
    тип_спецификаторы жарияланатын_объект **=** *өрнек* **;**

өңдеуіштер_тізімі:
    catch (*аластаманы жариялау***) { *нұсқау_тізімі*$_{opt}$ }**
    өңдеуіштер_тізімі өңдеуіштер_тізімі$_{opt}$

Жариялау – бұл нұсқау, ал меншіктеу мен функцияны шақыру өрнектер болып табылады. Бұл анықтамаға келесі тізімді қосу керек.

- Итерация (**for** және **while**); 4.4.2 бөлімді қ.
- Таңдау (**if**, **switch**, **case** және **break**); 4.4.1 бөлімді қ. **break** нұсқауы ең жақын қабаттаса орналасқан **switch**, **while**, **do** және **for** нұсқауларының жұмысын тоқтатады.
- Өрнектер; А.5 және 4.3 бөлімдерін қ.
- Жариялаулар; А.6 және 8.2 бөлімдерін қ.
- Аластамалар (**try** және **catch**); 5.6 және 19.4 бөлімдерін қ.

Нұсқаулардың әртүрлілігін (олар қандай есеп шығарады?) бейнелеу үшін жасалған мысал қарастырайық.

```
int* f(int p[], int n)
{
    if (p==0) throw Bad_p(n);
    vector<int> v;
    int x;
    while (cin>>x) {
        if (x==terminator) break;   // while циклынан шығу
        v.push_back(x);
    }
    for (int i = 0; i<v.size() && i<n; ++i) {
        if (v[i]==*p)
            return p;
        else
            ++p;
       }
    return 0;
}
```

A.7 Жариялаулар

Жариялау (declaration) үш бөліктен тұрады:

- жарияланатын болмыстың аты;
- жарияланатын болмыстың типі;
- жарияланатын болмыстың бастапқы мәні (көптеген жағдайларда міндетті емес).

Біз келесі болмыстарды жариялай аламыз:

- құрамдас типтер объектілері және қолданушы анықтаған типтер (A.8 бөлімі);
- қолданушы анықтаған типтер (кластар мен тізбелер) (A.10-A11бөлімдері, 9-тарау);
- шаблондар (шаблондық кластар мен функциялар) (A.13 бөлімі);
- алтернативті атаулар (A.16 бөлімі);
- атаулар кеңістігі (A.15 және 8.7 бөлімдер);
- функциялар (функция-мүшелер мен операторларды қоса) (A.9 бөлімі, 8-тарау);
- тізбелер (тізбелер мәндері) (A.11 және 9.5 бөлімдері);
- макростар (A.17.2 және 27.8 бөлімдері).

A7.1 Анықтаулар

Компьютердің жады аймағын сақтап қоятын немесе компиляторға программада атауын қолдануға керекті барлық ақпаратты белгілі бір басқа түрде беретін инициалдауы бар анықтауды *анықтама* деп атайды (definition). Программадағы әрбір типтің, объектінің және функцияның бір ғана анықтамасы болуы керек. Мысал қарастырайық.

```
double f();                     // жариялау
double f() { /* . . . */ };     // (және де) анықтау
extern const int x;             // жариялау
int y;                          //  (және де) анықтау
int z = 10;              // тікелей инициалдауы бар анықтау
```

Константалар инициалданған болуы тиіс. Егер константа `extern` түйінді сөзі арқылы жарияланбаса (мұндайда инициализатор анықтауымен бірге басқа жерде орналасуы тиіс) немесе константаның келісім бойынша конструкторы бар типі жоқ болса, онда оны (константаны) инициалдау үшін инициализатор қолданылады (A.12.3 бөлімі). Кластың константалық мүшелері әрбір конструкторда инициализатор арқылы инициалдануы керек (A.12.3 бөлімі).

A.8 Құрамдас типтер

C++ тілінің көптеген функционалдық типтері мен модификаторлар арқылы іргелі типтерден құралған типтері бар.

Құрамдас типтер	
`bool x`	`x` – бульдік айнымалы (`true` немесе `false`)
`char x`	`x` – символ (әдетте 8 бит)
`short x`	`x` – қысқа `int` типі (әдетте 16 бит)
`int x`	`x` – кәдімгі бүтін тип
`float x`	`x` – жылжымалы нүктелі сан (қысқа `double` типі)
`double x`	`x` – екі еселі дәлдіктегі жылжымалы нүктелі сан
`void* p`	`p` – (белгісіз типтегі) жады ұясына нұсқауыш
`T* p`	`p` – `T` типіндегі объектіге нұсқауыш
`T *const p`	`p` – `T` типіндегі объектіге константалық (өзгермейтін) нұсқауыш
`T a[n]`	`a` – `T` типіндегі `n` элементтен тұратын жиым
`T& r`	`r` – `T` типіндегі объектіге сілтеме
`T f(arguments)`	`f` – `arguments` аргументтер тізімін алатын және `T` типіндегі объектіні қайтаратын функция
`const T x`	`x` – `T` типіндегі константалық (өзгермейтін) нұсқадағы объект
`long T x`	`x` – `T` ұзын типіндегі объект
`unsigned T x`	`x` – таңбасыз `T` типіндегі объект
`signed T x`	`x` – таңбалы `T` типіндегі объект

Мұндағы `T` "белгілі бір" типті білдіреді, сондықтан оның `long unsigned int`, `long double`, `unsigned char` және `const char *` (`char` константалық символына нұсқауыш) нұсқалары бар. Бірақ бұл жүйе онша толық емес; мысалы, онда `short double` типі жоқ (оның рөлін `float` типі ойнайды); `signed bool` типі жоқ (тіпті, мағынасы жоқ); `short long int` (бұл артық болар еді) және `long long long long int` типі жоқ. Кейбір компиляторлар C++0x стандартын күте жүріп, `long long int` типін ("өте ұзын бүтін тип" деп оқылады) қолданады. `long long` типі 64 биттен төмен болмайды деп кепілдік беріледі.

Жылжымалы нүктелі тип (**floating-point types**) – **float, double** және **long double** типтері. Олар C++ тіліндегі нақты сандардың жуық мәні болып табылады.

Бүтінсандық типтер (**integer types**), кейде *интегралдық типтер* (**integral types**) деп аталатындар – бұлар **boll**, **char**, **short**, **int**, **long** және (C++0x тілінде) **long long** типтері және солардың таңбасыз нұсқалары. Тізбелер типі мен мәндерін көбінесе бүтінсандық тип немесе мән орнына қолдануға болады.

Құрамдас типтердің мөлшерлері 3.8, 17,3,1 және 25.5.1 бөлімдерінде талқыланды; нұсқауыштар мен жиымдар – 17 және 18-тарауларда; сілтемелер – 8.5.4-8.5.6 бөлімдерде қарастырылды.

A.8.1 Нұсқауыштар

Нұсқауыш (pointer) – бұл объект немесе функция адресі. Нұсқауыштар нұсқауыштық типтегі айнымалыларда сақталады. Объектіге дұрыс жасалған нұсқауышта сол объектінің адресі болады.

```
int x = 7;
int* pi = &x; // pi нұсқауышы x объектісіне сілтеме жасап тұр
int xx = *pi;  // *pi – pi нұсқауышы сілтеме жасап тұрған
объектінің мәні, яғни 7
```

Дұрыс жасалмаған нұсқауыш – бұл бірде бір объектіге нұсқауышы жоқ нұсқауыш.

```
int* pi2;         // инициалданбаған
*pi2 = 7;         // анықталмаған тәртіп
pi2 = 0;          // нөлдік нұсқауыш
                // (pi2 нұсқауышы дұрыс нұсқап тұрған жоқ)
*pi2 = 7;         // анықталмаған тәртіп
pi2 = new int(7); // енді pi2 нұсқауышы дұрыс істеп тұр
int xxx = *pi2;
// өте дұрыс: xxx айнымалысы 7-ге тең болып тұр
```

Біз барлық дұрыс жұмыс істемей тұрған нұсқауыштар нөлдік (0) болғанын қалаймыз, сондықтан тексеру жүргізе аламыз:

```
if (p2 == 0) {      // "егер нұсқауыш дұрыс болмаса"
        // *p2 пайдаланбаңыз
}
```

Немесе одан да қарапайым:

```
if (p2) {           // "егер нұсқауыш дұрыс болса"
        // *p2 пайдаланыңыз
}
```

17.4 және 18.5.4 бөлімдерді қ.

Объектілерге (`void` емес) нұсқауыштармен орындалатын операцияларды тізіп көрсетейік:

Нұсқауыштармен орындалатын операциялар	
`*p`	Атаусыз ету/жадыға тікелей емес (жанамалы) түрде қатынасу
`p[i]`	Атаусыз ету/индекстеу
`p=q`	Меншіктеу және инициалдау
`p==q`	Теңдік
`p!=q`	Теңсіздік
`p+i`	Бүтін санды қосу
`p-i`	Бүтін санды азайту
`p-q`	Қашықтық: нұсқауыштарды есептеу
`++p`	Префикстік инкременттеу (алға жылжу)
`p++`	Постфикстік инкременттеу (алға жылжу)
`--p`	Префикстік декременттеу (артқа жылжу)
`p--`	Постфикстік декременттеу (артқа жылжу)
`p+=i`	Алға `i` элементке жылжу
`p-=i`	Артқа `i` элементке жылжу

Нұсқауыштар арифметикасының операциялары (мысалы, `++p` және `p+=7`) жиым элементтеріне сілтеме жасайтын нұсқауыштарға қолданыла алады, ал жиымнан тыс жады аймағына сілтеме жасайтын нұсқауышты атаусыз ету әсері анықталмаған (дұрысын айтар болсақ, оны компилятор немесе программаны орындау жүйесі тексере алмайды).

Тек `void*` типті нұсқауышпен орындалатын операциялар көшіру (меншіктеу немесе инициалдау) және келтіру (типтерді түрлендіру) болып табылады.

Функцияға нұсқауышты (27.2.5 бөлімді қ.) тек көшіріп алуға және шақыруға болады. Мысал қарастырайық.

```
typedef void (*Handle_type)(int);
void my_handler(int);
Handle_type handle = my_handler;
handle(10);          // my_handler(10)  эквиваленті
```

А.8.2 Жиымдар

Жиым (array) ұзындықтары тұрақты болып келетін біртекті типтегі үздіксіз объектілер тізбегі.

```
int a[10];        //  10 бүтін сан
```

Егер жиым ауқымды (глобальді) болса, онда оның элементтері осы тип үшін келісім бойынша қабылданған сәйкес мәндермен инициалдана алады. Мысалы, `a[7]` мәні `0`-ге тең болсын. Егер жиым жергілікті (локальді) болса (айнымалы функцияда жарияланған) немесе `new` операторы арқылы құрылған болса, онда құрамдас типтердің элементтері инициалданбай қалады, ал қолданушы типі бар элементтер оның конструкторымен инициалданады.

Жиым аты жанамалы түрде оның бірінші элементіне нұсқауышқа айналады. Мысал қарастырайық.

```
int* p = a;  // p нұсқауышы a[0] элементіне
             // сілтеме жасайды (нұсқайды)
```

Жиым немесе жиым элементіне нұсқауыш `[ ]` операторы арқылы индекстеле алады. Мысал қарастырайық.

```
a[7] = 9;
int xx = p[6];
```

Жиым элементтері нөлден бастап нөмірленеді (18.5 бөлімі).

Жиым индекстерінің диапазоны тексерілмейді. Оның үстіне, олар көбінесе нұсқауыштар арқылы берілетіндіктен, диапазонды тексеруге арналған ақпарат қолданушыларға сенімсіз тәсілмен беріледі. `vector` класын пайдалануды ұсынамыз.

Жиым көлемі – оның элементтері көлемдерінің қосындысы. Мысал қарастырайық.

```
int a[max];
// sizeof(a)==sizeof(a[0])*max==sizeof(int)*max
```

Жиымдардың жиымын (екі өлшемді жиым), жиымдар жиымдарының жиымын (көп өлшемді жиым) анықтап алып, пайдалануға болады. Мысал қарастырайық.

```
double da[100][200][300];
// double типіндегі 100 элементтен тұратын  типтегі
// 200 элементтен тұратын  типтегі 300 элемент
da[7][9][11] = 0;
```

Көп өлшемді жиымдарды онша қарапайым емес түрде пайдалану – қатеге төтеп бере алмайтын әлсіз әрекет (24.4 бөлімді қ.). Егер сізде таңдау болса, `Matrix` класын қолданған дұрыс (24 тараудағы тәрізді).

A.8.3 Сілтемелер

Сілтеме (reference) – бұл синоним (alias), яғни объектінің баламалы аты.

```
int a = 7;
int& r = a;
r = 8;   // a айнымалысы 8-ге тең
```

Көшіруді болдырмас үшін сілтемелер көбінесе функциялар параметрі ретінде пайдаланылады.

```
void f(const string& s);
// . . .
f("бұл жолды көшіру өте қымбат,
            сондықтан сілтеме қолданылады");
```

8.5.4–8.5.6 бөлімдерді қ.

A.9 Функциялар

Функция (function) – бұл аргументтер жиынын алып (бос та болуы мүмкін), мән қайтаратын (міндетті емес), ат қойылған код фрагменті. Функция қайтарылатын мән типін көрсетіп, онан кейін функция атын және параметрлер тізімін беру арқылы жарияланады.

```
char f(string, int);
```

Сонымен, `f` – бұл `string` және `int` типіндегі объектілерді қабылдап алып, `char` типіндегі объектіні қайтаратын функция. Егер функция тек жай ғана жарияланып, бірақ анықталмайтын болса, онда оның жариялануы нүктелі үтірмен аяқталады. Егер функция анықталуы тиіс болса, онда аргументтерді жариялағаннан кейін, функция операторлары (денесі, тұлғасы) орналасады:

```
char f(string s, int i) { return s[i]; }
```

Функция тұлғасы блок (8.2 бөлімді қ.) немесе `try` блогы (5.6.3 бөлімді қ.) болуы тиіс.

Жариялануында белгілі бір мән қайтаратыны көрсетілген функция, оны қайтаруы тиіс (**return** операторын қолдану арқылы):

```
char f(string s, int i) { char c = s[i]; }
// қате: ештеңе қайтарылмаған
```

main() функциясы осы ережеге бағынбайтын жалғыз функция (А.1.2 бөлімді қ.) болып табылады. **main()** функциясынан бөлек басқа функциядан да мән қайтарғыңыз келмесе, функция алдына **void** түйінді сөзі қойыңыз. Басқаша айтқанда, **void** сөзін қайтарылатын мәннің типі ретінде пайдаланыңыз.

```
void increment(int& x) { ++x; } // OK: мән қайтару керек емес
```

Функция шақыру операторы () арқылы өзіне сәйкес керекті аргументтерімен шақырылады:

```
char x1 = f(1,2); // қате: f() функциясының бірінші
                  // аргументі тіркес болуы керек
string s = "Battle of Hastings";
char x2 = f(s);
// қате: f() функциясы екі аргументтің болуын талап етеді
char x3 = f(s,2);  // OK
```

Функциялар туралы толығырақ ақпаратты 8 тараудан қ.

А.9.1 Асыра жүктеуді шешу

Асыра жүктеуді шешу (overload resolution) – бұл аргументтердің жиыны негізінде функцияны шақыру үшін оны таңдау процесі. Мысал қарастырайық.

```
void print(int);
void print(double);
void print(const std::string&);

print(123);         // print(int) шақырылады
print(1.23);        // print(double) шақырылады
print("123");       // print(const string&) шақырылады
```

Тіл ережелерін негізге ала отырып, компилятор дұрыс функцияны өздігінен таңдап ала алады. Өкінішке орай, бұл ережелер бір шама күрделі, өйткені олар күрделі мысалдарды барынша есепке алуға тырысады. Бұл жерде біз олардың қарапайым етілген нұсқасын келтіреміз.

Асыра жүктелген функцияның дұрыс нұсқасын таңдау функция аргументтері типі мен оның параметрлері типі арасындағы ең жақсы сәйкестікті іздеу негізінде жүргізіледі.

Біздің бейнелеуімізді нақтылау үшін ең жақсы сәйкестікті таңдау туралы бірнеше критерийлерді келтірейік:

- Дәлме-дәл сәйкестік, яғни типтерді түрлендіру толық болмаған кезде немесе тек ең қарапайым түрлендірулер болғанда ғана сәйкес келуі (мысалы, жиым атын нұсқауышқа, функция атын – функцияға нұсқауышқа және `T` типін – `const T` типіне түрлендіру).
- Қозғаудан кейінгі сәйкестік, яғни бүтінсандық қозғаулар (`bool` – `int` типіне, `char` – `int` типіне, `short` – `int` типіне және солардың таңбасыз нұсқалары (аналогтары); А.8 бөлімін қ.) және де `float` типін `double` типіне түрлендіру.
- Стандартты түрлендірулерден кейінгі сәйкестік, мысалы, `int` – `double` типіне, `double` – `int` типіне, `double` – `long double` типіне, `Derived*` – `Base*` типіне (14.3 бөлімін қ.), `T*` – `void*` типіне, `int` – `unsigned int` типіне түрлендірулер (25.5.3 бөлімін қ.).
- Қолданушы анықтаған түрлендірулерден кейінгі сәйкестік (А.5.2.3 бөлімін қ.).
- Функция жариялауындағы ... эллипсис негізіндегі сәйкестік (А.9.3 бөлімін қ.).

Егер екі сәйкестік табылса, шақыру бірдей болмағандықтан аласталады. Асыра жүктеуді шешу ережелері негізінен құрамдас сандық типтерге бағытталған (А.5.3 бөлімін қ.).

Асыра жүктеуді бірнеше аргументтер негізінде шешу мақсатында біз алдымен әрбір аргумент үшін ең жақсы сәйкестікті табуымыз керек. Мұнда әрбір аргумент бойынша басқа функциялар тәрізді жақсы сәйкес келетін функция таңдалып алынады, бірақ шақыруға басқаларынан ең жақсы сәйкес келу бір аргумент бойынша ғана орындалады; әйтпесе шақыру бір мәнді болып саналмайды. Мысал қарастырайық.

```
void f(int, const string&, double);
void f(int, const char*, int);
f(1,"hello",1);     // OK: f(int, const char*, int) шақыру
f(1,string("hello"),1.0);
// OK: f(int, const string&, double) шақыру
f(1, "hello",1.0); // error: бір мәнді сәйкестік жоқ
```

Соңғы шақыруда `"hello"` тіркесі түрлендірусіз `const char*` типіне сәйкес келеді, ал `const string&` типіне – тек түрлендіруден кейін ғана сәйкес келеді.

Басқа жағынан, 1.0 саны түрлендірусіз **double** типіне, ал **int** типіндегі сан тек түрлендіруден кейін сәйкес келеді, сондықтан **f()** функциясының бірде бір нұсқасы ережелерге басқаларынан жақсы сәйкес келмейді.

Егер осы қарапайым етілген ережелер сіздің компиляторыңыздың ережелеріне және сіздің ойыңызға сәйкес келмесе, онда бірінші кезекте сіздің программаңызды қажетті мөлшерден күрделі деп ұққан жөн. Оны қарапайым етуге тырысыңыз, әйтпесе сарапшылардан кеңес алыңыз.

A.9.2 Келісім бойынша аргументтер

Кейде функциялардың жиі кездесетін қалыпты жағдайдан гөрі аргументтері көбірек болып жатады. Мұны есепке алу үшін, функцияны шақырған кезде, керекті аргументтер берілмей жатса, программалаушы келісім бойынша қолданылатын аргументтер қарастыра алады. Мысал қарастырайық.

```
void f(int, int=0, int=0);
f(1,2,3);
f(1,2);     // f(1,2,0) шақырулар
f(1);       // f(1,0,0) шақырулар
```

Келісім бойынша тек соңғы аргументтерді беруге болады. Мысал қарастырайық.

```
void g(int, int =7, int);
// қате: келісім бойынша аргумент соңғы емес
f(1,,1);          // қате: екінші аргумент қойылмаған
```

Келісім бойынша берілген аргументтерге балама (альтернатива) ретінде асыра жүктеу (және керісінше) бола алады.

A.9.3 Анықталмаған аргументтер

Функцияны аргументтер санын да, олардың типтерін де көрсетпей беруге болады. Ол үшін "мүмкін басқа аргументтер де" дегенді білдіретін (**...**) эллипсис пайдаланылады. Мысалы, C тіліндегі ең белгілі функция **printf()** болар (27.6.1 және Б.10.2 бөлімдерін қ.), соған орай жариялану мен кейбір шақырулар мынадай болып та көрінеді:

```
void printf(const char* format...); // форматтық тіркесті
           // алады және тағы да бірдеңелерді алуы мүмкін

int x = 'x';
printf("hello, world!");
```

```
printf("print a char '%c'\n",x);
// символ түрінде бүтін x санын баспаға шығарады
printf("print a string \"%s\"",x);  // "өз аяғын ату"
```

Форматтық тіркестегі `%c` және `%s` сияқты формат спецификаторлары аргументтерді пайдалану тәсілін анықтайды. Жоғарыда көрсетілген сияқты, бұл өте келеңсіз жағдайларға алып келуі мүмкін. C++ тілінде анықталмаған аргументтерден алшақ болған дұрыс.

A.9.4 Байланыстар спецификациясы

C++ тіліндегі код көбінесе бір программа ішінде C тілінде жазылған кодпен қатар қолданыла береді; басқаша айтқанда, бір бөлігі C++ тілінде (C++ тілі компиляторымен тексеріледі) жазылса, ал басқасы – C тілінде (C тілі компиляторы арқылы компиляциядан өтеді) жазылады. Мұндай мүмкіндікті пайдалану үшін C++ тілі программалаушыларға, бір немесе бірнеше функция C тілінен шақырылатынын көрсететін *байланыстар спецификациясын* (linkage specifications) ұсынады. C тілімен байланыстар спецификациясын функцияны жариялау алдына қоюға болады.

```
extern "C" void callable_from_C(int);
```

Мұны балама ретінде блоктағы барлық жарияланымдарға қолдануға болады.

```
extern "C" {
      void callable_from_C(int);
      int and_this_one_also(double, int*);
      /* . . . */
}
```

Нақтылықтарын 27.2.3 бөлімнен табуға болады.

C тілінде функцияларды асыра сілтеу мүмкіндігі жоқ, сондықтан C тілімен байланыс спецификациясын асыра сілтенген функцияның бір нұсқасына орналастыра аласыз.

A.10 Қолданушы анықтаған типтер

Жаңа типті (қолданушы анықтаған) класс түрінде (`class`, `struct` және `union`) және тізбелер түрінде (`enum`; A.11 бөлімін қ.) анықтайтын екі тәсіл бар.

A.10.1 Операцияларды асыра жүктеу

Программалаушы қолданушы типіндегі операндтарды қабылдайтын көптеген операторлардың мағынасын анықтай алады. Құрамдас операторлар үшін операторлардың стандартты мағынасын өзгерту немесе жаңа оператор енгізу мүмкін емес. Қолданушы анықтаған (асыра жүктелген оператор) оператор аты `operator` түйінді сөзінен және соған жалғасқан оператор символынан тұрады; мысалы, `+` операторын анықтайтын функция аты мынадай `operator +` болады.

```
Matrix operator+(const Matrix&, const Matrix&);
```

Мысалдарын `std::ostream` (Chapters 10–11), `std::vector` (Chapters 17–19, §B.4), `std::complex` (§B.9.3), және `Matrix` (Chapter 24) тұлғасы блок (8.2 бөлімді қ.) кластары анықтауларынан табуға болады.

Келесі операторлардан басқаларының барлығын да асыра жүктеуге болады:

```
?: . .*  ::  sizeof  typeid
```

Келесі операторлар анықтайтын функциялар класс мүшелері болуы тиіс:

```
=  [ ]  ( )  ->
```

Барлық қалған операторларды мүше-функциялар түрінде және жеке өзіндік функциялар ретінде анықтауға болады.

Әрбір қолданушы типінің келісім бойынша анықталған `=` (меншіктеу және инициалдау), `&` (адрес алу) және `,` (үтір) операторлары болатынына назар аударыңыз.

Операторларды асыра жүктеген кезде, ынсаптылық сақтап, жалпы бекітілген келісімдерді сақтауға тырысу керек.

A.11 Тізбелер

Тізбелер (enumeration) аттары тізіп көрсетілген мәндер жиыны бар типті анықтайды.

```
enum Color { green, yellow, red };
```

Келісім бойынша тізбедегі алғашқы мән `0`-ге тең, мұндағы `green==0`, ал қалған мәндер біртіндеп бірге артып отырады, мысалдағы `yellow == 1` және `red == 2`. Оған қоса, тізбедегі бір мәнді тікелей көрсетуге де болады:

```
enum Day { Monday=1, Tuesday, Wednesday };
```

Сонымен, мұндағы **Monday==1**, **Tuesday==2** және **Wednesday==3**.

Тізбедегі мәндер өз тізбесінің көріну аймағына емес, сыртқы көріну аймағына жататынын айта кету керек.

```
int x = green;          // OK
int y = Color::green;   // қате
```

Тізбелер және олардың мәндері жанамалы түрде бүтін сандарға түрленеді, бірақ бүтін сандар жанамалы түрде тізбелер типіне түрлендірілмейді.

```
intx=green; //OK: Color-ді int-ке жанамалы түрде түрлендіру
Color c = green;      // OK
c=2;  //қате: int-ті Color-ға жанамалы түрде түрлендіру жоқ
c=Color(2);    // OK: (тексерілмейтін) тікелей түрлендіру
inty=c;    //OK: Color-ді int-ке жанамалы түрде түрлендіру
```

Тізбелерді пайдалану жайлы 9.5 бөлімде айтылады.

A.12 Кластар

Класс (class) – бұл қолданушы, оның объектілері мен операциялары үшін бейнелеуді қолдануға болатын етіп анықтаған тип.

```
class X {
public:
        // қолданушы интерфейсі
private:
        // жүзеге асыру
};
```

Кластың жариялауында анықталған айнымалылар, функциялар және типтер осы кластың мүшелері деп аталады. Олардың техникалық нақтылықтары 9-тарауда баяндалған.

A.12.1 Класс мүшелеріне қол жеткізу

Кластың **ашық** мүшесіне қолданушылар қол жеткізе алады: **жабық** класс мүшелеріне тек класс мүшелері ғана қол жеткізе алады:

```
class Date {
public:
```

```
        // . . .
        int next_day();
private:
        int y, m, d;
};

void Date::next_day() { return d+1; }     // OK

void f(Date d)
{
        int nd = d.d+1;   // қате: Date::d – кластың жабық мүшесі
        // . . .
}
```

Құрылым – бұл мүшелері, келісім бойынша, ашық болып табылатын класс:

```
struct S {
    // (егер тікелей жабық болып жарияланбаса, ашық) мүшелер
};
```

Класс мүшелеріне қол жеткізу туралы толығырақ ақпарат, қорғалған мүшелерді талқылаумен бірге, 14.3.4 бөлімінде келтірілген.

Объект мүшелерін, оның атына қолданылған `.` (нүкте) операторы арқылы немесе оған нұсқауышқа қолданылған `->` (тілсызық) операторы арқылы, пайдалануға болады:

```
struct Date {
    int d, m, y;
    int day() const { return d; } //класта анықталған
    int month() const;
    // жай ғана жарияланған; басқа жерде анықталған
    int year() const;
    // жай ғана жарияланған; басқа жерде анықталған
};

Date x;
x.d = 15;                   // айнымалы арқылы қол жеткізу
int y = x.day();            // айнымалы арқылы шақыру
Date* p = &x;
p->m = 7;                   // нұсқауыш арқылы қол жеткізу
int z = p->month();         // нұсқауыш арқылы шақыру
```

Класс мүшелеріне `::` (көріну аймағын шешу) операторы арқылы сілтеме жасауға болады:

```
int Date::year() const { return y; } //out-of-class definition
```

Класс функция-мүшелерінде, класс атын көрсетпей-ақ, кластың басқа мүшелеріне сілтеме жасауға болады:

```
struct Date {
    int d, m, y;
    int day() const { return d; }
    // . . .
};
```

Мұндай атаулар функция шақыратын объектіге қатысты болады:

```
void f(Date d1, Date d2)
{
   d1.day();       // d1.d мүшесін пайдалану (қол жеткізу)
   d2.day();       // d2.d мүшесін пайдалану (қол жеткізу)
   // . . .
}
```

A.12.1.1 `this` нұсқауышы

Егер функция-мүшені шақырған объектіге тікелей сілтеме жасағыңыз келсе, онда қордағы `this` нұсқауышын пайдалануыңызға болады:

```
struct Date {
   int d, m, y;
   int month() const { return this->m; }
   // . . .
};
```

`const` спецификаторы арқылы жарияланған функция-мүшені оны шақырған объект мүшесінің мәнін өзгерте алмайды:

```
struct Date {
    int d, m, y;
    int month() const { ++m; }  // error: month() is const
    // . . .
};
```

Константалық функция-мүшелер туралы толығырақ ақпаратты 9.7.4 бөлімінен қ.

A.12.1.2 Достастар

Класс мүшесі болып табылмайтын функция, егер ол **friend** түйінді сөзі арқылы жарияланса, кластың барлық мүшелеріне қол жеткізе алады (пайдалана алады). Мысал қарастырайық:

```
// Matrix және Vector members класс мүшелеріне
// қол жеткізуді талап етеді:
Vector operator*(const Matrix&, const Vector&);

class Vector {
    friend
    Vector operator*(const Matrix&, const Vector&);
    // қол жеткізді
    // . . .
};

class Matrix {
    friend
    Vector operator*(const Matrix&, const Vector&);
    // қол жеткізді
    // . . .
};
```

Жоғарыда көрсетілгендей, бұл әдетте екі класқа қол жеткізуге тиіс функцияларға қатысты болып табылады. **friend** сөзінің атқаратын қызметі басқа – функция-мүше тәрізді шақыруға болмайтын қатынасу функциясын қамтамасыз ету.

```
class Iter {
public:
    int distance_to(const iter& a) const;
    friend int difference(const Iter& a, const Iter& b);
    // . . .
};
void f(Iter& p, Iter& q)
{
    int x = p.distance_to(q); //  функция-мүшені шақыру
    int y = difference(p,q);  //  математикалық синтаксис
                                    арқылы шақыру
    // . . .
}
```

friend түйінді сөзі арқылы жарияланған функцияны виртуалды деп жариялауға болмайтынын айта кету керек.

A.12.2 Класс мүшелерін анықтау

Бүтінсандық константа, функция немесе тип болып келетін класс мүшелері *класс ішінде* де және *сыртында* да анықтала алады:

```
struct S {
    static const int c = 1;
    static const int c2;

    void f() { }
    void f2();

    struct SS { int a; };
    struct SS2;
};
```

Класта анықталмаған мүшелер "бір жерде" анықталуы тиіс:

```
const int S::c2 = 7;

void S::f2() { }

struct S::SS2 { int m; };
```

Статикалық константалық бүтінсандық класс мүшелері (**static const int**) ерекше жағдай болып табылады. Олар жай ғана символикалық бүтінсандық константаларды анықтайды да, объект алып тұрған компьютер жадында болмайды. Статикалық емес мәлімет-мүшелер жеке анықтаманы талап етпейді, олар класта жеке анықтала алмайды және инициалданбайды да.

```
struct X {
    int x;
    int y = 7;  // қате: статикалық емес мәлімет-мүшелер
                // класс ішінде инициалдана алмайды
    static int z = 7;  // қате: константа емес
      // мәлімет-мүшелер класс ішінде инициалдана алмайды
    static const string ae = "7";  // қате: бүтінсандық емес
             // типті класс ішінде инициалдауға болмайды
    static const int oe = 7;
    // ОК: статикалық константалық бүтін тип
};

int X::x = 7;      // қате: статикалық емес класс мүшелерін
                   // кластан тыс анықтауға болмайды
```

Егер сізге статикалық емес және константалық та емес мәлімет-мүшелерді инициалдау керек болса, конструкторларды қолданыңыз.

Функция-мүшелер объектіге бөлінген жады аймағын алмайды.

```
struct S {
int m;
void f();
};
```

Мұнда `sizeof(S)==sizof(int)`. Негізінде, бұл шарт стандартпен шектелмеген, бірақ тілдің барлық белгілі нұсқаларында ол орындалады. Виртуалды функциясы бар кластың виртуалды шақыруды қамтамасыз ететін бір жасырын мүшесі болады (14.3.1 бөлімді қ.).

A.12.3 Құру, өшіру және көшіру

Класс объектісін инициалдау мағынасын бір немесе бірнеше конструкторларды (constructors) анықтап алып барып анықтауға болады. *Конструктор* – бұл аты класс атымен бірдей, қайтаратын мәні жоқ функция-мүше.

```
class Date {
public:
    Date(int yy, int mm, int dd) :y(yy), m(mm), d(dd) { }
    // . . .
private:
    int y,m,d;
};

Date d1(2006,11,15); // ОК: конструктор арқылы инициалдау
Date d2;             // қате: инициалдау жоқ
Date d3(11,15);      // қате: дұрыс емес инициалдау
                     // (үш инициализатор керек)
```

Мәлімет-мүшелер конструктордағы инициалдау тізімі арқылы инициалдана алатынына назар аударыңыз. Класс мүшелері кластағы анықталу реттілігі бойынша инициалданады.

Конструкторлар әдетте класс инварианттарын орнату мен ресурстар алу үшін пайдаланылады (9.4.2 және 9.4.3 бөлімдерін қ.).

Класс объектілері базалық класс объектілерінен (14.3.1 бөлімін қ.) бастап, оларды жариялау реттілігіне қарай төменнен жоғары қарай құрылады. Сонан соң жариялану реттілігіне қарай класс мүшелері құрылады да, одан кейін конструктор кодының өзі орналасады. Егер програмалаушы түсініксіз бір нәрсе істеп қоймаса, бұл кластың әрбір объектісі өзі пайдаланылғанша, құрылатынына кепілдік береді.

Егер бір аргументі бар конструктор **explicit** түйінді сөзі арқылы жарияланбаса, онда ол өз аргументінің типін жанамалы түрде өз класына түрлендіруді анықтайды.

```
class Date {
public:
    Date(string);
    explicit Date(long); // use an integer encoding of date
    // . . .
};

void f(Date);

Date d1 = "June 5, 1848";           // OK
f("June 5, 1848");                  // OK

Date d2 = 2007*12*31+6*31+5; // error: Date(long) is explicit
f(2007*12*31+6*31+5);  // error: Date(long) is explicit

Date d3(2007*12*31+6*31+5);         // OK
Date d4 = Date(2007*12*31+6*31+5);  // OK
f(Date(2007*12*31+6*31+5));         // OK
```

Егер базалық кластар немесе туынды класс мүшелері тікелей аргументтерді талап етпесе және класта басқа конструкторлар болмаса, онда автоматты түрде келісім бойынша конструктор туындайды (генерацияланады). Бұл конструктор базалық кластың әрбір объектісін және келісім бойынша конструкторы бар (келісім бойынша конструкторы жоқ мүшелерді инициалдаусыз қалдыра отырып) әрбір объектіні инициалдайды. Мысал қарастырайық.

```
struct S {
    string name, address;
    int x;
};
```

Бұл **S** класының тікелей емес **S()** конструкторы бар, ол **x**-ті емес, **name** және **address** мүшелерін инициалдайды.

A.12.3.1 Деструкторлар

Объектінің өшіру операциясы (яғни объект көріну аймағы шегінен шыққан кезде не болатынын) мағынасын *деструктор* (destructor) көмегімен анықтауға

болады. Деструктор аты ~ символынан (толықтыру операторынан) және соған жалғасқан класс атынан тұрады.

```
class Vector {        // double типіндегі сандар векторы
public:
    explicit Vector(int s) : sz(s), p(new double[s]) { }
    // конструктор
    ~Vector() { delete[] p; }
    // деструктор
    // . . .
private:
    int sz;
    double* p;
};

void f(int ss)
{
    Vector v(s);
    // . . .
}
// f() функциясынан шыққан кезде v объектісі жойылады;
// ол үшін Vector класының деструкторы шақырылады
```

Класс мүшелерінің деструкторларын шақыратын деструкторларды компилятор туындатады. Егер класс базалық болып пайдаланылатын болса, онда оның витруалдық деструкторы болуы тиіс (17.5.2 бөлімін қ.).

Деструкторлар көбінесе "тазалау үшін" және ресурстарды босату үшін қолданылады.

Класс объектілері деструктордың өз кодынан бастап, жоғарыдан төмен қарай жойылады, олардан кейін жариялану реттілігі бойынша олардың мүшелері, сонан соң – жариялану реттілігі бойынша базалық класс объектілері, яғни оларды құру реттілігіне қарама-қарсы ретпен жойылады (А.12.3.1 бөлімін қ.).

А.12.3.2 Көшіру

Класс объектісінің көшіру негізін анықтауға болады:

```
class Vector {        // double типіндегі сандар векторы
public:
    explicit Vector(int s) : sz(s),p(new double[s]) { }
    //конструктор
    ~Vector() { delete[] p; }    // деструктор
```

```
    Vector(const Vector&);        // көшіретін конструктор
    Vector& operator=(const Vector&);
    // көшіретін меншіктеу
    // . . .
private:
    int sz;
    double* p;
};

void f(int ss)
{
    Vector v(s);
    Vector v2 = v;   // көшіретін конструкторды пайдаланамыз
    // . . .
    v = v2;         // көшіретін меншіктеуді пайдаланамыз
    // . . .
}
```

Келісім бойынша (яғни егер сіз көшіру конструкторы мен көшіретін меншіктеуді анықтамасаңыз) компилятордың өзі көшіру операцияларын туындатады. Келісім бойынша көшіру әрбір мүшелер бойынша орындалады (14.2.4 және 18.2 бөлімдерін қ.).

A.12.4 Туынды кластар

Класты басқа кластардан туынды класс түрінде анықтауға болады. Мұндай жағдайда ол өзі шыққан (өз базалық кластарын) класс мүшелерін мұралайды.

```
struct B {
    int mb;
    void fb() { };
};

class D : B {
   int md;
   void fd();
};
```

Мұнда `B` класының екі мүшесі бар: `mb` және `fb()`, ал `D` класының төрт мүшесі бар: `mb`, `fb()`, `md` және `fd()`.

Класс мүшелері сияқты базалық кластар ашық және жабық (`public` немесе `private`) түрде бола алады:

```
Class DD : public B1, private B2 {
        // . . .
};
```

Мұндайда **B1** класының ашық мүшелері **DD** класының ашық мүшелері болып шығады да, ал **B2** класының ашық мүшелері – **DD** класының жабық мүшелері болып шығады. Туынды кластың базалық кластың мүшелеріне қол жеткізуде ерекше артықшылықтары жоқ, сондықтан **DD** класының мүшелері **B1** мен **B2** класының жабық мүшелеріне қол жеткізе алмайды.

Егер кластың бірнеше тікелей базалық кластары (мысалы, **DD** класы сияқты) бар болса, онда ол *көпше мұралауды* (multiple inheritance) пайдаланады деп айтылады.

Егер **B** класы **D** класына қатысты қолжетімді және бір мәнді болып саналатын болса, онда **D** туынды класына нұсқауышты жанамалы түрде оның базалық класына нұсқауышқа түрлендіруге болады. Мысал қарастырайық.

```
struct B { };
struct B1: B { }; // B – B1 класына қатысты ашық базалық класс
struct B2: B { }; // B – B2 класына қатысты ашық базалық класс
struct C { };
struct DD : B1, B2, private C { };

DD* p = new DD;
B1* pb1 = p;  // OK
B* pb = p;    // қате: бірмәнді емес: B1::B немесе B2::B
C* pc = p;    // қате: DD::C – жабық класс
```

Осыған ұқсас, туынды класқа сілтемені жанамалы түрде бір мәнді және қолжетімді базалық класқа сілтемеге түрлендіруге болады.

Туынды кластар жайлы толығырақ ақпаратты 14.3 бөлімінен табуға болады. Қорғалған мұралауды (**protected**) сипаттау көптеген мұнан күрделірек оқулықтар мен анықтамалықтарда келтірілген.

A.12.4.1 Виртуалдық функциялар

Виртуалдық функция (virtual function) – бұл туынды кластардағы аттары бірдей және аргументтерінің типтері де бірдей фукцияларды шақырудың интерфейсін анықтайтын функция-мүше. Виртуалдық функцияны шақырғанда, ол кем дегенде, туынды кластардың бірінде анықталған болуы тиіс. Мұндайда туынды класс базалық кластың виртуалдық функция-мүшесін алмастырады (override) деп айтылады.

```
class Shape {
public:
  virtual void draw();
  // "virtual" деген "алмастырылуы мүмкін" дегенді білдіреді
  virtual ~Shape() { }     // виртуалдық деструктор
  // . . .
};

class Circle : public Shape {
public:
  void draw();     // Shape::draw функциясын алмастырады
  ~Circle();  // Shape::~Shape() функциясын алмастырады
  // . . .
};
```

Негізінде, базалық кластың виртуалдық функциясы (мұнда **Shape** класының) туынды кластың функцияларын шақыру интерфейсін (мұнда **Circle** класының) анықтайды.

```
void f(Shape& s)
{
    // . . .
    s.draw();
}

void g()
{
    Circle c(Point(0,0), 4);
    f(c);   // will call Circle's draw
}
```

f() функциясының **Circle** класы туралы ешнәрсе білмейтіндігіне назар аударыңыз: оған тек **Shape** класы белгілі. Виртуалдық функциясы бар класс объектісінің виртуалдық функциялардың жиынын (14.3 бөлімін қ.) табу мүмкіндігін беретін қосымша бір нұсқауышы бар.

Виртуалдық функциялары бар кластың көбінесе виртуалдық деструкторы болуы тиіс (мысалы, **Shape** класы тәрізді) екенін айта кетейік; 17.5.2 бөлімін қ.

A.12.4.2 Абстрактылық кластар

Абстрактылық класс (abstract class) – бұл тек базалық класс ретінде пайдалануға болатын класс. Абстрактылық кластың объектісін жасау мүмкін емес.

```
Shape s;    // қате: Shape абстрактылық класс болып табылады

class Circle : public Shape {
public:
    void draw();    // Shape::draw алмастырылады
    // . . .
};

Circle c(p,20); // OK: Circle класы абстрактылық класс емес
```

Абстрактылық класты құрудың ең кең таралған тәсілі болып, кем дегенде, бір *таза виртуалдық функцияны* (pure virtual function), яғни алмастыру талап етілетін функцияны анықтау саналады.

```
class Shape {
public:
    virtual void draw() = 0;
    // =0 "таза виртуалдық" дегенді білдіреді
    // . . .
};
```

14.3.5 бөлімді қ.

Сирегірек, бірақ тиімділігі бұдан төмен емес абстрактылық кластар олардың барлығының да конструкторларын қорғалған (protected) етіп жариялау жолымен құрылады. 14.2.1 бөлімін қ.

A.12.4.3 Туындаған операциялар

Кластарды анықтау кезінде олардың объектілерімен орындалатын кейбір операциялар келісім бойынша анықталады.

- Келісім бойынша конструктор.
- Көшіретін операциялар (көшіретін меншіктеу және көшіретін инициалдау).
- Деструктор.

Бұлардың әрқайсысы (келісім бойынша да) өз базалық кластары мен мүшелерінің әрқайсысына рекурсивті түрде қолданыла алады. Бұларды құру төменнен жоғары қарай жүргізіледі, яғни базалық кластың объектісі туынды кластың мүшесі құрылғанға дейін құрылады. Туынды кластың мүшелері және базалық кластың объектілері өздерінің жариялану реттілігі бойынша құрылады да, кері бағытта жойылады. Сонымен, конструктор мен деструктор әрқашанда

базалық кластар мен туынды класс мүшелерінің дәл анықталған объектілерімен жұмыс істейді. Мысал қарастырайық.

```
struct D : B1, B2 {
    M1 m1;
    M2 m2;
};
```

B1, **B2**, **M1** және **M2** кластары анықталған деп есептеп, келесі кодты жаза аламыз:

```
void f()
{
    D d;        // келісім бойынша инициалдау
    D d2 = d;   // көшіретін инициалдау
    d = D();    // келісім бойынша инициалдау,
                // сонан соң көшіретін меншіктеу
} // d мен d2 объектілері осы жерде жойылады
```

Мысалы, **d** объектісін келісім бойынша инициалдау төрт конструкторды келісім бойынша шақыру (көрсетілген тәртіппен) арқылы: **B1::B1()**, **B2::B2()**, **M1::M1()** және **M2::M2()** орындалады. Егер конструкторлардың бірі анықталмаған болса немесе шақырыла алмайтын болса, онда **d** объектісін құру мүмкін емес. **d** объектісін жою төрт деструкторды шақыру (көрсетілген тәртіппен) арқылы орындалады: **M2::~M2()**, **M1::~M1()**, **B2::~B2()** және **B1::~B1()**. Егер осы деструкторлардың бірі анықталмаған болса немесе шақырыла алмайтын болса, онда **d** объектісін жою мүмкін емес. Осы конструкторлар мен деструкторлардың әрқайсысын қолданушы анықтай алады немесе автоматты түрде туындайды.

Егер кластың қолданушы анықтаған конструкторы болса, онда тікелей емес (компилятор арқылы пайда болған) келісім бойынша конструктор анықталмаған (туындамайды) болып қалады.

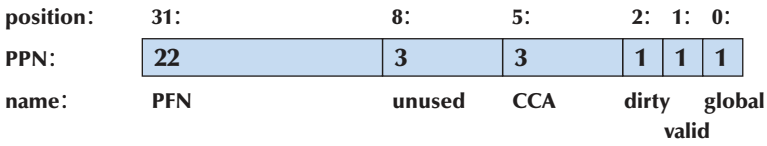
A.12.5 Биттік өрістер

Биттік өріс (bitfield) – бұл көптеген шағын мәндерді сөз түрінде немесе сырттан орнатылған биттік форматқа (мысалы, белгілі бір құрылғының регистрі форматына) сәйкес мәлімет түрінде сығу механизмі. Мысал қарастырайық.

```
struct PPN {
    unsigned int PFN : 22;
    int : 3;                    // пайдаланылмайды
    unsigned int CCA;
```

```
    bool nonreacheable;
    bool dirty;
    bool valid;
    bool global;
};
```

Биттік өрістерді солдан оңға қарай сөз түрінде сығу әрекеті келесі форматқа алып келеді (25.5.5 бөлімді қ.):



Биттік өрістің аты болуы міндетті емес, егер ол болмаса, онда оны пайдалану мүмкін емес.

Бір таңқаларлық нәрсе, көптеген шағын мәндерді бір жеке сөзге келтіріп сығу компьютер жадын кейде азайтпайды. Негізінде, осындай мәндердің бірін пайдалану `char` немесе `int` типін пайдалануға қарағанда, бір биттің өзін бейнелеуде де копьютер жадын артығырақ қолдануға алып келеді. Мұның себебі, басқа биттерді өзгертпей, бір сөзден бір бит шығарып алу үшін немесе сөзге бір бит енгізіп жазу үшін, бірнеше нұсқауларды орындау керек болады (олар да жады аймағынан орын алады). Егер сізде өте кіші мәліметтік өрістері бар объектілер саны аса үлкен болмаса, жады аймағын экономдау үшін биттік өрістер құруға тырыспаңыз.

A.12.6 Біріктірмелер

Біріктірмелер (union) – бұл барлық мүшелері жадының бір аймағында ғана орналасқан класс. Әрбір сәтте біріктірмеде тек бір элемент қана болады, мұнан тек соңғы жазылған элемент қана оқылады. Мысал қарастырайық.

```
union U {
    int x;
    double d;
}

U a;
a.x = 7;
int x1 = a.x;     // OK
a.d = 7.7;
int x2 = a.x;     // Ой!
```

Біріктірме мүшелерін сәйкестендірілген түрде оқу мен жазу ережелерін компилятор тексермейді. Біз сіздерге ескерттік.

A.13 Шаблондар

Шаблон (template) – бұл типтер жиынымен және/немесе бүтін сандармен параметрленген класс немесе функция.

```
template<class T>
class vector {
public:
    // . . .
    int size() const;
private:
    int sz;
    T* p;
};

template<class T>
int vector<T>::size() const
{
    return sz;
}
```

Шаблондық аргументтер тізімінде `class` түйінді сөзі типті білідіреді; оның өзіне парапар баламасы болып `typename` түйінді сөзі есептеледі. Келісім бойынша Шаблондық кластың функция-мүшесі болып кластағы сияқты дәл сол шаблондық аргументтер тізімі бар шаблондық функция есептеледі.

Бүтінсандық шаблондық аргументтер константалық өрнектер болуы тиіс.

```
template<typename T, int sz>
class Fixed_array {
public:
    T a[sz];
    // . . .
    int size() const { return sz; };
};

Fixed_array<char,256> x1;    // OK
int var = 226;
Fixed_array<char,var> x2;
// қате: константалық емес шаблондық аргумент
```

А.13.1 Шаблондық аргументтер

Шаблондық класс аргументтері оның аты пайдаланылған сайын көрсетіледі.

```
vector<int> v1;        // ОК
vector v2;   // қате: шаблондық аргумент жазылмай кеткен
vector<int,2> v3;  // қате: шаблондық аргументтер өте көп
vector<2> v4; // қате: шаблондық аргументтің типі күтіледі
```

Шаблондық функция аргументтері әдетте оның аргументтерінен шығарылады:

```
template<class T>
T find(vector<T>& v, int i)
{
    return v[i];
}

vector<int> v1;
vector<double> v2;
// . . .
int x1 = find(v1,2);        // мұндағы T – int типі
int x2 = find(v2,2);        // мұндағы T – double типі
```

Өзінің шаблондық аргументтерін шығаруға болмайтын шаблондық функцияны жариялауға болады. Мұндай жағдайда біз шаблондық аргументтерді тікелей нақтылап (дәл шаблондық кластардағы тәрізді етіп) алуымыз керек. Мысал қарастырайық:

```
template<classT,classU>T*make(constU&u){returnnewT(u);}
int* pi = make<int>(2);
Node* pn = make<Node>(make_pair("hello",17));
```

Бұл код тек `Node` класының объектісін `pair<const char *,int>` класының объектісімен инициалдауға болатын болса ғана жұмыс істейді (Б.6.3 бөлімін қ.). Шаблондық функцияны тікелей нақтылау механизмінен тек соңғы шаблондық аргументтерді (шығарылуға тиіс болатын) алып тастауға болады.

А.13.2 Шаблондарды нақтылау

Шаблондық аргументтердің нақты жиыны үшін жасалған шаблон нұсқасы *мамандандыру* (специализация – specialization) деп аталады. Шаблон және аргументтер жиыны негізінде мамандандыруларды туындату (генерациялау)

процесі *шаблондарды нақтылау* (template instantiation) деп аталады. Көбінесе бұл есепті компилятор шешеді, бірақ программааушы да өздігінен жеке мамандандыруды анықтай алады. Бұл әдетте жалпы шаблонды аргументтердің нақты жиыны үшін қолдануға болмайтын кезде жасалады. Мысал қарастырайық:

```
template<class T> struct Compare {     // жалпы салыстыру
    bool operator()(const T& a, const T& b) const
    {
        return a<b;
    }
};

template<> struct Compare<const char*> {
// С-тіркесті салыстыру
    bool operator()(const char* a, const char* b) const
    {
        return strcmp(a,b)==0;
    }
};

Compare<int> c2;                    // жалпы салыстыру
Compare<const char*> c;             // С-тіркесті салыстыру

bool b1 = c2(1,2);          // жалпы салыстыруды пайдалану
bool b2 = c("asd","dfg"); // С-тіркесті салыстыруды пайдалану
```

Функция үшін мамандандырудың аналогы болып асыра жүктеу есептеледі.

```
template<class T> bool compare(const T& a, const T& b)
{
       return a<b;
}

bool compare (const char* a, const char* b)
// С-тіркесті салыстыру
{
       return strcmp(a,b)==0;
}

bool b3 = compare(2,3);   // жалпы салыстыруды пайдалану
bool b4 = compare("asd","dfg");
// С-тіркесті салыстыруды пайдалану
```

Шаблондарды жеке компиляциядан өткізу (тақырыптық файлдарда тек жариялау бар да, ал бастапқы файлдарда – бірмәнді анықтаулар бар) программалардың ауысымдылығына кепілдеме бермейді, сондықтан, егер шаблонды әртүрлі бастапқы файлдарда пайдалану қажет болса, тақырыптық файлда оның толық анықтамасын беру керек.

A.13.3 Мүше-кластардың шаблондық типтері

Шаблонның типтер болып табылатын мүшелермен қатар типтер болып табылмайтын мүшелері де (мәлімет-мүшелер және функция-мүшелер тәрізді). Бұл негізінде, мүше аты типке қатысты ма, әлде жоқ па, соны айтудың қиын екенін білдіреді. Программалау тілінің ерекшеліктеріне байланысты техникалық себептерге орай компилятор мұны білуге тиіс, сондықтан біз оған кез келген бір жолмен осы ақпаратты беруіміз керек. Ол үшін **typename** түйінді сөзі қолданылады. Мысал қарастырайық:

```
template<class T> struct Vec {
    typedef T value_type;       // мүше типі
    static int count;           // мәлімет-мүше
    // . . .
};

template<class T> void my_fct(Vec<T>& v)
{
    int x = Vec<T>::count; // келісім бойынша мүшелер аттары
                           // типке қатысты болып саналмайды
    v.count = 7;      // тип болып саналмайтын мүшеге
                      // сілтеме жасаудың қарапайымдау тәсілі
    typename Vec<T>::value_type xx = x;
    // бұл жерге "typename" сөзі керек
    // . . .
}
```

Шаблондар туралы толығырақ ақпарат 19-тарауда келтірілген.

A.14 Аластамалар

Аластамалар шақыратын функцияға сол орында өңдеуге болмайтын қате кеткені туралы хабарлама жасау үшін пайдаланылады (**throw** нұсқауы арқылы). Мысалы, **Vector** класында **Bad_size** аластамасын шығару әрекетін көрсетейік:

```
struct Bad_size {
    int sz;
    Bad_size(int s) : ss(s) { }
};

class Vector {
    Vector(int s) { if (s<0 || maxsize<s) throw Bad_size(s); }
    // . . .
};
```

Көбінесе біз нақты қатені бейнелеу үшін арнайы анықталған типті туындатамыз. Шақыратын функция сол аластаманы ұстап алуы мүмкін:

```
void f(int x)
{
    try {
        Vector v(x);        // аластама туындатуы мүмкін
        // . . .
    }
    catch (Bad_size bs) {
        cerr << "Vector with bad size (" << bs.sz << ")\n";
        // . . .
    }
}
```

Барлық аластамаларды ұстау үшін `catch (...)` нұсқауын пайдалануға болады:

```
try {
     // . . .
} catch (...) {    // барлық аластамаларды ұстау
     // . . .
}
```

Көбінесе, тікелей `try` және `catch` нұсқауларын (19.5 бөлімін қ.) қолданғаннан гөрі RAII ("Resource Acquisition Is Initialization" – ресурстарды бөлу – бұл инициалдау") технологиясын қолданған жақсы (қарапайымдау, жеңілірек, сенімдірек).

Аргументсіз `throw` нұсқауы (яғни `throw;`) қайтадан ағымдағы аластаманы туындатады. Мысал қарастырайық:

```
try {
     // . . .
} catch (Some_exception& e) {
```

```
        // локальдік тазалау
        throw;     //  қалғанын шақырушы функция орындайды
    }
```

Аластама ретінде қолданушы анықтаған типтерді пайдалануға болады. Стандартты кітапханада пайдалануға болатын (Б.2.1 бөлімі) тағы бірнеше аластамалар типтері бар. Ешқашанда аластама ретінде құрамдас типтерді (оны тағы біреу жасауы мүмкін, сонда сіздің аластамаларыңыз шатастырып жібереді) қолданбаңыз.

Аластама туындатылған кезде, C++ тіліндегі программаны орындауды сүйемелдеу жүйесі стек бойынша жоғары қарай туындатылған объектінің типіне сәйкес типтегі **catch** бөлімін іздейді. Басқаша айтқанда, ол **try** нұсқауын аластама туындататын функция ішінен іздейді, сонан кейін, сәйкестікті тапқанша, аластама туындатқан функцияны шақырған функциядан іздейді. Егер сәйкестік табылмайтын болса, программа жұмысын тоқтатады. Осы жолда табылған әрбір функцияда және іздеу жүргізіліп жатқан әрбір көріну аймағында деструктор шақырылады. Осы процесс *стекті айналдыру* (stack unwinding) деп аталады.

Объект оның конструкторы жұмысын аяқтаған кезде құрылған болып есептеледі. Ол стекті айналдыру кезінде немесе өз көріну аймағынан әйтеуір бір шығу кезінде өшіріледі. Бұл жартылай жасалған объектілер (кейбір мүшелері мен базалық объектілері жасалған, ал кейбіреулері – жасалмаған объектілер), көріну аймағында орналасқан жиымдар мен айнымалылар дұрыс өңделетінін көрсетеді. Егер бұдан бұрын құрылған ішкі объектілер бар болған болса, олар өшіріледі.

Деструктордан шақыратын модульге берілетін аластама жасамаңыз. Басқаша айтқанда, деструктор ақаулық бермеуі тиіс. Мысал қарастырайық.

```
X::~X() {if (in_a_real_mess())throw Mess();}
// ешқашан бұлай жасамаңыз!
```

Бұл дракондық ереженің негізгі себебі мынаған саяды: егер деструктор стекті айналдыру барысында аластама туындататын болса (немесе өзі аластама ұстап алса), онда біз қандай аластаманы өңдеу керек екенін біле алмаймыз. Деструктордан шығу аластама туындату арқылы іске асатын жағдайлардан, барлық мүмкіндіктерді пайдалана отырып, алшақ болуға тырысу керек, өйткені бұл іске асатын жағдайда, оған дұрыс код жасаудың тұрақты бір тәсілі жоқ. Ашығын айтқанда, егер бұл орын алса, онда бір де бір функцияның немесе стандартты кітапханадан алынған кластың дұрыс жұмыс істеуіне кепілдік берілмейді.

А.15 Атаулар кеңістіктері

Атаулар кеңістігі (namespace) бір-бірімен байланысты жариялауларды біріктіреді де, атаулар қайшылығын (коллизиясын) болдырмайды.

```
int a;

namespace Foo {
    int a;
    void f(int i)
    {
        a+= i;      // that's  oo's a (  oo::a)
    }
}

void f(int);

int main()
{
    a = 7;          // that's the global a (::a)
    f(2);           // that's the global f (::f)
    Foo::f(3);      // that's  oo's f
    ::f(4);         // that's the global f (::f)
}
```

Аттарды, атаулардың ауқымды кеңістігіне жататын, олардың тікелей атаулар кеңістігіндегі аттарымен (мысалы, `Foo::f(3)`) немесе көріну аймағын шешу операторымен `::` (мысалы, `::f(2)`) нақтылап анықтауға болады.

Атаулар кеңістігіндегі барлық аттарды (мысалы, стандартты `std` кеңістігінде) келесі директивамен қол жетімді етуге болады:

```
using namespace std;
```

`using` директивасымен жұмыс істегенде, сақ болу керек. Оның беретін ыңғайлылығы аттардың әлеуеттік коллизиясы арқылы жүзеге асырылады. Анығын айтсақ, тақырыптық файлдарда `using` директиваларын пайдаланбауға тырысыңыз. Атаулар кеңістігіндегі жеке атты атаулар кеңістігін жариялау арқылы қолжетімді етуге болады.

```
using Foo::g;
g(2);
// Бұл Foo (Foo::g) атаулар кеңістігінен алынған g функциясы
```

Атаулар кеңістігі жайлы толығырақ ақпарат 8.7 бөлімінде келтірілген.

A.16 Баламалы атаулар

Атау үшін *баламалы (альтернативті) ат* (alias) анықтауға болады; басқаша айтқанда, сол аттың өзімен байланысқан (осы атты қолданатын көптеген жағдайларда) атты көрсететін символикалық ат анықтауға болады.

```
typedef int* Pint;        //  Pint means pointer to int

namespace Long_library_name { /* . . . */ }
namespace Lib = Long_library_name;
//  Lib means Long_library_name

int x = 7;
int& r = x;               // r means x
```

Сілтемелер (8.5.5 және A.8.3) – бұл программаның орындалуы барысында объектілерге нұсқау механизмі. **typedef** және **namespace** түйінді сөздері компиляциядан өткізу кезеңінде жұмыс істейтін атауларға сілтеме жасау механизміне жатады. Бөліп айтқанда, **typedef** нұсқауы жаңа тип енгізбейді, ол тек бұрыннан бар типке жаңа ат береді. Мысал қарастырайық.

```
typedef char* Pchar;      // Pchar is a name for char*
Pchar p = "Idefix";       // OK: p is a char*
char* q = p;              // OK: p and q are both char*s
int x = strlen(p);        // OK: p is a char*
```

A.17 Препроцессор директивалары

C++ тілінің әрбір нұсқасында *препроцессор* (preprocessor) бар. Негізінде, препроцессор компиляторға дейін жұмыс істейді де, біз жазған бастапқы кодты компилятор түзеген нұсқаға түрлендіреді. Нақтысын айтқанда, бұл әрекет компиляторда орындалады да, оның мәселе туындатуынан басқа ешқандай әсері болмайды. # символынан басталатын әрбір жол препроцессор директивасы болып табылады.

A.17.1 #include директивасы

Біз тақырыптық файлдарды іске қосу үшін препроцессорды кеңінен пайдаландық. Мысал қарастырайық.

```
#include "file.h"
```

Бұл директива препроцессорға `file.h` файлының ішкі мәліметтерін бастапқы мәтіннің сол директива тұрған нүктесіне қосуды бұйырады. Стандартты тақырыптар үшін қостырнақшалар `"..."` емес, бұрыштық жақшалар `<...>` қолданылады. Мысалы:

```
#include<vector>
```

Бұл стандартты тақырыптарды қосу үшін ұсынылған белгілеулер жүйесі.

A.17.2 #define директивасы

Препроцессор макроқойылым (macro substitution) деп аталып жүрген, символдармен орындалатын белгілі бір әрекеттерді (манипуляцияларды) де орындайды. Мысалы, символдық тіркес атын анықтайық.

```
#define FOO bar
```

Енді препроцессор барлық жерлерде `FOO` символдарын көрсе, оларды `bar` символдарымен алмастырады:

```
int FOO = 7;
int FOOL = 9;
```

Мұндағы жағдайда компилятор мынадай мәтінді көреді:

```
int bar = 7;
int FOOL = 9;
```

Препроцессор C++ тіліндегі атаулар жайлы көп нәрсе біледі, мысалы, ол `FOOL` сөзінің бөлігі болып табылатын `FOO` символдарын алмастырмайды.

`Define` директивасы арқылы параметрлер қабылдайтын макростарды да анықтауға болады.

```
#define MAX(x,y) (((x)>(y))?(x) : (y))
```

Оларды былай пайдалануға болады:

```
int xx = MAX(FOO+1,7);
int yy = MAX(++xx,9);
```

Бұл өрнектер мыналарға айналады:

```
int xx = (((bar+1)>( 7))?(bar+1) : (7));
int yy = (((++xx)>( 9))?(++xx) : (9));
```

Жақшалар **FOO+1** өрнегін есепеу кезінде дұрыс нәтиже алу үшін қажет екенін айта кетейік. Оған қоса, **xx** айнымалысы әдеттегідей емес тәсілмен екі рет инкременттеліп шықты. Макростар өте пайдалы, өйткені C тіліндегі программалаушылардың мұндай баламалы мүмкіндіктері аз болды. Қарапайым тақырыптық файлдарда мыңдаған макростардың анықтаулары сақталады. Сақ болыңыз!

Егер сізге макростарды пайдалану керек болса, оларды бас әріптер арқылы белгілеңіз, мысалы, **ALL_CAPITAL_LETTERS** ал, кәдімгі атаулар толығынан бас әріптерден тұрмауы тиіс. Жақсы кеңестерге құлақ салыңыз. Мысалы, бір беделді тақырыптық файлдардан біз **max** макросын таптық.

Тағы 27.8 бөлімін қ.

Стандартты кітапханаға шолу

"Мүмкіндігінше, барлық күрделілік тыс көзден жасырын болуы тиіс"

- Дэвид Дж. Уилер (David J. Wheeler)

Бұл қосымшада C++ тілінің стандартты кітапханасының негізгі мүмкіндіктеріне қысқаша шолу жасалған. Мұнда баяндалған ақпарат таңдаулы сипатта келтіріліген, ол стандартты кітапхананың мүмкіндіктері жайлы жалпы мағлұмат алғысы келетін және кітаптың негізгі мәтінінде жазылғандардан гөрі көбірек білуге талпынған жаңадан үйреніп жүргендерге арналған.

Б.1 Шолу

Бұл қосымша анықтамалық болып табылады. Оны жай тарау сияқты басынан аяғына дейін оқу міндетті емес. Мұнда жүйелі түрде (толығынан немесе жартылай) C++ тілінің түйінді элементтері сипатталған. Дегенмен, бұл толық анықтамалық емес; ол түйінді мүмкіндіктерді бейнелейтін аздаған мысалдар келтірілген қысқаша шолу түрінде берілген. Оқырмандар бұдан толығырақ ақпарат алу үшін осы кітаптың тақырыптарға сәйкес тарауларына жиі көңіл бөлулері керек. Оның үстіне, біз стандарт дәлдігіне ұмтылмадық та және оның терминологиясын да сақтауға тырыспағанымызды айта кетуіміз керек. Оқырмандар бұдан толығырақ ақпаратты Stroustrup, *The C++ Programming Language*[2] кітабынан таба алады. Тілдің толық анықталуы ISO C++ стандарты болып табылады, бірақ ол құжат жаңа үйреніп жүргендерге арналмаған және тілді алғаш рет үйренуге де ол сәйкес келе қоймайды. Және де Интернеттен қол жеткізуге болатын құжаттарды да ұмытпаңыз.

Таңдалып алынған (сол себепті толық емес) шолулардан қандай пайда табуға болады? Сіз белгілі бір операцияны жылдам іздеп таба аласыз немесе керекті операцияларды іздеу кезінде бөлімді жылдам шолып та шығуыңызға болады. Сіз өте толық мәліметті басқа дереккөздерден таба аласыз: бірақ нақты нені іздеу керек екендігін сізге осы қысқаша шолу айта алады. Бұл қосымшада басқа тараулардағы материалдарға кез келген орыннан сілтемелер жасалған және де стандартты кітапхананың мүмкіндіктері де қысқаша баяндалған. Мұндағы жазылғандарды есте сақтауға онша тырысудың қажеті жоқ, бұлар ол үшін келтірілмеген. Керісінше, бұл қосымша сізге артық заттарды есте сақтамауға мүмкіндік береді.

Мұнда сіз көптеген нәрселерді, өзіңіз толығынан жасағаннан гөрі, олардың дайындалған мысалдарын таба аласыз. Стандартты кітапханада бар заттардың барлығы да (әсіресе қосымшада келтірілгендерінің барлығы) көптеген адамдарға пайдалы болып табылады. Кітапхананың стандартты мүмкіндіктері практикалық тұрғыдан алғанда, сіздердің асығыс сәттерде құрғандарыңызға қарағанда, әлдеқайда өте тиянақты түрде жасалған, жүзеге асырылған және толыққанды құжаттармен де толықтырылған. Оған қоса, мұнда программалардың бір ортадан екінші ортаға ауысуы да өте жақсы ұйымдастырылған. Сонымен, мүмкіндігіне қарай, өзіңіз салған "құрқылтайдың ұясынан" (home brew) гөрі әрқашанда стандартты кітапханалық құралдарға басымдық берген дұрыс. Мұндай жағдайда, сіздің кодыңыз басқаларына қарағанда, түсініктірек болады.

Егер сіз сезімтал жан болсаңыз, онда мүмкіндіктердің осыншама молдығы сізді шошытуы да ықтимал. Қорықпаңыз, өзіңізге қажет емесін алып тастаңыз. Егер сіз қадағалауға шебер жан болсаңыз, онда біздің көп нәрселер жайлы айтпай кеткенімізді байқайсыз. Толықтық тек сарапшыларға арналған анықтамалықтарға және онлайн-құжаттамаларға тән болып саналады. Барлық жағынан алғанда да, көптеген нәрселер сізге жұмбақ тәрізді және қызықты да болып көрінуі мүмкін. Солардың сырын ашуға тырысыңыз!

---

[2] Страуструп Б. Язык программирования C++. Специальное издание. –М., СПб.: "Издательство БИНОМ" – Невский диалект, 2001. -1099 с.

Б.1.1 Тақырыптық файлдар

Стандартты кітапханадағы құралдар интерфейстері тақырыптарда анықталған. Келесі кестеде көрсетілген тақырыптардың кейбірі C++ тілінің 1998 жылы қабылданған ISO стандартына кірмей қалды. Дегенмен, олар келесі стандарттың бөлігі болатын болады және қазіргі кезде де кеңінен қолжетімді болып саналады. Мұндай тақырыптар "C++0x" болып белгіленген. Оларды пайдалану үшін жеке инсталляция және/немесе std атаулар кеңістігінен басқа атаулар (мысалы, tr1 немесе boost) кеңістігі қажет болуы мүмкін. Осы бөлімде сіздің программаңызда қандай құралдарға қол жеткізуге болатынын біле аласыз және де олардың қайда анықталып сипатталғанын да байқауыңызға болады.

STL кітапханасының тақырыптары (контейнерлер, итераторлар және алгоритмдер)	
`<algorithm>`	Алгоритмдер; `sort()`, `find()`, т.б. (Б.5, 21.1 бөлімдері)
`<array>`	Бекітілген мөлшердегі жиым (C++0x) (20.9 бөлім)
`<bitset>`	`bool` типіндегі жиым (25.5.2 бөлім)
`<deque>`	Екі жақты кезек
`<functional>`	Объект-функциялар (Б.6.2 бөлім)
`<iterator>`	Итераторлар (Б.4.4 бөлім)
`<list>`	Қос байланысты тізім (Б.4.4, 20.4 бөлім)
`<map>`	`map` және `multimap` ассоциативті контейнерлер (кілт, мән) (21.6.1-21.6.3 бөлімдер)
`<memory>`	Контейнерлер үшін компьютер жадын бөлушілер
`<queue>`	`queue` және `priority_queue` кластары
`<set>`	`set` және `multiset` кластары (21.6.5, Б.4 бөлімдер)
`<stack>`	`stack` класы
`<unordered_map>`	Хештелген ассоциативті жиымдар (C++0x) (21.6.4 бөлім)
`<unordered_set>`	Хештелген жиындар (C++0x)
`<utility>`	Операторлар және `pair` класы (Б.6.3 бөлімі)
`<vector>`	`vector` класы (динамикалық түрде кеңітілетін) (B.4, 20.8 бөлімдер)

Енгізу-шығару ағымдары	
`<iostream>`	Енгізу-шығару ағымдарының объектілері (Б.7 бөлімі)
`<fstream>`	Файлдық ағымдар (Б.7.1 бөлімі)
`<sstream>`	Тіркестік ағымдар (Б.7.1 бөлімі)

`<iosfwd>`	Енгізу-шығару ағымдары құралдарын жариялау (бірақ анықтау емес)
`<ios>`	Енгізу-шығару ағымдарының базалық кластары
`<streambuf>`	Ағымдық буферлер
`<istream>`	Енгізу ағымдары (Б.7 бөлімі)
`<ostream>`	Шығару ағымдары (Б.7 бөлімі)
`<iomanip>`	Форматтау және манипуляторлар (Б.7.6 бөлімі)

Тіркестермен манипуляциялар (орындалатын іс-әрекеттер)	
`<string>`	String класы. (Б.8.2 бөлімі)
`<regex>`	Регулярлық өрнектер (C++0x) (23 тарау)

Сандық тәсілдер	
`<complex>`	Алгоритмдер; `sort()`, `find()`, т.б. (Б.5, 21.1 бөлімдері)
`<random>`	Кездейсоқ сандарды туындату (генерациялау) (C++0x)
`<valarray>`	Сандық жиымдар
`<numeric>`	Жалпыланған сандық алгоритмдер, мысалы, `accumulate()` (Б.9.5 бөлімі)
`<limits>`	Шектік мәндер (Б.9.1 бөлімі)

Қосалқы құралдар және тілді сүйемелдеу	
`<exception>`	Аластамалар типтері (Б. 2.1 бөлімі)
`<stdexcept>`	Аластамалар иерархиясы (Б.2.1. бөлімі)
`<locale>`	Жергілікті ерекшеліктерге байланысты форматтау
`<typeinfo>`	Типтер туралы стандартты информация (`typeid` операторы)
`<new>`	Компьютер жадын бөлу және босату үшін функция

C тілінің стандартты кітапханасы	
`<cstring>`	C тілі стилінде тіркестермен іс-әрекеттер (Б.10.3 бөлімі)
`<cstdio>`	C тілі стилінде енгізу-шығару (Б.10.2. бөлімі)
`<ctime>`	`clock()`, `time()`, т.б. функциялар (Б.10.5. бөлімі)
`<cmath>`	Жылжымалы нүктелі сандармен жұмыс істеу үшін стандартты математикалық функциялар (Б.9.2. бөлімі)
`<cstdlib>`	`abort()`, `abs()`, `malloc()`, `qsort()`, т.б. функциялар (27-тарау)
`<cerrno>`	C тілі стилінде қателерді өңдеу (24.8 бөлімі)
`<cassert>`	`assert` макросы (27.9 бөлімі)

С тілінің стандартты кітапханасы (жалғасы)	
`<clocale>`	Жергілікті ерекшеліктерге байланысты форматтау
`<climits>`	С тілі стиліндегі шектік мәндер (Б.9.1 бөлімі)
`<cfloat>`	Жылжымалы нүктелі сандардың шектік мәндері (Б.9.1 бөлімі)
`<cstddef>`	С тілін сүйемелдеу; `size_t`, т.б.
`<cstdarg>`	Айнымалы аргументтерді өңдеу үшін макростар
`<csetjmp>`	`stejmp()` және `longjmp()` функциялары (ешқашанда бұларды пайдаланбаңыз)
`<csignal>`	Сигналдарды өңдеу
`<cwchar>`	Кеңейтілген символдар
`<cctype>`	Символдық типтердің жіктелуі (Б.8.1 бөлімі)
`<cwctype>`	Кеңейтілген символдық типтердің жіктелуі

С тілі стандартты кітапханасының әрбір тақырыбы үшін осы сияқты тақырыптық файлдар бар, тек олардың атында бірінші `c` әрпі жоқ және кеңейтілулері `.h` болып келеді, мысалы, `<ctime>` тақырыбы үшін `<time.h>` тақырыптық файлы. Тақырыптың `.h` жалғауы бар нұсқалары `std` атаулар кеңістігінде атауларды емес, олар ауқымды атауларды анықтайды.

Осы тақырыптарда анықталған құралдардың барлығы емес, тек кейбіреулері кітаптың келесі бөлімдері мен тарауларында сипатталған. Егер сізге толығырақ ақпарат қажет болса, онда онлайн-құжаттамаларды немесе сараптамалық деңгейдегі C++ тіліне арналған кітаптарды қараңыз.

Б.1.2 `std` атаулар кеңістігі

Стандартты кітапхананың құралдары `std` атаулар кеңістігінде анықталған, сондықтан оларды пайдалану үшін солардың тікелей біліктілігін (квалификациясын) `usig` жариялануы немесе `usig` директивасын беру арқылы орындау керек.

```
std::string s;                  // тікелей квалификация

using std::vector;              // using жариялануы
vector<int>v(7);

using namespace std;            // using директивасы
map<string,double> m;
```

Бұл кітапта **std** атаулар кеңістігіне қол жеткізу үшін біз **using** директивасын пайдаландық. **using** директивасын қолдану кезінде сақ болыңыздар (А.15 бөлімін қ.)

Б.1.3 Сипаттау стилі

Стандартты кітапханадағы ең қарапайым операцияның толық сипатталуының өзі, мысалы, конструктордың немесе алгоритмнің сипатталуы бірнеше парақты алып тұрады. Сол себепті біз бейнелеудің ең икемді қысқаша стилін пайдаланамыз. Мысал қарастырайық.

Сипаттау мысалы	
`p=op(b,e,x)`	**op** функциясы **[b:e)** диапазонымен және **x** айнымалысымен p айнымалысын қайтара отырып, кейбір әрекеттерді орындайды
`foo(x)`	**x** айнымалысымен бірдеңе орындайды да, бірақ ешқандай нәтиже қайтармайды
`bar(b,e,x)`	**x** объектісі **[b:e)** диапазонымен бір нәрсе жасауы керек пе?

Біз мнемоникалық идентификаторларды таңдауға тырыстық, сондықтан **b**, **e** символдары диапазонның басы мен соңын беретін итераторларды белгілейтін болады; **p** – нұсқауыш немесе итератор; **x** – мәтінге толық тәуелді түрде алынған бір мән. Осы белгілеулер жүйесінде ешқандай мән қайтармайтын функцияны бульдік типтегі мән қайтаратын айнымалыдан, қосымша түсініктеме бермей, ажыратып алу қиын, сондықтан қосымша бірдеңелер жасалмаса, оларды шатастырып алуға болады. **Bool** типіндегі айнымалы қайтаратын операциялар үшін түсініктеме беруде әдетте сұрақ белгісі тұрады.

Егер алгоритмдер "ақау", "табылмады", т.с.с. оқиғаларды белгілеу үшін кіріс тізбегінің соңын қайтаратын жалпы бекітілген келісімдерге (Б.3.1 бөлімі) сай құрылса, біз оны тікелей айқындап көрсетпейміз.

Б.2 Қателерді өңдеу

Стандартты кітапхана қырық жылдай мерзім бойынша жасалған компоненттерден тұрады. Сол себепті оның қателерін өңдеу стилі мен қағидалары келісілмеген түрде болып табылады.

- C тілі стиліндегі кітапхана қателерді көрсету үшін **errno** жалауын орнататын функциялардан тұрады (24.8 бөлімі).
- Көптеген алгоритмдер элементтер тізбегі үшін соңғы элементтерін кейінгі элементке орнатылған итераторды қайтарады, бұл қате шыққанының немесе ізделген элементтің табылмағанын көрсетеді.

- Енгізу-шығару ағымдары кітапханасы қателер туралы мәлімдемелер беру үшін әрбір ағымның қалып-күйін пайдаланады да, аластама туындата (егер қолданушы осыны талап етсе) алады (10.6 және Б.7.2 бөлімдерін қ.).
- Стандартты кітапхананың `vector`, `string` және `bitset` сияқты кейбір компоненттері қате байқалған сәтте аластама туындатады.

Стандартты кітапхана оның барлық құралдары базалық шарттарды қанағаттандыратындай етіп жасалған (19.5.3 бөлімін қ.). Басқаша айтқанда, тіпті аластама туындаса да, бір де бір ресурс (мысалы, жады) жоғалмайды және стандартты кітапхана класының бір де бір инварианты бұзылмайды да.

Мысалы, 12 литералы он екі бүтін санын көрсетсе, `"Morning"` литералы – *Morning* сиволдық тіркесін, ал `true` литералы *true* бульдік мәнін бейнелейді.

Б.2.1 Аластамалар

Стандартты кітапхананың кейбір құралдары аластама туындата отырып, қателер туралы мәлімдемелер береді.

Стандартты аластамалар	
`bitset`	`invalid_argument`, `out_of_range`, `overflow_error` аластамаларын туындатады
`dynamic_cast`	Егер түрлендіру жасай алмаса, `bad_cast` аластамасын туындатады
`iostream`	Егер түрлендіруге рұқсат болса, `ios_base::failure` аластамасын туындатады
`new`	Егер жады бөле алмаса, `bad_alloc` аластамасын туындатады
`regex`	`regex_error` аластамасын туындатады
`string`	`length_error` және `out_of_range` аластамаларын туындатады
`typeid`	Егер `type_info` объектісін қайтара алмаса, `bad_typeid` аластамасын туындатады
`vector`	`out_of_range` аластамасын туындатады

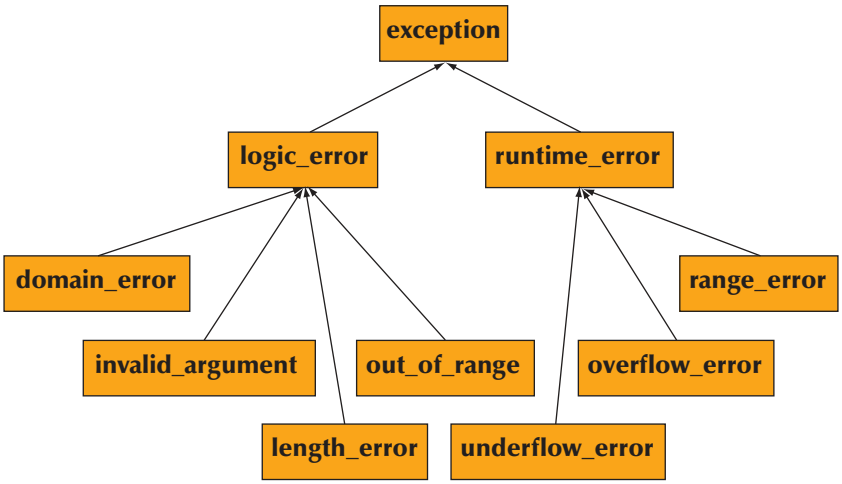
Бұл аластамалар кітапхананың осы көрсетілген құралдарын тікелей немесе жанамалы түрде пайдаланатын кез келген кодта пайда бола алады. Егер сіз барлық пайдаланылған құралдар дұрыс қолданылғанына сенімді болсаңыз және сол себепті аластамалар туындата алмасаңыз, онда әрқашанда программаның бір жерінде (мысалы, `main()` функциясында) стандартты кітапханадағы аластамалар иерархиясының (мысалы, `exception`) түпкі кластарының бірінің объектілерін ұстап алған дұрыс болып саналады.

Біз құрамдас типтер аластамаларын, мысалы, **int** типіндегі санның немесе C тілі стиліндегі тіркестің аластамасын туындатпауды тинақты түрде ұсынамыз. Оның орнына аластама ретінде пайдалану үшін арнайы жасалған типтер объектілерін туындату қажет. Ол үшін **exception** стандартты кітапханалық класынан туынды класты пайдалануға болады.

```
class exception {
public:
     exception();
     exception(const exception&);
     exception& operator=(const exception&);
     virtual ~exception();
     virtual const char* what() const;
};
```

what() функциясын аластаманы туындатқан қате туралы ақпаратты бейнелеуге арналған тіркесті алу үшін пайдалануға болады.

Төменде келтірілген стандартты аластамалар иерархиясы сізге аластамаларды (классификациялауға) жіктеуге көмектесе алады.

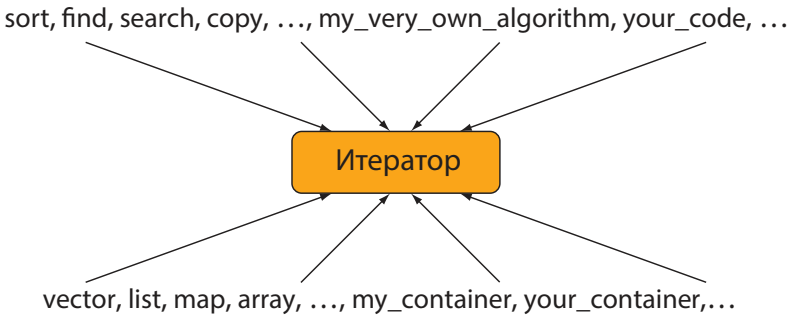


Стандартты кітапханалық аластамадан шығара отырып, аластаманы келесідей түрде анықтай аласыз:

```
struct My_error : runtime_error {
     My_error(int x) : interesting_value(x) { }
     int interesting_value;
     const char* what() const { return "My_error"; }
};
```

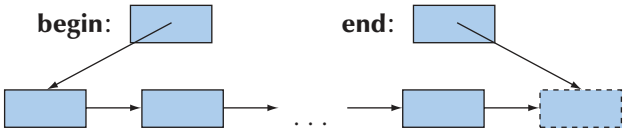
Б.3 Итераторлар

Итераторлар – бұл стандартты кітапхана алгоритмдерін солардың мәліметтерімен қоса отырып біріктіретін желім. Итераторларды алгоритмдердің олар амалдар орындайтын мәліметтер құрылымынан тәуелсіздігін төмендететін (кішірейтетінін) механизм деп атауға болады (20.3 бөлімін қ.).



Б.3.1 Итераторлар моделі

Итератор – бұл нұсқауыш аналогы, мұнда жанамалы түрде қол жеткізу (мысалы, атаусыз ету `*` операторы) және жаңа элементке көшу (мысалы, келесі элементке көшу үшін `++` операторы) операциялары жүзеге асырылған. Элементтер тізбегі жартылай ашық диапазонды `[begin:end)` беретін итераторлар жұбымен анықталады.



Басқаша айтқанда, `begin` итераторы тізбектің алғашқы элементіне сілтеме жасайды, ал `end` итераторы – тізбектің ең соңғы элементінен кейінгі элементке сілтеме жасайды. Ешқашанда `*end` мәнін оқымаңыз және жазбаңыз. Бос тізбек үшін әрқашанда `begin== end` шарты орындалады. Басқаша сөзбен айтқанда, кез келген p итераторы үшін `[p:p)` тізбегі бос болып табылады.

Тізбекті оқу үшін алгоритм әдетте `(b,e)` итератолар жұбын алады да, `++` операторы арқылы оның соңына жеткенше элементтер бойынша жылжып отырады.

```
while (b!=e) {  // < таңбасын емес, != тіркесін пайдаланыңыз
    // кейбір операциялар
    ++b;          // соңғы элементке көшеміз
}
```

Тізбекте элемент іздеу ісін атқаратын алгоритмдер, нәтижесі табыссыз болса, тізбек соңына орнатылған итераторды қайтарады. Мысал қарастырайық.

```
p=find(v.begin(),v.end(),x); // v тізбегінен x-ті іздейміз
if (p!=v.end()) {
     // x  p ұясынан табылған
}
else {
     // [v.begin():v.end()) диапазонынан x табылмаған
}
```

20.3 бөлімді қ.

Тізбек элементтерін жазатын алгоритмдер көбінесе тек оның бірінші элементіне орнатылған итераторды алады. Мұндайда программалаушының өзі осы тізбек шегінен шығып кетуді болдырмауы тиіс. Мысал қарастырайық.

```
template<class Iter> void f(Iter p, int n)
{
     while (n>0) *p = -- n;
}

vector<int> v(10);
f(v.begin(),v.size());        // OK
f(v.begin(),1000);            // үлкен мәселе
```

Стандартты кітапхананың кейбір жүзеге асырылған нұсқалары берілген диапазоннан шығып кетуді `f()` функциясын соңғы шақыру кезінде тексереді, яғни аластама туындатады, бірақ бұл кодты ауысымды деп санауға болмайды; көптеген нұсқалар мұндай тексеруді жасамайды.

Итераторлармен орындалатын операцияларды тізіп шығайық.

Итераторлармен орындалатын операциялар	
`++p`	Префикстік инкременттеу: `p` итераторын тізбектің келесі элементіне немесе соңғы элементтен кейінгі элементке ("бір элемент алға") орнатады; нәтижесі `p+1` мәні болып табылады.
`p++`	Постфикстік инкременттеу: `p` итераторын тізбектің келесі элементіне немесе соңғы элементтен кейінгі элементке ("бір элемент алға") орнатады; нәтижесі `p` мәні (инкременттеуге дейін) болып табылады.
`--p`	Префикстік декременттеу: `p` итераторын тізбектің алдыңғы элементіне ("бір элемент артқа") орнатады; нәтижесі `p-1` мәні болып табылады.

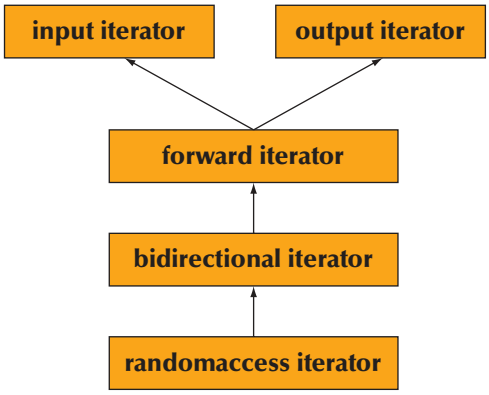
Итераторлармен орындалатын операциялар (*жалғасы*)	
`p--`	Постфикстік декременттеу: `p` итераторын тізбектің алдыңғы элементіне ("бір элемент артқа") орнатады; нәтижесі `p` мәні (декременттеуге дейін) болып табылады.
`*p`	Қол жеткізу (атаусыз ету): `*p` мәні `p` итераторы көрсетіп тұрған элементке қатысты болады.
`p[n]`	Қол жеткізу (индекстеу): `p[n]` мәні `p+n` итераторы көрсетіп тұрған элементке қатысты болады; `*(p+n)` өрнегінің эквиваленті.
`p->m`	Қол жеткізу (мүшеге қол жеткізу): `(*p).m` өрнегінің эквиваленті.
`p==q`	Теңдік: егер `p` және `q` итераторлары бір элементке сілтеме жасап тұрса немесе екеуі де соңғы элементтен кейінгі элементке сілтеме жасап тұрса, ақиқат болады.
`p!=q`	Теңсіздік: `!(p==q)`.
`p<q`	`p` итераторы `q` итераторы көрсетіп тұрған элементке дейін орналасқан элементті көрсетіп тұр ма?
`p<=q`	`p<q || p==q`
`p>q`	`p` итераторы `q` итераторы көрсетіп тұрған элементтен кейін орналасқан элементті көрсетіп тұр ма?
`p>=q`	`p<q || p==q`
`p+=n`	`n` элемент алға: `p` итераторын қазір ол көрсетіп тұрған элементтен алға қарай `n`-элементке орнатады.
`p-=n`	`-n` элемент алға: `p` итераторын қазір ол көрсетіп тұрған элементтен артқа қарай `n`-элементке орнатады.
`q=p+n`	`q` итераторы `p` итераторы сілтеме жасап тұрған элементтен алға қарай `n`-элементке сілтеме жасап тұр.
`q=p-n`	`q` итераторы `p` итераторы сілтеме жасап тұрған элементтен артқа қарай `n`-элементке сілтеме жасап тұр; ол орындалған соң `q+n==p`
`advance(p,n)`	Алға жылжу: `p+=n` өрнегінің аналогы; `advance(p,n)` функциясын `p` итераторы кез келген бағытта қол жеткізу итераторы болмаса да пайдалануға болады; бұл операция барлық `n` қадамды (тізім бойынша) орындай алады.
`x=distance(p,q)`	Айырмасы: `q-p` өрнегінің аналогы; `distance()` функциясын `p` итераторы кез келген бағытта қол жеткізу итераторы болмаса да пайдалануға болады; бұл операция барлық `n` қадамды (тізім бойынша) орындай алады.

Б.3.2 Итераторлар санаттары

Стандартты кітапханада итераторлардың бес түрі қарастырылған.

Итераторлар санаттары (категориялары)	
`input iterator`	`++` операторының көмегімен алға жылжып, әрбір элементті тек бір рет қана `*`операторы арқылы оқи аламыз. Итераторларды `==` және `!=` операторлары арқылы салыстыруға болады. Итераторлардың бұл түрі `istream` класы түрінде жүзеге асырылған (21.7.2 бөлімін қ.).
`output iterator`	`++` операторының көмегімен алға жылжып, әрбір элементті тек бір рет қана `*`операторы арқылы жаза аламыз. Итераторлардың бұл түрі `ostream` класы түрінде жүзеге асырылған (21.7.2 бөлімін қ.).
`forward iterator`	`++` операторын қайта қолдана отырып, алға жылжи аламыз және де `*`операторы арқылы элементтерді жаза және оқи аламыз (егер олар константалық болмаса). Егер итератор класс объектісін көрсетіп тұрса, онда оның мүшесіне қол жеткізу үшін `->` операторын қолдануға болады.
`bidirectional iterator`	Алға (`++` және `--` операторларын қолдана отырып) және артқа (`--`және `-=` операторларын қолдана отырып) жылжи аламыз және де `*` операторы арқылы элементтерді жаза және оқи аламыз (егер олар константалық болмаса). Итераторлардың бұл түрі `list, map` және `set` кластарында жүзеге асырылған.
`random-access iterator`	Алға (`++` және `+=` операторларын қолдана отырып) және артқа (`--`және `-=` операторларын қолдана отырып) жылжи аламыз және де `*` немесе `[]` операторы арқылы элементтерді жаза және оқи аламыз (егер олар константалық болмаса). Біз индекстеуді қолдана аламыз, кез келген бағытта қол жеткізуге болатын итераторға `+` операторы арқылы бүтін сан қоса аламыз және де `-` операторы арқылы одан бүтін сан азайта аламыз. Біз екеуі де бір тізбекке орнатылған кез келген бағытта қолжетімді екі итератор арасындағы қашықтықты олардың бірін екіншісінен азайта отырып есептеп шығара аламыз. Кез келген бағытта қол жеткізуге болатын итераторларды `<`, `<=`, `>`, `>=` операторлары арқылы салыстыруға болады. Итераторлардың бұл түрі `vector` класында жүзеге асырылған.

Логикалық тұрғыдан алғанда, итераторлар иерархия құрайды (20.8 бөлімін қ.):



Итераторлар санаттары класс болып табылмайтындықтан, бұл иерархияны мұралау арқылы жүзеге асырылған кластар иерархиясы деп санауға болмайды. Егер сізге итераторлар арқылы онша қарапайым емес әрекет орындау керек болса, кәсіби анықтамалықтан `iterator_traits` класын іздеңіз.

Әрбір контейнердің нақты санаттардағы жеке итераторлары бар:

- `vector` – кез келген бағыттағы қол жетімді итераторлар;
- `list` – қос бағытты итераторлар;
- `deque` – кез келген бағыттағы қол жетімді итераторлар;
- `bitset` – итераторлар жоқ;
- `set` – қос бағытты итераторлар;
- `multiset` – қос бағытты итераторлар;
- `map` – қос бағытты итераторлар;
- `multimap` – қос бағытты итераторлар;
- `unordered_set` – бір бағытты итераторлар;
- `unordered_multiset` – бір бағытты итераторлар;
- `unordered_map` – бір бағытты итераторлар;
- `unordered_multimap` – бір бағытты итераторлар.

Б.4 Контейнерлер

Контейнерде элементтер тізбегі болады. Бұл тізбек элементтерінің типі `value_type`. Ең пайдалы контейнерлер болып келесілер есептеледі.

Тізбекті контейнерлер	
`array<T,N>`	Бекітілген өлшемдегі `n` элементтерден тұратын `T` типіндегі жиым (C++0x)
`deque<T>`	Екі жақты кезек
`list<T>`	Қосбайланысты тізім
`vector<T>`	`T` типіндегі элементтердің динамикалық жиымы

Ассоциативтік контейнерлер	
`map<K,V>`	`K` типіндегі элементтерді `V` типіндегі элементтерде бейнелеу; жұптар тізбегі (K,V).
`multimap<K,V>`	K-дан V-ға бейнелеу; кілттер дубликатына рұқсат етілген.
`set<K>`	`K` типіндегі элементтер жиыны.
`multiset<K>`	`K` типіндегі элементтер жиыны (дубликаттарға рұқсат етілген).
`unordered_map<K,V>`	`K` типіндегі элементтерді `V` типіндегі элементтерде бейнелеу; хештеу функциясы арқылы (C++0x).
`unordered_multimap<K,V>`	`K` типіндегі элементтерді `V` типіндегі элементтерде хештеу функциясы арқылы бейнелеу; кілттер дубликатына рұқсат етілген. (C++0x).
`unordered_set<K>`	хештеу функциясы бар `K` типіндегі элементтер жиыны; (C++0x).
`unordered_multiset<K>`	хештеу функциясы бар `K` типіндегі элементтер жиыны; кілттер дубликатына рұқсат етілген (C++0x).

Реттелген ассоциативті контейнерлердің (`map`, `set`, т.б) қосымша шаблондардың аргументі бар, ол салыстыру үшін қолданылады; мысалы, `set<K,C>`, C тілін `K` мәндерін салыстыру үшін қолданады.

Контейнерлер адаптерлері	
`priority_queue<T>`	Приоритеті (басымдылығы) бар кезек
`queue<T>`	`push()` және `pop()` функциялары бар кезек
`stack<T>`	`push()` және `pop()` функциялары бар стек

Бұл контейнерлер `<vector>`, `<list>`, т.б. кластарда анықталған (Б.1.1 бөлімін қ.). Тізбекті контейнерлер компьютер жадының үздіксіз аймағында орналасады немесе `value_type` (жоғарыда біз оны `T` әрпімен белгілегенбіз) сәйкес типіндегі элементтері бар байланысқан тізімдер түрінде болады. Ассоциативтік контейнерлер `value_type` сәйкес типіндегі түйіндері бар байланысқан құрылымдар (бұтақтар) түрінде болады (жоғарыда біз оны `pair(K,V)` түрінде белгілегенбіз). `set`, `map` немесе `multimap` контейнерлеріндегі элементтер тізбегі кілт (`K`) бойынша рет-

телген. Контейнерлердегі аттары **unordered** сөзінен басталатын тізбектердің кепілдеме берілген тәртібі жоқ. **multimap** контейнері **map** контейнерінен оның кілтінің мәні бірнеше рет қайталанатындығымен айрықшаланады. Контейнерлер адаптерлері – бұлар басқа контейнерлерден жасалған арнайы операциялары бар контейнерлер.

Егер күмәндансаңыз, **vector** класын пайдаланыңыз. Егер сіздің басқа контейнерді пайдалануға нақты себебіңіз болмаса, **vector** класын пайдаланыңыз.

Контейнерлер компьютер жадын бөлу және босату үшін жады бөлгіштерді қолданады. Біз мұнда оларды сипаттамаймыз; қажет болса, оқырмандар олар жайлы ақпаратты кәсіби анықтамалықтардан тауып алады. Контейнер элементтері үшін қажетті компьютер жадын алу немесе босату үшін, келісім бойынша, жады бөлгіштер **new** және **delete** операторларын пайдаланады.

Керекті жерлерінде қолдану үшін қол жеткізу операциясы екі нұсқада жүзеге асырылған: біріншісі – константалық объектілер үшін, екіншісі – константалық емес объектілерге арналған (18.4 бөлімін қ.).

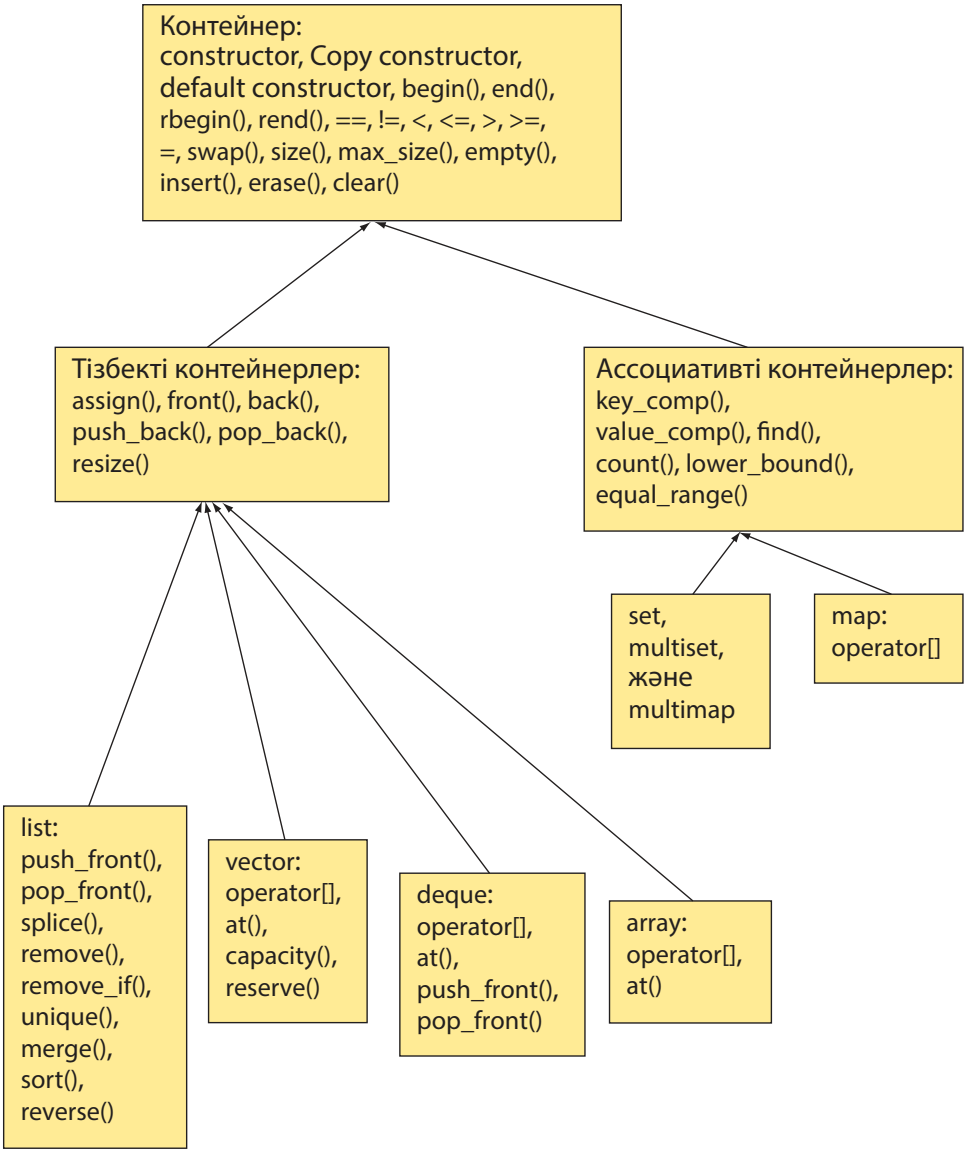
Бұл бөлімде стандартты контейнерлердің жалпы және "жалпыға жақын" мүшелері көрсетілген (толығырақ ақпаратты 20 тараудан қ.). Нақты бір контейнерге тән мүшелер, мысалы, **list** класындағы **splice()** функциясы сияқтылар, көрсетілмеген; олардың сипаттамаларын кәсіби анықтамалықтардан табуға болады.

Мәліметтердің кейбір типтері стандартты контейнерден талап етілетін көптеген операцияларды, бірақ барлығын емес, қамтамасыз етеді. Кейде мұндай типтерді "контейнерлерге жақын" деп атайды. Солардың ең қызықтыларын тізіп көрсетейік.

"Контейнерлерге жақындары"	
T[n] құрамдас жиым	**size()** функциясы және басқа да функция-мүшелері жоқ. Егер таңдауға мүмкіндік болса, балама ретінде **vector**, **string** немесе **array** сияқты контейнерлерін пайдалануды ұсынамыз.
string	Тек символдары бар, бірақ мәтінмен жұмыс істеуге пайдалы операцияларды, мысалы, конкатенацияны (**+** және **+=**) қамтамасыз етеді. Жай тіркестер орнына стандартты **string** класын пайдалануды ұсынамыз.
valarray	Векторлық операциялары бар сандық вектор, бірақ жоғары өнімділікке қол жеткізу жолында енгізілген көптеген шектеулері бар. Егер өте көптеген арифметикалық операцияларды орындау керек болса, пайдалануға ұсынамыз.

Б.4.1 Шолу

Стандартты контейнерлерде қарастырылған операцияларды келесідей түрде бейнеленеді:



Б.4.2 Мүшелер типтері

Контейнер оның мүшелері типтерінің жиынын анықтайды.

	Мүшелер типтері
`value_type`	Элементтер типі
`size_type`	Индекстер, элементтер санауыштары және т.б. типі
`difference_type`	Итераторлар арасындағы айырма типі
`iterators`	`value_type*` нұсқауышының аналогы
`const_iterator`	`const value_type*` нұсқауышының аналогы
`reverse_iterator`	`value_type*` нұсқауышының аналогы
`const_reverse_iterator`	`const value_type*` нұсқауышының аналогы
`reference`	`value_type&`
`const_reference`	`const value_type&`
`pointer`	`value_type*` нұсқауышының аналогы
`const_pointer`	`const value_type*` нұсқауышының аналогы
`key_type`	Кілт типі (тек ассоциативтік контейнерлер үшін)
`mapped_type`	Бейнеленген мән типі (тек ассоциативтік контейнерлер үшін)
`key_compare`	Салыстыру үшін критерий типі (тек ассоциативтік контейнерлер үшін)
`allocator_type`	Жады менеджері типі

Б.4.3 Конструкторлар, деструктолар және меншіктеу

Контейнерлердің көптеген әртүрлі конструкторлары мен меншіктеу операторлары бар. С контейнерлері үшін конструкторларды, деструкторларды және меншіктеу операторларын тізіп көрсетейік (мысалы, `vector<double>` немесе `map<string,int>` типі).

Конструкторлар, деструкторлар және меншіктеу операторлары	
`C c;`	`c` – бос контейнер
`C()`	Бос контейнер жасайды
`C c(n);`	`c` контейнері мәндері келісім бойынша берілген `n` элементтермен инициалданады (ассоциативтік контейнерлер үшін емес)
`C c(n,x);`	`c` контейнері `x` объектісінің `n` көшірмелерімен инициалданады (ассоциативтік контейнерлер үшін емес)

Конструкторлар, деструкторлар және меншіктеу операторлары *(жалғасы)*	
`C c(b,e);`	`c` контейнері `[b:e)` диапазонының элементтерімен инициалданады
`C c(c2);`	`c` контейнері `c2` контейнерінің көшірмесі болып табылады
`~C()`	`C` типіндегі контейнерді және оның барлық элементтерін жою (әдетте жанамалы түрде шақырылады)
`c1=c2`	Көшіретін меншіктеу; `c2` контейнерінен барлық элементтерді `c1` контейнеріне көшіреді; меншіктеуден соң `c1==c2` шарты орындалады.
`c.assign(n,x)`	`x` объектісінің `n` көшірмесін меншіктейді (ассоциативтік контейнерлер үшін емес)
`c.assign(b,e)`	`[b:e)` диапазонының объектілерін меншіктейді

Кейбір контейнерлер және элементтер типтері үшін конструктор немесе көшіру операциясы аластама туындата алады.

Б.4.4 Итераторлар

Контейнерді элементтерінің орналасу реті контейнер итераторымен немесе оған керісінше тәртіппен орналасқан тізбек ретінде көрсетуге болады. Ассоциативтік контейнер үшін орналасу тәртібі салыстыру критериімен анықталады (келісім бойынша `<` операторымен).

Итераторлар	
`p=c.begin()`	`p` `c` контейнерінің бірінші элементіне сілтеме жасап тұр
`p=c.end()`	`p` `c` контейнерінің соңғы элементінен кейінгі элементіне сілтеме жасап тұр
`p=c.rbegin()`	`p` кері бағытта жазылған `c` тізбегінің бірінші элементіне сілтеме жасап тұр
`p=c.rend()`	`p` кері бағытта жазылған `c` тізбегінің соңғы элементінен кейінгі элементіне сілтеме жасап тұр

Б.4.5 Элементтерге қол жеткізу

Кейбір элементтерге тікелей қол жеткізуге болады.

Элементтерге қол жеткізу	
`c.front()`	`c` контейнерінің бірінші элементіне сілтеме
`c.back()`	`c` контейнерінің соңғы элементіне сілтеме
`c[i]`	`c` контейнерінің `i`-элементіне сілтеме; тексерусіз қол жеткізу (тізім үшін емес)

Элементтерге қол жеткізу *(жалғасы)*	
`c.at(i)`	`c` контейнерінің `i`-элементіне сілтеме; тексеру арқылы қол жеткізу (тек `vector` және `deque` контейнерлері үшін)

Кейбір нұсқалар (жүзеге асырулар) – әсіресе солардың тесттік нұсқалары – әрқашанда диапазондарды тексереді, бірақ мұның дұрыстығын немесе әртүрлі компьютерлерде осындай тексерудің болуын есепке алуға болмайды. Егер бұл сұрақ маңызды болса, құжаттаманы қараңыз.

Б.4.6 Стекпен және екіжақты кезекпен операциялар орындау

Стандартты `vector` және `deque` контейнерлері элементтер тізбектерінің шеттерімен (`back`) тиімді операциялар орындауды қамтамасыз етеді. Одан басқа, `list` және `deque` контейнерлері өз тізбегінің басындағы (`front`) элементімен де осындай операциялар орындай алады.

Стекпен және кезекпен операциялар орындау	
`c.push_back(x)`	`x` объектісін `c` контейнерінің соңына кірістіріп қою
`c.pop_back()`	`c` контейнерінің соңғы элементін өшіру
`c.push_front(x)`	`x` объектісін `c` контейнерінің бірінші элементінің алдына кірістіріп қою (тек `list` және `deque` контейнерлері үшін)
`c.at(i)`	`c` контейнерінің `i`-элементіне сілтеме; тексеру арқылы қол жеткізу (тек `vector` және `deque` контейнерлері үшін)
`c.pop_front()`	`c` контейнерінің бірінші элементін өшіру (тек `vector` және `deque` контейнерлері үшін)

`push_front()` және `push_back()` функцияларының элементті контейнерге көшіретініне назар аударыңыз. Бұл контейнер мөлшерінің үлкейетінін көрсетеді (бірге артады). Егер элементтің көшіретін конструкторы аластама туындататын болса, онда кірісітіріп қою әрекеті сәтсіз аяқталуы мүмкін.

Өшірілетін элементтер операциялары мәндер қайтармайтынын айта кетейік. Егер олар мұны істесе, онда аластама туындататын көшіру конструкторлары бұларды жүзеге асыруды әжептеуір күрделендіріп жіберер еді. Стек пен кезектердің элементтеріне қол жеткізу үшін `front()` және `back()` функцияларын пайдалануды ұсынамыз (Б.4.5 бөлімін қ.). Біз барлық шектеулерді тізіп шығуды міндет етіп қоймадық; қалғандары жайлы өзіңіз ой түюге тырысыңыз (көбінесе компиляторлар қолданушыларға олардың қателері жайлы мәлімдеме жасайды) немесе толығырақ құжаттаманы іздестіріңіз.

Б.4.7 Тізімдермен операциялар орындау

Төменде тізімдермен орындалатын операциялар келтірілген:

Тізімдермен орындалатын операциялар	
`q = c.insert(p,x)`	`x` объектісін `p` түйінінің алдына кірістіріп қою
`q = c.insert(p,n,x)`	`x` объектісінің `n` көшірмесін `p` түйінінің алдына кірістіріп қою
`q = c.insert(p,first, last)`	`[first:last)` диапазонының элементтерін `p` түйінінің алдына кірістіріп қою
`q = c.erase(p)`	`c` контейнерінен `p` түйінінде орналасқан элементті өшіру
`q = c.erase(first,last)`	`c` контейнерінен `[first:last)` диапазонында орналасқан элементтерді өшіру
`c.clear()`	`c` контейнеріндегі барлық элементтерді өшіру

`insert()` функциясының `q` нәтижесі ең соңғы кірістіріліп қойылған элементке сілтеме жасайды. `erase()` функциясының `q` нәтижесі ең соңғы өшірілген элементтен кейінгі элементке сілтеме жасайды.

Б.4.8 Мөлшер және сиымдылық

Мөлшер (size) – бұл контейнердегі элементтер саны; сыйымдылық (capacity) – бұл компьютер жадын үлкейту талап етілгенге дейінгі контейнер сақтай алатын элементтер саны.

Мөлшер және сыйымдылық	
`x=c.size()`	`x` – `c` контейнеріндегі элементтер саны
`c.empty()`	`c` контейнері бос па?
`x=c.max_size()`	`x` – `c` контейнеріндегі мүмкін болатын ең үлкен элементтер саны
`x=c.capacity()`	`x` – `c` контейнері үшін бөлінге жады (тек `vector` және `string` кластары үшін)
`c.reserve(n)`	`c` контейнерінен `n` элемент үшін жады бөледі (тек `vector` және `string` кластары үшін)
`c.resize(n)`	`c` контейнерінің мөлшерін `n` -ге тең етіп өзгертеді (тек `vector, string` және `deque` кластары үшін)

Мөлшер мен сыйымдылықты өзгерте отырып, элементтерді жаңа орынға көшіруге болады. Бұдан элементтерге итератордың (және де нұсқауыштар мен сілтемелердің) дұрыс болмай шығуы (яғни ескі адрестерге қатысты болуы мүмкін) орын алады.

Б.4.9 Басқа операциялар

Контейнерлерді көшіруге (Б.4.3 бөлімін қ.), салыстыруға және алмастыруға болады.

Салыстырулар және алмастыру	
`c1==c2`	`c1` және `c2` контейнерлеріндегі барлық сәйкес элементтер бір-біріне тең бе?
`c1!=c2`	`c1` және `c2` контейнерлерінде бір-біріне тең емес сәйкес элементтер бар ма?
`c1<c2`	`c1` контейнері лексикографикалық тәртіппен `c2` контейнерінің алдында орналасқан ба?
`c1<=c2`	`c1` контейнері лексикографикалық тәртіппен `c2` контейнерінің алдында орналасқан ба немесе олар тең бе?
`c1>c2`	`c1` контейнері лексикографикалық тәртіппен `c2` контейнерінен кейін орналасқан ба?
`c1>=c2`	`c1` контейнері лексикографикалық тәртіппен `c2` контейнерінен кейін орналасқан ба немесе олар тең бе?
`swap(c1,c2)`	`c1` контейнері мен `c2` контейнерінің элементтерін бір-бірімен алмастыру
`c1.swap(c2)`	`c1` контейнері мен `c2` контейнерінің элементтерін бір-бірімен алмастыру

Егер контейнерлерді салыстыру соларға сәйкес оператор арқылы орындалса (мысалы, `<`), онда олардың элементтері элементтерді салыстыруға арналған (мысалы, `<`) эквивалентті оператор арқылы салыстырылады.

Б.4.10 Ассоциативтік контейнерлермен орындалатын операциялар

Ассоциативтік контейнерлер кілттер негізінде іздеуді қамтамасыз етеді.

Ассоциативтік контейнерлермен орындалатын операциялар	
`c[k]`	`k` кілті бар элементке сілтеме жасайды (бірегей кілті бар контейнерлер үшін)
`p=c.find(k)`	`p` итераторы `k` кілті бар бірінші элементті көрсетіп тұрады
`p=c.lower_bound(k)`	`p` итераторы `k` кілті бар бірінші элементті көрсетіп тұрады
`p=c.upper_bound(k)`	`p` итераторы `k` кілтінен үлкен бірінші элементті көрсетіп тұрады

Ассоциативтік контейнерлермен орындалатын операциялар *(жалғасы)*	
`pair(p1,p2)= c.equal_range(k)`	`[p1,p2)` диапазоны `k` кілті бар элементтерден тұрады
`r=c.key_comp()`	`r` объектісі – бұл кілттерді салыстыру критериіне сәйкес объектінің көшірмесі
`r=c.value_comp()`	`r` объектісі – бұл бейнеленген мәндерді салыстыру критериіне сәйкес объектінің көшірмесі

`Equal_range` функциясымен қайтарылған жұптың алғашқы итераторы `lower_bound`-қа тең, ал екіншісі – `upper_bound`-қа тең. Сіз баспаға `multimap<string,int>` контейнеріндегі `"Marian"` кілті бар барлық элементтер мәнін шығара аласыз, ол үшін келесі кодты жазу керек:

```
string k = "Marian";
typedef multimap<string,int>::iterator MI;
pair<MI,MI> pp = m.equal_range(k);
if (pp.first!=pp.second)
   cout << "elements with value ' " << k << " ':\n";
else
   cout << "no element with value ' " << k << " '\n";
for(MIp=pp.first;p!=pp.second;++p) cout<<p->second<<'\n';
```

Бұған балама ретінде келесі эквивалентті нұсқауды орындауға болады:

```
pair<MI,MI> pp =
           make_pair(m.lower_bound(k),m.upper_bound(k));
```

Дегенмен, бұл нұсқау екі есеге ұзақ орындалады. `equal_range`, `lower_bound` және `upper_bound` алгоритмдерін реттелген тізбектер (Б.5.4 бөлімі) үшін де орындауға болады. `pair` класының анықтауы Б.6.3 бөлімінде келтірілген.

Б.5 Алгоритмдер

`<algorithm>` тақырыбында 60 шамасында алгоритмдер анықталған. Олардың барлығы итераторлар жұбымен (енгізу үшін) немесе бір итератормен (шығару үшін) анықталған тізбектерге қатысты болып табылады.

Екі тізбекпен көшіруді, салыстыруды және де басқа операцияларды орындау кезінде олардың біріншісі `[b:e)` итераторлар жұбымен беріледі де, ал екіншісі – тек бір `b2` итераторымен беріледі, ол сандары алгоритмді орындауға жеткілікті элементтері бар тізбек басы болып саналады, мысалы, бірінші тізбекте қанша элемент болса, сонша элементі: `[b2:b2+(e- b))` бар екінші тізбек жұмыс істей алады.

`sort` сияқты кейбір алгоритмдер кез келген бағытта қолжетімді итераторларды пайдаланады, ал `find` сияқты көптеген басқалары элементтерді бірбағытты итератор көмегімен тек оқиды.

Көптеген алгоритмдер қалыпты келісімді сақтай отырып, "табылмады" деген сөзді оқиға белгісі ретінде тізбек соңы етіп қайтарады. Біз кезекті алгоритмді сипаттау кезінде бұдан былай бұл туралы қайталап айта бермейміз.

Б.5.1 Тізбектерді толықтырмайтын алгоритмдер

Толықтырылмайтын алгоритм тізбек элементтерін жай ғана оқиды; ол тізбек элементтерінің орналасу тәртібін және олардың мәндерін өзгертпейді.

Тізбектерді толықтырмайтын алгоритмдер	
`f=for_each(b,e,f)`	`f` функциясын `[b:e)` диапазонының әрбір элементіне қолданады; `f` мәнін қайтарады
`p=find(b,e,v)`	`p` итераторы `v` элементінің `[b:e)` диапазонына алғашқы кіруін көрсетіп тұрады
`p=find_if(b,e,f)`	`p` итераторы `[b:e)` диапазонына `f(*p)` шартын қанағаттандыратын алғашқы элементін көрсетіп тұрады
`p=find_first_of(b,e,b2,e2)`	`p` итераторы `[b:e)` диапазонындағы `*p==*q` шартын қанағаттандыратын `[b2:e2)` диапазонының бір `q` элементі үшін бірінші элементті көрсетіп тұрады
`p=find_first_of(b,e,b2,e2,f)`	`p` итераторы `[b:e)` диапазонындағы `f(*p,*q)` шартын қанағаттандыратын `[b2:e2)` диапазонының бір `q` элементі үшін бірінші элементті көрсетіп тұрады
`p=adjacent_find(b,e)`	`p` итераторы `[b:e)` диапазонындағы `*p==*(p+1)` шартын қанағаттандыратын бірінші элементті көрсетіп тұрады
`p=adjacent_find(b,e,f)`	`p` итераторы `[b:e)` диапазонындағы `f(*p,*(p+1))` шартын қанағаттандыратын бірінші элементті көрсетіп тұрады
`equal(b,e,b2)`	`[b:e)` диапазоны мен `[b2:b2+(e-b))` диапазонының барлық элементтері бір-біріне тең бе?
`equal(b,e,b2,f)`	`[b:e)` диапазоны мен `[b2:b2+(e-b))` диапазонының барлық элементтері тексеру үшін `f(*p,*q)` шарты қолданылғанда, бір-біріне тең бе?
`pair(p1,p2)=mismatch(b,e,b2)`	`(p1,p2)` итераторлар жұбы элементтердің `[b:e)` және `[b2:b2+(e-b))` диапазонындағы `!(*p1==*p2)` шарты орындалатын бірінші жұбына қатысты болады

Тізбектерді толықтырмайтын алгоритмдер *(жалғасы)*	
`pair(p1,p2)=mismatch(b,e, b2,f)`	`[b:e)` және `[b2:b2+(e-b))` диапазонындағы `!f(*p1,*p2)` шарты орындалатын барлық элементтер бір-біріне тең бе?
`p=search(b,e,b2,e2)`	`p` итераторы `[b2:e2)` диапазонының кез келген бір элементіне тең болатын `[b:e)` диапазонындағы `*p` бірінші элементін көрсетіп тұрады
`p=search(b.e,b2,e2,f)`	`p` итераторы `[b:e)` диапазонындағы `f(*p,*q)` шартын қанағаттандыратын бірінші элементті көрсетіп тұрады, мұндағы `*q` – `[b2:e2)` диапазонының элементі
`p=find_end(b,e,b2,e2)`	`p` итераторы `[b2:e2)` диапазонының кез келген бір элементіне тең болатын `[b:e)` диапазонындағы соңғы элементті көрсетіп тұрады
`p=find_end( b,e,b2,e2,f)`	`p` итераторы `[b:e)` диапазонындағы `f(*p,*q)` шартын қанағаттандыратын `*p` соңғы элементін көрсетіп тұрады, мұндағы `*q` – `[b2:e2)` диапазонының элементі
`p=search_n(b,e,n,v)`	`p` итераторы `[b:e)` диапазонындағы `*p` бірінші элементін көрсетіп тұрады, бірақ мұнда `[p:p+n)` диапазонының әрбір элементі `v` -ға тең болуы тиіс
`p=search_n(b,e,n,v,f)`	`p` итераторы `[b:e)` диапазонындағы `*p` бірінші элементін көрсетіп тұрады, бірақ мұнда `[p:p+n)` диапазонының әрбір `*q` элементі үшін `f(*q,v)` шарты орындалуы тиіс
`x=count(b,e,v)`	`x` – `v` мәнінің `[b:e)` диапазонына кіру саны
`x=count_if(b,e,v,f)`	`x` – `[b:e)` диапазонындағы `f(*p,v)` шарты орындалатын элементтерінің саны

Элементтерді толықтыруды болдырмауды алгоритмге берілетін `for_each` операциясымен жасау мүмкін емес; ол дұрыс болып саналады. Өзі тексеретін элементтерді өзгертетін операцияны басқа алгоритмдерге беру (мысалы, `count` немесе `==`) орындалмауы тиіс.

Алгоритмді дұрыс пайдалану мысалын қарастырайық.

```
bool odd(int x) { return x&1; }

int n_even(const vector<int>& v)
// v-дағы  жұп сандар санын есептейді
{
    return v.size()- count_if(v.begin(),v.end(),odd);
}
```

Б.5.2 Тізбектерді толықтыратын алгоритмдер

Толықтыратын алгоритмдер олардың аргументтері болып келетін элементтерді өзгерте алады.

Тізбектерді толықтыратын алгоритмдер	
`p=transform(b,e,out,f)`	`*p2=f(*p1)` функциясын `*p2` сәйкес элементтерін `[out:out+(e–b))`; `p=out+(e–b)` диапазонына жаза отырып, `[b:e)` диапазонының әрбір `*p1` элементіне қолданады.
`p=transform(b,e,b2,out,f)`	`*p3=f(*p1,*p2)` функциясын `*p3` мәндерін `[out:out+(e–b))`; `p=out+(e–b)` диапазонына жаза отырып, `[b:e)` диапазонының әрбір `*p1` элементіне және `[b2:b2+(e–b))`-нің `*p2` сәйкес элементіне қолданады.
`p=copy(b,e,out)`	`[b:e)` диапазонын `[out:p)` диапазонына көшіреді
`p=copy_backward(b,e,out)`	`[b:e)` диапазонын `[out:p)` диапазонына өзінің соңғы элементінен бастап көшіреді
`p=unique(b,e)`	`[b:e)` диапазоны элементтерін `[b:p)` диапазонына сыбайлас дубликаттар болмайтындай етіп жылжытады (дубликаттар `==` операторы арқылы анықталады)
`p=unique(b,e,f)`	`[b:e)` диапазоны элементтерін `[b:p)` диапазонына сыбайлас дубликаттар болмайтындай етіп жылжытады (дубликаттар `f` функциясы арқылы анықталады)
`p=unique_copy(b,e,out)`	`[b:e)` диапазонын `[out:p)` диапазонына сыбайлас дубликаттарысыз көшіреді
`p=unique_copy(b,e,out,f)`	`[b:e)` диапазонын `[out:p)` диапазонына сыбайлас дубликаттарын өшіре отырып көшіреді (дубликаттар `f` функциясы арқылы анықталады)
`replace(b,e,v,v2)`	`[b:e)` диапазонындағы `*q==v` теңдігі орындалатын `*q` элементтерін `v2` мәнімен алмастырады
`replace(b,e,f,v2)`	`[b:e)` диапазонындағы `f(*q)` шарты орындалатын `*q` элементтерін `v2` мәнімен алмастырады

Тізбектерді толықтыратын алгоритмдер *(жалғасы)*	
`p=replace_copy(b,e,out,v, v2)`	`[b:e)` диапазонындағы `*q==v` шарты орындалатын `*q` элементтерін `v2` мәнімен алмастыра отырып, `[out:p)` диапазонына көшіреді
`p=replace_copy(b,e,out,f, v2)`	`[b:e)` диапазонындағы `f(*q)` шарты орындалатын `*q` элементтерін `v2` мәнімен алмастыра отырып, `[out:p)` диапазонына көшіреді
`p=remove(b,e,v)`	`[b:e)` диапазонындағы `!(*q==v)` шарты орындалатын `*q` элементтері ғана `[b:p)` диапазонында болатындай етіп соған жылжытады
`p=remove(b,e,v,f)`	`[b:e)` диапазонындағы `!f(*q)` шарты орындалатын `*q` элементтері ғана `[b:p)` диапазонында болатындай етіп соған жылжытады
`p=remove_copy(b,e,out,v)`	`[b:e)` диапазонындағы `!(*q==v)` шарты орындалатын элементтерді ғана `[out:p)` диапазонына көшіреді
`p=remove_copy_if(b,e,out, f)`	`[b:e)` диапазонындағы `!f(*q,v)` шарты орындалатын элементтерді ғана `[out:p)` диапазонына көшіреді
`reverse(b,e)`	`[b:e)` диапазонындағы элементтердің орналасу тәртібін керіге алмастырады
`p=reverse_copy(b,e,out)`	`[b:e)` диапазонындағы элементтердің орналасу тәртібін кері ауыстырып, `[out:p)` диапазонына көшіреді
`rotate(b,m,e)`	Элементтерді циклдік алмастыруды орындайды: `[b:e)` диапазоны бірінші элементі соңғы элементінен кейін тұрған дөңгелек ретінде қарастырылады. `*b` элементін `*m` элементінің орнына жылжытады, жалпы алсақ, `*(b+i)` элементтерін `*((b+(i+ (e-m))%(e-b))` элементтері орнына жылжытады
`p=rotate_copy(b,m,e,out)`	`[b:e)` диапазонын циклдік алмастыру арқылы алынған `[out:p)` тізбегіне көшіреді
`random_shuffle(b,e)`	`[b:e)` диапазоны элементтерін бірқалыпты таратылған кездейсоқ сандар генераторы арқылы араластырады
`random_shuffle(b,e,f)`	`[b:e)` диапазоны элементтерін `f` таратылымы бар кездейсоқ сандар генераторы арқылы араластырады

Shuffle алгоритмі тізбекті ойын картасы қағаздары араласқандай етіп араластырады; басқаша айтқанда, араластырған соң, элементтер кездейсоқ түрде орналасады да, мұндағы "кездейсоқ" сөзінің мағынасы кездейсоқ сандар генераторы туындатқан таралыммен анықталады.

Бұл алгоритмдер олардың аргументтері контейнер болатынын білмейді, сондықтан элементтерді қоса да, өшіре де алмайды. Сонымен, **remove** сияқты алгоритм кіріс тізбегінің ұзындығын, оларды өшіре отырып, кішірейте алмайды; оның орнына мұнда ол бұл элементтерді тізбектің басына қарай жылжытады.

```
typedef vector<int>::iterator VII;

void print_digits(const string& s, VII b, VII e)
{
    cout << s;
    while (b!=e) { cout << *b; ++b; }
    cout << '\n';
}

void ff()
{
    int a[] = { 1,1,1, 2,2, 3, 4,4,4, 3,3,3, 5,5,5,5, 1,1,1 };
    vector<int> v(a,a+sizeof(a)/sizeof(int));
    print_digits("all: ",v.begin(), v.end());

    vector<int>::iterator pp = unique(v.begin(),v.end());
    print_digits("head: ",v.begin(),pp);
    print_digits("tail: ",pp,v.end());

    pp=remove(v.begin(),pp,4);
    print_digits("head: ",v.begin(),pp);
    print_digits("tail: ",pp,v.end());
}
```

Мұның нәтижесі төменде келтірілген:

```
all:  1112234443335555111
      head: 1234351
      tail: 443335555111
      head: 123351
      tail: 1443335555111
```

Б.5.3 Қосалқы алгоритмдер

Техникалық тұрғыдан қарағанда, қосалқы алгоритмдер де тізбектерді толықтыра алады, бірақ олар ұзын тізімде көрінбей қалмас үшін біз оларды жеке көрсетуді жөн көрдік.

Қосалқы алгоритмдер	
`swap(x,y)`	`x` пен `y` орындарын ауыстырады.
`iter_swap(p,q)`	`*p` және `*q` орындарын ауыстырады.
`swap_ranges(b,e,b2)`	`[b:e)` және `[b2:b2+(e-b))` диапазондары элементтерінің орындарын ауыстырады
`fill(b,e,v)`	`[b:e)` диапазонының әрбір элементіне `v` мәнін меншіктейді
`fill_n(b,n,v)`	`[b:b+n)` диапазонының әрбір элементіне `v` мәнін меншіктейді
`generate(b,e,f)`	`[b:e)` диапазонының әрбір элементіне `f()` мәнін меншіктейді
`generate_n(b,n,f)`	`[b:b+n)` диапазонының әрбір элементіне `f()` мәнін меншіктейді
`uninitialized_fill(b,e,v)`	`[b:e)` диапазонының барлық элементтерін `v` мәнімен инициалдайды
`uninitialized_copy(b,e,out)`	`[out:out+(e-b))` диапазонының барлық элементтерін `[b:e)` диапазонының сәйкес элементтерімен инициалдайды

Инициалданбаған тізбектер көбінесе контейнерлерді іске асырудағы программалаудың ең төменгі деңгейлерінде қолданылуы тиіс екеніне назар салыңыз. `uninitialized_fill` және `uninitialized_copy` алгоритмдерінің мақсаты болып саналатын элементтер құрамдас типте болуы тиіс немесе инициалданбаған болуы керек.

Б.5.4 Сұрыптау және іздеу

Сұрыптау мен іздеу іргелі алгоритмдер санатына жатады. Соның өзінде программалаушы талаптары әртүрлі болып келеді. Келісім бойынша салыстыру `<` операторы арқылы орындалады, ал `a` мен `b` мәндері жұптарының эквиваленттігі `==` операторымен емес, `!(a<b)&&!(b<a)` шартымен анықталады.

Сұрыптау және іздеу	
`sort(b,e)`	`[b:e)` диапазонын реттейді.
`sort(b,e,f)`	`[b:e)` диапазонын критерий ретінде `f(*p,*q)` функциясын пайдалана отырып реттейді.
`stable_sort(b,e)`	`[b:e)` диапазонын эквивалентті элементтер реттілігін (тәртібін) сақтай отырып реттейді.
`stable_sort(b,e,f)`	`[b:e)` диапазонын критерий ретінде `f(*p,*q)` функциясын пайдалана отырып және эквивалентті элементтер реттілігін сақтай отырып реттейді.
`partial_sort(b,m,e)`	`[b:e)` диапазонының ішкі `[b:m)` диапазонындағы элементтерді реттейді; ішкі `[m:e)` диапазоны реттелмей қалуы да мүмкін.
`partial_sort(b,m,e,f)`	`[b:e)` диапазонының ішкі `[b:m)` диапазонындағы элементтерді критерий ретінде `f(*p,*q)` функциясын пайдалана отырып реттейді; ішкі `[m:e)` диапазоны реттелмей қалуы да мүмкін.
`partial_sort_copy(b,e,b2,e2)`	`[b:e)` диапазоны элементтерін реттейді, олардың саны `e2-b2` алғашқы элементтерін `[b2:e2)` диапазонына көшіруге жеткілікті болуы тиіс.
`partial_sort_copy(b,e,b2, e2,f)`	`[b:e)` диапазоны элементтерін реттейді, олардың саны `e2-b2` алғашқы элементтерін `[b2:e2)` диапазонына көшіруге жеткілікті болуы тиіс; салыстыру критериі ретінде `f` функциясы қолданылады.
`nth_element(b,e)`	`[b:e)` диапазонының `n`-элементін сәйкес орынға кірістіріп қояды
`nth_element(b,e,f)`	`[b:e)` диапазонының `n`-элементін сәйкес орынға салыстыру критериі ретінде `f` функциясын қолдана отырып кірістіріп қояды.
`p=lower_bound(b,e,v)`	`p` итераторы `[b:e)` диапазонына `v` мәнінің алғашқы кіруін көрсетеді.
`p=lower_bound(b,e,v,f)`	`p` итераторы `[b:e)` диапазонына `v` мәнінің алғашқы кіруін көрсетеді; салыстыру критериі ретінде `f` функциясы қолданылады.
`p=upper_bound(b,e,v)`	`p` итераторы `[b:e)` диапазонына `v` мәнінен артық мәннің алғашқы кіруін көрсетеді.
`p=upper_bound(b,e,v,f)`	`p` итераторы `[b:e)` диапазонына `v` мәнінен артық мәннің алғашқы кіруін көрсетеді; салыстыру критериі ретінде `f` функциясы қолданылады.
`binary_search(b,e,v)`	`[b:e)` реттелген тізбегінде `v` мәні бар ма?

Сұрыптау және іздеу *(жалғасы)*	
`binary_search(b,e,v,f)`	`[b:e)` реттелген тізбегінде `v` мәні бар ма? Салыстыру критериі ретінде `f` функциясы қолданылады.
`pair(p1,p2)=equal_range(b,e,v)`	`[p1,p2)` – бұл `v` мәні бар `[b:e)` диапазонының ішкі тізбегі; негізінде, алгоритм `v` мәнін бинарлық іздеуді орындайды.
`pair(p1,p2)=equal_range(b,e,v,f)`	`[p1,p2)` – бұл `v` мәні бар `[b:e)` диапазонының ішкі тізбегі; негізінде, алгоритм `v` мәнін бинарлық іздеуді салыстыру критериі ретінде `f` функциясы қолдана отырып орындайды.
`p=merge(b,e,b2,e2,out)`	Екі реттелген `[b2:e2)` және `[b:e)` тізбектерін `[out:p)` тізбегіне біріктіреді.
`p=merge(b,e,b2,e2,out,f)`	Екі реттелген `[b2:e2)` және `[b:e)` тізбектерін салыстыру критериі ретінде `f` функциясы қолдана отырып, `[out,out+p)` тізбегіне біріктіреді.
`inplace_merge(b,m,e)`	Екі реттелген `[b:m)` және `[m:e)` тізбектерін реттелген `[b:e)` тізбегіне біріктіреді.
`inplace_merge(b,m,e,f)`	Екі реттелген `[b:m)` және `[m:e)` тізбектерін салыстыру критериі ретінде `f` функциясы қолдана отырып, реттелген `[b:e)` тізбегіне біріктіреді.
`p=partition(b,e,f)`	`f(*p1)` шарты орындалатын элементтерді `[b:p)` диапазонына, ал қалғандарын `[p:e)` диапазонына орналастырады.
`p=stable_partition(b,e,f)`	`f(*p1)` шарты орындалатын элементтерді `[b:p)` диапазонына, ал қалғандарын `[p:e)` диапазонына солардың салыстырмалы тәртібін сақтай отырып орналастырады.

Келесі мысалды қарастырайық.

```
vector<int> v;
list<double> lst;
v.push_back(3); v.push_back(1);
v.push_back(4); v.push_back(2);
lst.push_back(0.5); lst.push_back(1.5);
lst.push_back(2); lst.push_back(2.5);  // lst is in order
sort(v.begin(),v.end());               // put v in order
vector<double> v2;
merge(v.begin(),v.end(),lst.begin(),
                lst.end(),back_inserter(v2));
for (int i = 0; i<v2.size(); ++i) cout << v2[i] << ", ";
```

Кірістіру алгоритмдері Б.6.1 бөлімінде сипатталған. Қорытындысында келесі нәтиже алынады:

```
0.5, 1, 1.5, 2, 2, 2.5, 3, 4,
```

`Equal_range`, `lower_bound` және `upper_bound` алгоритмдері ассоциативтік контейнерлерге арналған солардың эквиваленттері тәрізді пайдаланылады (Б.4.10 бөлімі).

Б.5.5 Жиындарға арналған алгоритмдер

Бұл алгоритмдер тізбектерді элементтер жиыны ретінде көрсетеді де, жиындармен атқарылатын негізгі операцияларды орындайды. Кіріс және шығыс тізбектері реттелген болып саналады.

Жиындарға арналған алгоритмдер	
`includes(b,e,b2,e2)`	`[b2:e2)` диапазонының барлық элементтері сол сәтте `[b:e)` диапазонына да жата ма?
`includes(b,e,b2,e2,f)`	Егер салыстыру критериі ретінде `f` функциясы пайдаланылатын болса, `[b2:e2)` диапазонының барлық элементтері сол сәтте `[b:e)` диапазонына да жата ма?
`p=set_union(b,e,b2,e2,out)`	`[b:e)` диапазонына немесе `[b2:e2)` диапазонына жататын элементтерден тұратын реттелген `[out:p)` тізбегін құрады.
`p=set_union (b,e,b2,e2,out,f)`	Егер салыстыру критериі ретінде `f` функциясы пайдаланылатын болса, `[b:e)` диапазонына немесе `[b2:e2)` диапазонына жататын элементтерден тұратын реттелген `[out:p)` тізбегін құрады.
`p=set_intersection (b,e,b2,e2,out)`	`[b:e)` диапазонына немесе `[b2:e2)` диапазонына жататын элементтерден тұратын реттелген `[out:p)` тізбегін құрады.
`p=set_intersection (b,e,b2,e2,out,f)`	Салыстыру критериі ретінде `f` функциясын пайдалана отырып, `[b:e)` диапазонына немесе `[b2:e2)` диапазонына жататын элементтерден тұратын реттелген `[out:p)` тізбегін құрады.
`p=set_difference (b,e,b2,e2,out)`	`[b:e)` диапазонына жататын, бірақ `[b2:e2)` диапазонына жатпайтын элементтерден тұратын реттелген `[out:p)` тізбегін құрады.

Жиындарға арналған алгоритмдер *(жалғасы)*	
`p=set_difference(b,e,b2,e2,out,f)`	Салыстыру критериі ретінде `f` функциясын пайдалана отырып, `[b:e)` диапазонына жататын, бірақ `[b2:e2)` диапазонына жатпайтын элементтерден тұратын реттелген `[out:p)` тізбегін құрады.
`p=set_symmetric_difference(b,e,b2,e2,out)`	`[b:e)` диапазонына немесе `[b2:e2)` диапазонына жататын, бірақ бір мезетте екеуіне де жатпайтын элементтерден тұратын реттелген `[out:p)` тізбегін құрады.
`p=set_symmetric_difference(b,e,b2,e2,out,f)`	Салыстыру критериі ретінде `f` функциясын пайдалана отырып, `[b:e)` диапазонына немесе `[b2:e2)` диапазонына жататын, бірақ бір мезетте екеуіне де жатпайтын элементтерден тұратын реттелген `[out:p)` тізбегін құрады.

Б.5.6 Үйінділер

Үйінді (heap) – бұл төбесінде ең үлкен мәні бар элемент тұратын мәліметтер құрылымы. Үйінділермен орындалатын алгоритмдер программалаушыларға кез келген бағытта қолжетімді тізбектермен жұмыс істеуге мүмкіндік береді.

Үйінділермен орындалатын операциялар	
`make_heap(b,e)`	Үйінді ретінде пайдалануға болатын тізбек құрады.
`make_heap(b,e,f)`	Салыстыру критериі ретінде `f` функциясын пайдалана отырып, үйінді ретінде пайдалануға болатын тізбек құрады.
`push_heap(b,e)`	Үйіндіге элемент қосады (сәйкес орынға)
`push_heap(b,e,f)`	Салыстыру критериі ретінде `f` функциясын пайдалана отырып, үйіндіге элемент қосады (сәйкес орынға)
`pop_heap(b,e)`	Үйіндіден ең үлкен (бірінші) элементті жояды.
`pop_heap(b,e,f)`	Салыстыру критериі ретінде `f` функциясын пайдалана отырып, үйіндіден ең үлкен (бірінші) элементті жояды.
`sort_heap(b,e)`	Үйіндіні реттейді (сұрыптайды).
`sort_heap(b,e,f)`	Салыстыру критериі ретінде `f` функциясын пайдалана отырып, үйіндіні реттейді.

Үйінді элементтерді жылдам қосу мүмкіндігін береді де, ең үлкен мәнді элементке жылдам қол жеткізуді қамтамасыз етеді. Негізінде, үйінділер приоритеттері (басымдықтары) бар кезектерді іске асыру кезінде пайдаланылады.

Б.5.7 Алмастырулар

Алмастырулар тізбектің элементтері комбинацияларын жасау (туындату, генерациялау) үшін пайдаланылады. Мысалы, `abc` тізбегінің алмастырулары болып `abc, acb, bac, bca, cab` және `cba` тізбектері саналады.

Алмастырулар	
`x=next_permutation(b,e)`	`[b:e)` тізбегінің лексикографикалық тәртіптегі келесі алмастыруын құрады.
`x=next_permutation(b,e,f)`	Салыстыру критериі ретінде `f` функциясын пайдалана отырып, `[b:e)` тізбегінің лексикографикалық тәртіптегі келесі алмастыруын құрады.
`x=prev_permutation(b,e)`	`[b:e)` тізбегінің лексикографикалық тәртіптегі алдыңғы алмастыруын құрады.
`x=prev_permutation(b,e,f)`	Салыстыру критериі ретінде `f` функциясын пайдалана отырып, `[b:e)` тізбегінің лексикографикалық тәртіптегі алдыңғы алмастыруын құрады.

Егер `[b:e)` тізбегінде соңғы алмастыру (бұл мысалда ол `cba` алмастыруы) болатын болса, онда `next_permutation` алгоритмі `false` мәніне тең `x` мәнін қайтарады; мұндай жағдайда алгоритм бірінші алмастыруды (бұл мысалда ол `abc` алмастыруы) жасайды. Егер `[b:e)` тізбегінде бірінші алмастыру (бұл мысалда ол `abc` алмастыруы) болатын болса, онда `prev_permutation` алгоритмі `false` мәніне тең `x` мәнін қайтарады; мұндай жағдайда алгоритм соңғы алмастыруды (бұл мысалда ол `cba` алмастыруы) жасайды.

Б.5.8 `min` және `max` функциялары

Мәндерді салыстыру көптеген жағдайларда пайдалы болып табылады.

`min` және `max`	
`x=max(a,b)`	`x` – `a` және `b` мәндерінің үлкені.
`x=max(a,b,f)`	`x` – `a` және `b` мәндерінің үлкені; салыстыру критериі ретінде `f` функциясы пайдаланылады.
`x=min(a,b)`	`x` – `a` және `b` мәндерінің кішісі.
`x=min(a,b,f)`	`x` – `a` және `b` мәндерінің кішісі; салыстыру критериі ретінде `f` функциясы пайдаланылады.
`p=max_element(b,e)`	`p` итераторы `[b:e)` диапазонының ең үлкен элементін көрсетіп тұрады.

min және max *(жалғасы)*	
`p=max_element(b,e,f)`	`p` итераторы `[b:e)` диапазонының ең үлкен элементін көрсетіп тұрады; салыстыру критериі ретінде `f` функциясы пайдаланылады.
`p=min_element(b,e)`	`p` итераторы `[b:e)` диапазонының ең кіші элементін көрсетіп тұрады.
`p= min_element(b,e,f)`	`p` итераторы `[b:e)` диапазонының ең кіші элементін көрсетіп тұрады; салыстыру критериі ретінде `f` функциясы пайдаланылады.
`lexicographical_compare(b,e,b2,e2)`	`[b:e)<[b2:e2)` шарты орындала ма?
`lexicographical_compare(b,e,b2,e2,f)`	Егер салыстыру критериі болып `f` функциясы пайдаланылса, `[b:e)<[b2:e2)` шарты орындала ма?

Б.6 STL кітапханасының утилиттері

Стандартты кітапханада стандартты кітапханалық алгоритмдерді пайдалануды жеңілдететін бірнеше құралдар бар.

Б.6.1 Кірістірмелер

Контейнерге итератор арқылы нәтижелерді жазу итератор көрсетіп тұрған элементтерді қайта жазуға болады дегенді білдіреді. Бұл компьютер жадының толып кетуіне және соның салдарынан зақымдануына әкеліп соғады. Келесі мысалды қарастырайық.

```
void f(vector<int>& vi)
{
    fill_n(vi.begin(),200,7);
    // vi[0]..[199] элементтеріне 7 мәнін меншіктейміз
}
```

Егер `vi` векторының элементтері саны 200-ден аз болса, онда қауіп туындайды.

Стандартты кітапхананың `<iterator>` тақырыбында осы мәселені контейнерге, оның ескі элементтерін қайта жазу емес, элементтер қосу (кірістіру) арқылы шешуге арналған үш итератор бар. Осы кірістіру итераторларын іске қосу үшін үш функция қолданылады.

Кірістіру алгоритмдері	
`r=back_inserter(c)`	`*r=x c.push_back(x)` шақыруды орындайды
`r=front_inserter(c)`	`*r=x c.push_front(x)` шақыруды орындайды
`r=inserter(c,p)`	`*r=x c.insert(p,x)` шақыруды орындайды

`insert(p,x)` алгоритмінің дұрыс жұмыс істеуі үшін `p` итераторы `c` контейнері үшін дұрыс итератор ретінде жұмыс атқаруы тиіс. Әрине, кірістіру итераторы арқылы келесі элементті әрбір жазған сайын контейнер бір элементке артып отырады. Жазу кезінде кірістіру алгоритмі бұрыннан бар элементті қайта жазбайды, ол `push_back(x)`, `c.push_front()` немесе `insert()` функциялары арқылы тізбекке жаңа элемент қосады. Келесі мысалды қарастырайық.

```
void g(vector<int>& vi)
{
    fill_n(back_inserter(vi),200,7);
    // vi соңына 200 жетілікті қосады
}
```

Б.6.2 Объект-функциялар

Көптеген стандартты алгоритмдер есепті шығару тәсілін анықтау үшін аргументтер ретінде объект-функцияларды (немесе функцияларды) қабылдайды. Әдетте бұл функциялар предикаттар (bool типіндегі мәндер қайтаратын функциялар), арифметикалық операциялар және салыстыру критерилері ретінде пайдаланылады. Бірнеше ең жалпы объект-функциялар стандартты кітапхананың `<functional>` тақырыбында сипатталған.

Предикаттар	
`p=equal_to<T>()`	Егер `x` пен `y` `T` типінде болса, `p(x,y)` предикаты `x==y` дегенді білдіреді.
`p=not_equal_to<T>()`	Егер `x` пен `y` `T` типінде болса, `p(x,y)` предикаты `x!=y` дегенді білдіреді.
`p=greater<T>()`	Егер `x` пен `y` `T` типінде болса, `p(x,y)` предикаты `x>y` дегенді білдіреді.
`p=less<T>()`	Егер `x` пен `y` `T` типінде болса, `p(x,y)` предикаты `x<y` дегенді білдіреді.
`p=greater_equal<T>()`	Егер `x` пен `y` `T` типінде болса, `p(x,y)` предикаты `x>=y` дегенді білдіреді.
`p=less_equal<T>()`	Егер `x` пен `y` `T` типінде болса, `p(x,y)` предикаты `x<=y` дегенді білдіреді.

Предикаттар *(жалғасы)*	
`p=logical_and<T>()`	Егер `x` пен `y` `T` типінде болса, `p(x,y)` предикаты `x&&y` дегенді білдіреді.
`p=logical_or<T>()`	Егер `x` пен `y` `T` типінде болса, `p(x,y)` предикаты `x||y` дегенді білдіреді.
`p=logical_not<T>()`	Егер `x` пен `y` `T` типінде болса, `p(x,y)` предикаты `!x` дегенді білдіреді.

Келесі мысалды қарастырайық.

```
vector<int> v;
// . . .
sort(v.begin(),v.end(),greater<int>());
// v-ны кемуі бойынша сұрыптау
```

`logical_and` және `logical_or` предикаттары әрқашанда өзінің екі аргументін де (ал `&&` және `||` операторлары есептемейді) есептеп шығаратынына назар аударыңыз.

Арифметикалық операциялар	
`f=plus<T>()`	`f(x,y) x+y` дегенді білдіреді, мұндағы `x` пен `y` `T` типінде.
`f=minus<T>()`	`f(x,y) x-y` дегенді білдіреді, мұндағы `x` пен `y` `T` типінде.
`f=multiplies<T>()`	`f(x,y) x*y` дегенді білдіреді, мұндағы `x` пен `y` `T` типінде.
`f=divides<T>()`	`f(x,y) x/y` дегенді білдіреді, мұндағы `x` пен `y` `T` типінде.
`f=modulus<T>()`	`f(x,y) x%y` дегенді білдіреді, мұндағы `x` пен `y` `T` типінде.
`f=negate<T>()`	`f(x,y) -x` дегенді білдіреді, мұндағы `x` `T` типінде.
`f=bind2nd(g,y)`	`f(x) g(x,y)` дегенді білдіреді.
`f=bind1st(g,x)`	`f(y) g(x,y)` дегенді білдіреді.
`f=mem_fun(mf)`	`f(p) p->mf()` дегенді білдіреді.
`f=mem_fun_ref(mf)`	`f(r) r.mf()` дегенді білдіреді.
`f=not1(g)`	`f(x) !g(x)` дегенді білдіреді.
`f=not2(g)`	`f(x,y) !g(x,y)` дегенді білдіреді.

Б.6.3 `pair` класы

Стандартты кітапхананың `<utility>` тақырыбында, `pair` класын қосып есептегенде, бірнеше қосалқы компоненттер бар.

```
template <class T1, class T2>
    struct pair {
      typedef T1 first_type;
      typedef T2 second_type;
      T1 first;
      T2 second;
      pair();   // default constructor
      pair(const T1& x , const T2& y );

      // copy operations:
      template<class U, class V > pair(const pair<U,V >& p);
};

template <class T1, class T2>
    pair<T1,T2> make_pair(T1 x,T2 y)
                {return pair<T1,T2>(x,y);}
```

`make_pair` функциясы жұптарды қолдануды қарапайым етеді. Мысалы, мән қайтаратын функция сұлбасы мен қателер индикаторын қарастырайық.

```
pair<double,error_indicator> my_fct(double d)
{
    errno = 0;
    // C тілі стилінде қателер индикаторын тазалаймыз
    // d айнымалысымен байланысты есептеулер жүргіземіз,
                                          x-ті есептейміз
    error_indicator ee = errno;
    errno = 0;
    // C тілі стилінде қателер индикаторын тазалаймыз
    return make_pair(x,ee);
}
```

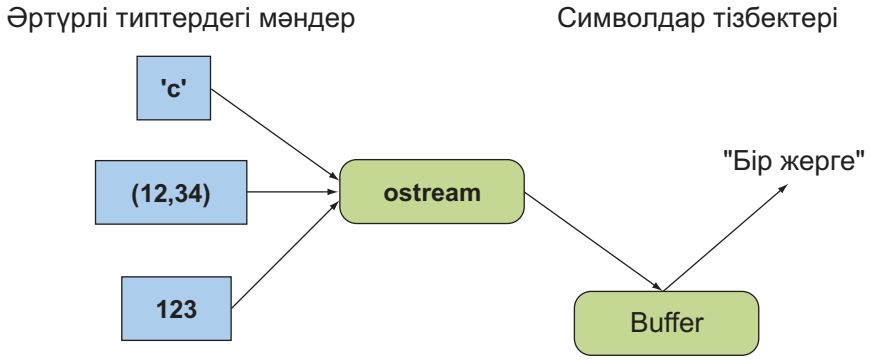
Бұл мысал пайдалы идиома болып табылады. Оны мынадай түрде пайдалануға болады:

```
pair<int,error_indicator> res = my_fct(123.456);
if (res.second==0) {
     // use res.first
}
else {
     // ой: қате
}
```

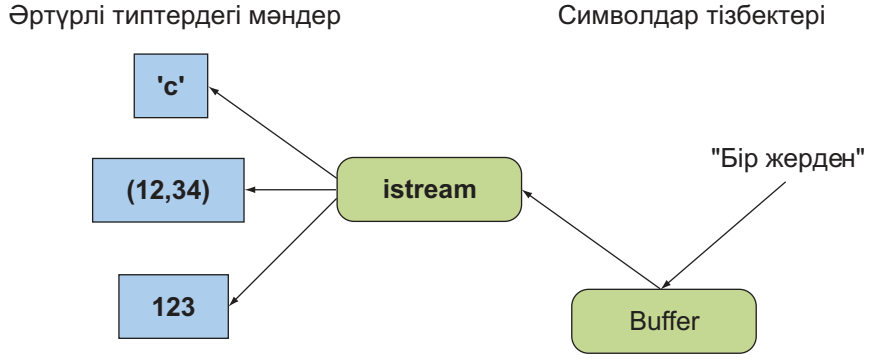
Б.7 Енгізу-шығару ағымдары

Енгізу-шығару ағымдары кітапханасында мәтін мен сандық мәндерді форматталған және форматталмаған буферленген енгізу-шығару құралдары бар. Енгізу-шығару ағымдарын анықтаулар **<istream>**, **<ostream>** және т.с.с. (Б.1.1 бөлімін қ.) тақырыптарда орналасқан.

ostream класының объектісі типі бар объектілерді символдар ағымына (байттарға) түрлендіреді.



istream класының объектісі символдар ағымын (байттарға) типі бар объектілерге түрлендіреді.



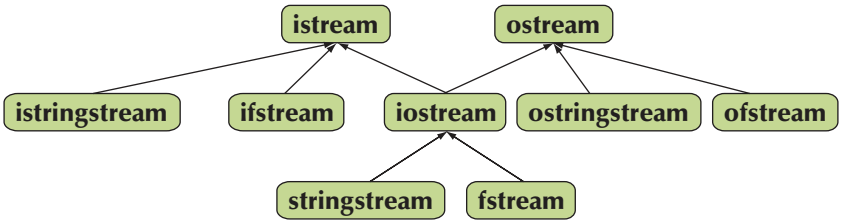
iostream класының объектісі – бұл **istream** класының объектісі болып та және **ostream** класының объектісі болып та жұмыс істей алатын ағым. Символдар ағымын (байттарға) типі бар объектілерге түрлендіреді. Диаграммада көрсетілген буферлер ағымдық буферлер (streambufs) болып табылады. Егер оқырмандарға **iostream** класынан басқа жаңа құрылғылар, файлдар немесе жадылар түрлеріне көшу керек болса, олар керекті сипаттамаларды кәсіби оқулықтардан таба алады.

Үш стандартты ағым бар:

Стандартты енгізу-шығару ағымдары	
`cout`	Стандартты шығару ағымы (келісім бойынша, көбінесе экран).
`cin`	Стандартты енгізу ағымы (келісім бойынша, көбінесе пернетақта).
`cerr`	Стандартты қате мәлімдемелер ағымы (буферленбеген).

Б.7.1 Енгізу-шығару ағымдарының иерархиясы

`istream` ағымын енгізу құрылғысымен (мысалы, пернетақтамен), файлмен немесе `string` класының объектісімен байланыстыруға болады. Осылайша, `ostream` ағымын шығару құрылғысымен (мысалы, мәтін терезесімен), файлмен немесе `string` класының объектісімен байланыстыруға болады. Енгізу-шығару ағымдары кластар иерархиясын құрады.



Ағымды конструктор арқылы немесе `open()` функциясын шақыру жолымен ашуға болады.

Енгізу ағымдары	
`stringstream(m)`	`m` режимінде бос тіркестік ағым жасайды.
`stringstream(s,m)`	`m` режимінде `string s` объектісі бар тіркестік ағым жасайды.
`fstream()`	Кейінірек ашылатын файл жасайды.
`fstream(s,m)`	`m` режимінде `s` файлын ашады және оған сілтеме жасайтын файл ағымын жасайды.
`fs.open(s,m)`	`m` режимінде `s` файлын ашады және осы файл мен `fs` ағымы арасында байланыс орнатады.
`fs.is_open()`	`fs` ағымы ашылды ма?

Файл ағымдары үшін файл аттары С тілі стиліндегі тіркес болып табылады.

Файлды төменде көрсетілген режимдердің бірінде ашуға болады:

Ағымдар режимдері	
`ios_base::app`	Қосу (яғни файл соңына жазбалар қосу).
`ios_base::ate`	"ate" сөзі "at end" дегенді білдіреді (яғни файлды ашу және соңын іздеу).
`ios_base::binary`	Бинарлық режим. Бұл режимнен сақтаныңыз, өйткені ол нақты жүйенің спецификасына тәуелді болады.
`ios_base::in`	Оқу үшін
`ios_base::out`	Жазу үшін
`ios_base::trunc`	Файлды нөлдік ұзындыққа дейін қысқарту

Бұл режимдердің әрқайсысында файлды ашу операциялық жүйеге және оның файлды тек осылай ашу керек, басқаша емес деген сияқты программалаушы талаптарын есепке алу мүмкіндіктеріне тәуелді бола алады. Нәтижесінде ағым `good()` қалпында бола алмай қалуы мүмкін. Мысал қарастырайық.

```
void my_code(ostream& os); // my code функциясы кез келген
                           // шығару ағымын пайдалана алады

ostringstream os;  //"o" әрпі "шығару үшін" дегенді білдіреді
ofstream of("my_file");
if (!of) error("жазу үшін 'my_file' ашу мүмкін емес");
my_code(os);  // use a string класының объектісі қолданылады
my_code(of); // файл қолданылады
```

11.3 бөлімін қ.

Б.7.2 Қателерді өңдеу

`iostream` ағымы төрт қалыпты жағдайдың бірінде болуы мүмкін.

Ағымдар қалып-күйі	
`good()`	Операциялар табысты аяқталған.
`eof()`	Файл соңы анықталған ("end of file").
`fail()`	Күтілмеген жағдай орын алған (мысалы, цифр ізделіп, оның орнына `'x'` символы анықталған).
`bad()`	Күтілмеген әрі күрделі жағдай орын алған (мысалы, дискіден мәлімет оқу кезінде қате шыққан).

`s.exceptions()` функциясын пайдалану кезінде, `iostream` ағымы `good()` қалпынан басқа қалыпқа ауысқан болса (10.6 бөлімін қ.), онда программалаушы ағымның аластама туындатуын талап етуі мүмкін.

Кез келген операция, өзінің орындалу нәтижесінде ағымды `good()` қалпында келтіре алмаса, оның ешқандай әсері болмайды; мұндай жағдай "no op" деп аталады.

`iostream` класының объектісін шарт түрінде қолдануға болады. Мұндағы жағдайда, егер `iostream` ағымы `good()` қалпында болса, шарт ақиқат (табысты) болып табылады. Осы әрекет ағым мәндерін оқуға арналған идиоманың таралуына негіз болды.

```
X x; // X типті бір мәнді сақтауға арналған "енгізу буфері"
while (cin>>x) {
     // x объектісімен кез келген бір әрекет орындау
}
// егер >> операторы cin ағымынан X класының кезекті
   объектісін оқи алмаса,біз осы нүктеде боламыз
```

Б.7.3 Енгізу операциялары

`string` класының объектісіне енгізу операцияларынан басқа енгізу операцияларының басым бөлігі `istream` тақырыбында сипатталған; бұл операциялар `<string>` тақырыбында сипатталған:

Форматталған енгізу	
`in >> x`	`x` объектісі типінің ережелері бойынша `in` ағымынан `x` объектісіне енгізу.
`getline(in,s)`	Сөз тіркесін `in` ағымынан `string` тіркесінің `s` объектісіне енгізу.

Егер басқасы көрсетілмесе, енгізу операциясы `istream` класының объектісіне сілтемені қайтарады, сондықтан осындай операциялардың шағын тізбегін жасауға болады, мысалы, `cin>>x>>y;`

Форматталмаған енгізу	
`x=in.get()`	`in` ағымынан бір символ енгізеді де, оның бүтінсандық мәнін қайтарады.
`in.get(c)`	`in` ағымынан бір символды `c` айнымалысына енгізеді.
`in.get(p,n)`	`in` ағымынан `n`-нен аспайтын символдар санын `p` позициясынан басталатын жиымға енгізеді.
`in.get(p,n,t)`	`in` ағымынан `n`-нен аспайтын символдар санын `p` позициясынан басталатын жиымға енгізеді; `t` символы енгізудің соңы болып есептеледі.
`in.getline(p,n)`	`in` ағымынан `n`-нен аспайтын символдар санын `p` позициясынан басталатын жиымға енгізеді; `in` ағымындағы енгізу соңы белгісін өшіреді.

Форматталмаған енгізу (*жалғасы*)	
`in.getline(p,n,t)`	`in` ағымынан `n`-нен аспайтын символдар санын `p` позициясынан басталатын жиымға енгізеді; `t` символы енгізудің соңы болып есептеледі; `in` ағымындағы енгізу соңы белгісін өшіреді.
`in.read(p,n)`	`in` ағымынан `n`-нен аспайтын символдар санын `p` позициясынан басталатын жиымға енгізеді.
`x=in.gcount()`	`x` – бұл `in` ағымынан мәліметтерді форматталмаған енгізудегі уақыт бойынша соңғы операцияның орындалу кезінде енгізілген символдар саны.

`get()` және `getline()` функциялары `p[0]` . . . және т.с.с. ұяларға жазылған символдардан кейін 0 санын орналастырады (егер символдар енгізілген болса); `getline()` функциясы, егер ол анықталған болса, енгізу ағымынан енгізу соңы (`t`) белгісін өшіреді, ал `get()` функциясы мұны жасамайды. `read(p,n)` функциясы оқылған символдардан соң, жиымға `0` санын жазбайды. Әрине, форматталмаған енгізу операцияларына қарағанда, форматталған енгізу операторларын пайдалану жеңіл және олар қателерге де төтеп бере алады.

Б.7.4 Шығару операциялары

`string` класының объектісіне жазу операцияларынан басқа шығару операцияларының басым бөлігі `ostream` тақырыбында сипатталған; мұндай операциялар `<string>` тақырыбында сипатталған:

Шығару операциялары	
`out << x`	`x` объектісі типінің ережелері бойынша `out` ағымына `x` объектісін жазады.
`out.put(c)`	`out` ағымына `c` символын `c` жазады.
`out.write(p,n)`	`out` ағымына `p[0]..p[n-1]` символдарын жазады.

Егер басқасы көрсетілмесе, `ostream` ағымына кірістіру операциялары оның объектілеріне сілтемені қайтарады, сондықтан шығару операцияларының шағын тізбегін жасауға болады, мысалы, `cout<<x<<y;`

Б.7.5 Форматтау

Енгізу-шығару ағымының форматы объект типінің, ағым қалып-күйінің, локализациялау туралы ақпараттың (`<locale>` бөлімін қ.) комбинациясымен және

тікелей операциялармен басқарылады. Бұл туралы ақпараттың басым бөлігі 10-11 тарауларда баяндалған. Бұл жерде біз жай ғана стандартты манипуляторларды (ағымды толықтыратын операциялар) тізіп келтіреміз, өйткені солар форматты өзгертудің ең қарапайым тәсілін қамтамасыз етеді.

Локализациялау сұрақтарын қарастыру бұл кітап аясынан тысқары аймақта жатыр.

Б.7.6 Стандартты манипуляторлар

Стандартты кітапханада форматты әртүрлі өзгерулерге сәйкес манипуляторлар қарастырылған. Стандартты манипуляторлар `<ios>`, `<istream>`, `<ostream>`, `<iostream>` және `<iomanip>` тақырыптарында анықталған (аргументтер қабылдайтын манипуляторлар үшін).

Енгізу-шығару манипуляторлары	
`s<<boolalpha`	`true` және `false` мәндерінің символикалық бейнелерін пайдалану (енгізу және шығару).
`s<<noboolalpha`	`s.unsetf(ios_base::boolalpha)`
`s<<showbase`	`oct` шығару префиксін `0` түрінде және `hex` шығару префиксін `0x` түрінде шығару.
`s<<noshowbase`	`s.unsetf(ios_base::showbase)`
`s<<showpoint`	Әрқашанда ондық нүктені көрсету.
`s<<noshowpoint`	`s.unsetf(ios_base::showpoint)`
`s<<showpos`	Оң сандар алдында `+` символын көрсету.
`s<<noshowpos`	`s.unsetf(ios_base::showpos)`
`s>>skipws`	Босорындарды өткізіп жіберу.
`s>>noskipws`	`s.unsetf(ios_base::skipws)`
`s<<uppercase`	Сандарды шығаруда жоғарғы регистрді қолдану, мысалы, `1.2E10` және `0X1A2`, `1.2e10` және `0x1a2` емес.
`s<<nouppercase`	`x` және `e` шығару, `X` және `E` емес.
`s<<internal`	Формат шаблондары көрсетілген орындарда босорын кірістіру.
`s<<left`	Мәннен кейін босорын кірістіру.
`s<<right`	Мән алдына босорын кірістіру.
`s<<dec`	Санау жүйесінің негізі `10`-ға тең.
`s<<hex`	Санау жүйесінің негізі `16`-ға тең.
`s<<oct`	Санау жүйесінің негізі `8`-ге тең.
`s<<fixed`	Жылжымалы нүктелі сандар форматы `dddd.dd`
`s<<scientific`	Ғылыми формат `d.ddddEdd`
`s<<endl`	`'\n'` кірістіріп, буферді тазарту

Енгізу-шығару манипуляторлары *(жалғасы)*	
`s<<ends`	`'\0'` кірістіру
`s<<flush`	Ағымды тазарту
`s>>ws`	Ажыратқыштарды жою
`s<<resetiosflags(f)`	`f` жалаушаларын түсіру
`s<<setiosflags(f)`	`f` жалаушаларын орнату
`s<<setbase(b)`	Негізі `b` бүтін сандарды шығару
`s<<setfill(c)`	`c` символын толтыру символы ету.
`s<<setprecision(n)`	Дәлдігі `n` цифрға тең
`s<<setw(n)`	Келесі өрістің ені `n` символға тең

Осы операциялардың әрқайсысы `s` ағымының өзіндегі бірінші операндына сілтеме қайтарады. Мысал қарастырайық.

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << endl;
```

Осы код экранға мынаны шығарады:

```
1234,4d2,2322
```

Өз кезегінде мынадай код:

```
cout<<'('<<setw(4)<<setfill('#')<<12<<") ("<<12<<")\n";
```

экранға мынаны шығарады:

```
(##12) (12)
```

Жылжымалы нүктелі сандарды шығару форматын тікелей орнату үшін, келесі нұсқауды пайдаланыңыз:

```
b.setf(ios_base::fmtflags(0), ios_base::floatfield)
```

11 тарауды қ.

Б.8 Тіркестермен әрекеттер орындау

Стандартты кітапханада символдарды жіктеу операциялары `<cctype>` тақырыбында, сәйкес операциялары бар тіркестер (сөз тіркестері) `<string>` тақырыбында, регулярлық өрнектер `<regex>` тақырыбында (C++0x) және C-тіркестерін сүйемелдеу `<cstring>` тақырыбында қарастырылған.

Б.8.1 Символдарды жіктеу

Негізгі топтағы символдарды төменде көрсетілгендей түрде жіктеуге болады:

Символдарды жіктеу	
`isspace(c)`	`c` символы ажыратқыш (`' '`, `'\t'`, `'\n'`, т.с.с.) болып табыла ма?
`isalpha(c)`	`c` символы әріп (`'a'..'z'`, `'A'..'Z'`) болып табыла ма? (Ескерту: бірақ `'_'` емес)
`isdigit(c)`	`c` символы ондық цифр (`'0'..'9'`) болып табыла ма?
`isxdigit(c)`	`c` символы оналтылық цифр (яғни ондық цифр немесе `'a'..'f'` немесе `'A'..'F'` символдары болып табыла ма?
`isupper(c)`	`c` символы жоғарғы регистрдегі әріп болып табыла ма?
`islower(c)`	`c` символы төменгі регистрдегі әріп болып табыла ма?
`isalnum(c)`	`c` символы әріп немесе ондық цифр болып табыла ма?
`iscntrl(c)`	`c` символы басқару символы (`ASCII 0..31` және `127`) болып табыла ма?
`ispunct(c)`	`c` символы әріп емес, цифр емес, ажыратқыш емес және көрінбейтін басқару символы емес болып табыла ма?
`isprint(c)`	`c` символын баспаға шығаруға бола ма (яғни ол мынадай теру элементі `ASCII ' '..'~'` бола ала ма)?
`isgraph(c)`	`c` символы әріп, цифр немесе пунктуация белгісі (`isalpha()|isdigit()|ispunct()`) болып табыла ма? (Ескерту: бос орын емес)

Бұған қоса, стандартты кітапханада символ регистрлерін өзгерту үшін екі пайдалы функция сипатталған.

Жоғарғы және төменгі регистрлер	
`toupper(c)`	`c` символы немесе оның эквиваленті жоғарғы регистрде
`tolower(c)`	`c` символы немесе оның эквиваленті төменгі регистрде

`Unicode` сияқты кеңейтілген символдар тобы да стандартты кітапханамен сүйемелденеді, бірақ ол тақырып бұл кітап аясынан тыс жатыр.

Б.8.2 Сөз тіркестері

Стандартты кітапханадағы `string` класы `char` символдық типі үшін `basic_string` класы жалпы шаблонының мамандандырылуы болып саналады; басқаша айтқанда, `string` объектісі – бұл `char` типіндегі айнымалылар тізбегі.

Тіркестермен орындалатын операциялар	
`s=s2`	`s` тіркесіне `s2` тіркесін меншіктейді; `s2` операнды `string` класының объектісі немесе С тілі стиліндегі тіркес бола алады.
`s+=x`	`s` тіркесінің соңына `x` операндын қосады; `x` операнды символ, `string` класының объектісі немесе С тілі стиліндегі тіркес бола алады.
`s[i]`	Индекстеу
`s+s2`	Конкатенация; нәтижесі соңына `s2` тіркесінің символдары орналасқан `s` тіркесінен тұратын жаңа тіркес болып табылады
`s==s2`	Тіркестерді салыстыру; `s` және `s2` операндыларының бірі С тілі стиліндегі тіркес бола алады, бірақ бір сәтте екеуі де олай болмайды
`s!=s2`	Тіркестерді салыстыру; `s` және `s2` операндыларының бірі С тілі стиліндегі тіркес бола алады, бірақ бір сәтте екеуі де олай болмайды
`s<s2`	Тіркестерді лексикографикалық салыстыру; `s` және `s2` операндыларының бірі С тілі стиліндегі тіркес бола алады, бірақ бір сәтте екеуі де олай болмайды
`s<=s2`	Тіркестерді лексикографикалық салыстыру; `s` және `s2` операндыларының бірі С тілі стиліндегі тіркес бола алады, бірақ бір сәтте екеуі де олай болмайды
`s>s2`	Тіркестерді лексикографикалық салыстыру; `s` және `s2` операндыларының бірі С тілі стиліндегі тіркес бола алады, бірақ бір сәтте екеуі де олай болмайды
`s>=s2`	Тіркестерді лексикографикалық салыстыру; `s` және `s2` операндыларының бірі С тілі стиліндегі тіркес бола алады, бірақ бір сәтте екеуі де олай болмайды
`s.size()`	`s` тіркесіндегі символдар саны
`s.length()`	`s` тіркесіндегі символдар саны
`s.c_str()`	С тілі стиліндегі символдар тіркесі түріндегі `s` объектісінің нұсқасы (нөлмен аяқталатын)
`s.begin()`	Бірінші символға орнатылған итератор
`s.end()`	`s` тіркесінің соңғы символынан кейінгі символға орнатылған итератор
`s.insert(pos,x)`	`s[pos]` символының алдына `x` объектісін кірістіреді; `x` объектісі символ, `string` класының объектісі немесе С тілі стиліндегі тіркес бола алады.
`s.insert(pos,x)`	`s[pos]` символынан кейін `x` объектісін кірістіреді; `x` объектісі символ, `string` класының объектісі немесе С тілі стиліндегі тіркес бола алады.

Тіркестермен орындалатын операциялар *(жалғасы)*	
`s.append(pos,x)`	`s[pos]` символынан кейін `x` объектісін кірістіреді; `x` объектісі символ, `string` класының объектісі немесе С тілі стиліндегі тіркес бола алады.
`s.erase(pos)`	`s[pos]` позициясындағы символды өшіреді
`s.push_back(c)`	`c` символын тіркес соңына қосады
`pos=s.find(x)`	`s` тіркесіндегі `x` объектісін табады; `x` объектісі символ, `string` класының объектісі немесе С тілі стиліндегі тіркес бола алады; `pos` операнды – ол бірінші табылған символдың индексі немесе `npos` саны (`s` тіркесінің соңынан кейінгі позиция)
`in>>s`	`in` ағымынан `s` тіркесіне сөз енгізеді

Егер `new` операторы арқылы сіз компьютер жадына құрамдас типтегі объектілер орналастырсаңыз, егер инициализатор көрсетілмесе, олар инициалданбайды. Егер `new` операторы арқылы сіз компьютер жадына конструкторы бар класс объектілерін орналастырып, инициализаторды көрсетпесеңіз, онда сол конструктор шақырылады (17.4.4 бөлімді қ.).

`delete` операторы, егер бар болатын болса, әрбір операндтың деструкторын шақырады. Деструктордың виртуалды (A.12.3.1 бөлім) бола алатынына назар аударыңыз.

Б.8.3 Регулярлық өрнектерді салыстыру

Регулярлық өрнектер кітапханасы әлі стандартты кітапхана бөлігі болып табылмайды, бірақ жақында қосылады және сонда оған жеңіл қол жеткізілетін болады, сондықтан біз оны осы бөлімде келтіреміз. Бұдан толығырақ түсініктемелер 23 тарауда баяндалған. Төменде `<regex>` тақырыбының негізгі функциялары берілген.

- Мәліметтер ағымындағы регулярлық өрнекке сәйкес келетін тіркесті *іздеу* (searching) – `regex_search()` функциясымен қамтамасыз етіледі.
- Регулярлық өрнекті тіркеспен (көлемі белгілі) *салыстыру* (matching) – `regex_match()` функциясымен қамтамасыз етіледі.
- *Сәйкестіктерді алмастыру* (replacement of mathes) – `regex_replace()` функциясымен қамтамасыз етіледі. Бұл кітапта сипатталмайды; кәсіби оқулықтарды немесе анықтамалықтарды қ.

`regex_search()` және `regex_match()` функцияларының жұмыс нәтижесі көбінесе `smatch` класының объектісі түрінде бейнеленген сәйкестіктер коллекциясы болып табылады.

```
regex row( "^[\\w ]+(     \\d+)( \\d+)( \\d+)$");
// мәліметтер жолы

while (getline(in,line)) { // мәліметтер жолын тексеру
      smatch matches;
      if (!regex_match(line, matches, row))
          error("bad line", lineno);
      // жолды тексеру:
      int field1 = from_string<int>(matches[1]);
      int field2 = from_string<int>(matches[2]);
      int field3 = from_string<int>(matches[3]);
      // . . .
}
```

Регулярлық өрнектердегі арнайы символдар	
`.`	Кез келген жеке символ ("джокер")
`[`	Символдар класы
`{`	Санауыш
`(`	Топ басы
`)`	Топ соңы
`\`	Келесі символдың ерекше мағынасы бар
`*`	Нөл немесе көп
`+`	Біреу немесе көп
`?`	Міндетті емес (нөл немесе бір)
`|`	Балама (альтернатива) (немесе)
`^`	Жол басы; терістеу
`$`	Жол соңы

Қайталау	
`{ n }`	Дәл `n` рет
`{ n, }`	`n` немесе одан көп рет
`{n,m}`	`n`-нен аз емес және `m` -нен көп емес рет
`*`	Нөл немесе одан көп, яғни `{ n, }`
`+`	Бір немесе одан көп, яғни `{ 1, }`
`?`	Міндетті емес (нөл немесе бір), яғни `{0,1}`

Символдар класы	
`alnum`	Кез келген алфавиттік-цифрлық символ немесе астын сызу символы
`alpha`	Кез келген алфавиттік символ
`blank`	Кез келген ажыратқыш, жолды бөлетін емес

Символдар класы *(жалғасы)*	
`cntrl`	Кез келген басқару символы
`d`	Кез келген ондық цифр
`digit`	Кез келген ондық цифр
`graph`	Кез келген графикалық символ
`lower`	Төменгі регистрдегі кез келген символ
`print`	Кез келген басылатын символ
`punct`	Кез келген пунктуация белгісі
`s`	Кез келген ажыратқыш
`upper`	Жоғарғы регистрдегі кез келген символ
`w`	Кез келген сөздік символы (алфавиттік-цифрлық символ)
`xdigit`	Кез келген оналтылық цифр

Символдардың кейбір кластары аббревиатурамен сүйемелденеді.

Символдар класының аббревиатуралары	
`\d`	Ондық цифр `[[:digit:]]`
`\l`	Төменгі регистрдегі символ `[[:lower:]]`
`\s`	Босорын (босорын символы, табуляция және басқалар) `[[:space:]]`
`\u`	Жоғарғы регистрдегі символ `[[:upper:]]`
`\w`	Әріп, ондық цифр немесе астын сызу символы `(_)` `[[:alnum:]]`
`\D`	`\d [^[:digit:]]` емес
`\L`	`\l [^[:lower:]]` емес
`\S`	`\s [^[:space:]]` емес
`\U`	`\u [^[:upper:]]` емес
`\W`	`\w [^[:alnum:]]` емес

Б.9 Сандық әдістер

C++ тілінің стандартты кітапханасында математикалық (ғылыми, инженерлік және т.с.с.) есептеулер үшін негізгі құрылыс конструкциялары бар.

Б.9.1 Шектік мәндер

C++ тілінің әрбір іске асырылған нұсқасы программалаушылар шектік мәндерді тексеру, алдын ала қорғағыштарды орнату, т.с.с. жасау мақсатында осы құралдарды пайдалануы үшін құрамдас типтердің қасиеттерін анықтайды.

<limits> тақырыбында әрбір құрамдас немесе кітапханалық **T** типі үшін **numeric_limits <T>** класы анықталған. Одан өзге, программалаушы қолданушының **X** сандық типі үшін **numeric_limits<X>** класын анықтай алады. Мысал қарастырайық.

```
class numeric_limits<float> {
public:
    static const bool is_specialized = true;

    static const int radix = 2;   // санау жүйесінің негізі
                                  // (мұнда екілік жүйе)
    static const int digits = 24; // ағымдағы санау жүйесіндегі
                                  // мантисса цифрларының саны
    static const int digits10 = 6;  // мантиссадағы
                                    // ондық цифрлар саны

    static const bool is_signed = true;
    static const bool is_integer = false;
    static const bool is_exact = false;

    static float min(){return 1.17549435E-38F; }  // мысал
    static float max(){return 3.40282347E+38F; }  // мысал

    static float epsilon(){return 1.19209290E-07F;} // мысал
    static float round_error() { return 0.5F; }   // мысал

    static float infinity() {return /* белгілі бір мән */;}
    static float quiet_NaN() {return /* белгілі бір мән */;}
    static float signaling_NaN(){return /* белгілі бір мән */;}
    static float denorm_min() {return min();}

    static const int min_exponent = - 125;    // мысал
    static const int min_exponent10 = -37;    // мысал
    static const int max_exponent = +128;     // мысал
    static const int max_exponent10 = +38;    // мысал

    static const bool has_infinity = true;
    static const bool has_quiet_NaN = true;
    static const bool has_signaling_NaN = true;
    static const float_denorm_style has_denorm =
                                          denorm_absent;
    static const bool has_denorm_loss = false;
```

```
        static const bool is_iec559 = true;
        // IEC C-559 жүйесіне сәйкес
        static const bool is_bounded = true;
        static const bool is_modulo = false;
        static const bool traps = true;
        static const bool tinyness_before = true;

        static const float_round_style round_style =
                                        round_to_nearest;
    };
```

`<limits.h>` және `<float.h>` тақырыптарында бүтін сандар мен жылжымалы нүктелі сандардың негізгі қасиеттерін анықтайтын макростар анықталған.

Шектік мәндер макростары	
`CHAR_BIT`	`char` типіндегі биттер саны (әдетте 8)
`CHAR_MIN`	`char` типінің ең кіші мәні
`CHAR_MAX`	`char` типінің ең үлкен мәні (әдетте 127, егер `char` типінің таңбасы болса және 255 егер `char` типінің таңбасы болмаса)
`INT_MIN`	`int` типінің ең кіші мәні
`INT_MAX`	`int` типінің ең үлкен мәні
`LONG_MIN`	`long` типінің ең кіші мәні
`LONG_MAX`	`long` типінің ең үлкен мәні
`FLT_MIN`	`float` типінің ең кіші оң мәні (мысалы, `1.175494351e-38F`)
`FLT_MAX`	`float` типінің ең үлкен мәні (мысалы, `3.402823466e+38F`)
`FLT_DIG`	Берілген дәлдіктегі ондық цифрлар саны (мысалы, 6)
`FLT_MAX_10_EXP`	Ең үлкен ондық дәреже (мысалы, 38)
`DBL_MIN`	`double` типінің ең кіші мәні
`DBL_MAX`	`double` типінің ең үлкен мәні (мысалы, `1.7976931348623158e+308`)
`DBL_EPSILON`	`1.0+DBL_EPSILON != 1.0` шартын қанағаттандыратын ең кіші мән

Б.9.2 Стандартты математикалық функциялар

Стандартты кітапханада негізгі математикалық функциялар (`<cmath>` және `<complex>` тақырыптарында) анықталған.

Стандартты математикалық функциялар	
`abs(x)`	Абсолюттік шама
`ceil(x)`	`x`-тен үлкен немесе тең (`>= x`) ең кіші бүтін сан
`floor(x)`	`x`-тен кіші немесе тең (`<= x`) ең үлкен бүтін сан
`sqrt(x)`	Квадрат түбір; `x` аргументі теріс емес болуы тиіс
`cos(x)`	Косинус
`sin(x)`	Синус
`tan(x)`	Тангенс
`acos(x)`	Арккосинус; нәтиже теріс емес болып табылады
`asin(x)`	Арксинус; нөлге ең жақын нәтиже қайтарылады
`atan(x)`	Арктангенс
`sinh(x)`	Гиперболалық синус
`cosh(x)`	Гиперболалық косинус
`tanh(x)`	Гиперболалық тангенс
`exp(x)`	Экспонента; негізі e-ге тең
`log(x)`	Натурал логарифм; негізі e-ге тең; `x` аргументі оң сан болуы тиіс
`log10(x)`	Ондық логарифм

Бұл функциялардың аргументтері `float`, `double`, `long double` және `complex` типтегі мән қабылдайтын нұсқалары бар. Бұл функциялардың әрқайсысының қайтаратын мәндерінің типі аргумент типімен бірдей болады.

Егер стандартты математикалық функция математикалық тұрғыдан алғанда, дұрыс нәтиже қайтара алмайтын болса, онда ол `errno` айнымалысын орнатады.

Б.9.3 Комплекс сандар

Стандартты кітапханада комплекс сандар үшін `complex<float>`, `complex<double>` және `complex<long double>` типтері анықталған. `complex<Scalar>` класы, мұндағы `Scalar` – кәдімгі арифметикалық операцияларды сүйемелдейтін бір басқа тип, ол жұмыс істей алады, бірақ программалардың ауысымдылығына кепілдік бере алмайды.

```
template<class Scalar> class complex {
    // комплекс сан – бұл скалярлық мәндер жұбы,
    // негізінде – координаталар жұбы
    Scalar re, im;
public:
    complex(const Scalar &r,const Scalar &i):re(r),im(i){}
    complex(const Scalar & r) :re(r),im(Scalar ()) { }
    complex() :re(Scalar ()), im(Scalar ()) { }
```

```
        Scalar real() { return re; }  // нақты бөлігі
        Scalar imag() { return im; }  // мнимый бөлігі

        // операторлар: =  +=  - =  *=  /=
    };
```

Бұл мүшелерден басқа, `<complex>` класында көптеген пайдалы операциялар қарастырылған.

Комплекс сандар үшін операторлар	
`z1+z2`	Қосу
`z1-z2`	Азайту
`z1*z2`	Көбейту
`z1/z2`	Бөлу
`z1==z2`	Теңдік
`z1!=z2`	Теңсіздік
`norm(z)`	`abs(z)` шамасының квадраты
`conj(z)`	Айқасқан сандар: егер `z` – `{re,im}` жұбы болса, онда `conj(z)` – `{re,-im}` жұбы
`polar(x,y)`	Полярлық координаталар жүйесіндегі комплекс санды `(rho,theta)` бейнелейді
`real(z)`	Нақты бөлігі
`imag(z)`	Мнимый бөлігі
`abs(z)`	`rho` синонимі
`arg(z)`	`theta` синонимі
`out << z`	Комплек сан шығару
`in >> z`	Комплек сан енгізу

Бұған қоса, кешенді сандарға стандартты математикалық функцияларды қолдануға болады (Б.9.2 бөлімін қ.). Ескерту: `complex` класында `<` немесе `%` операциялары жоқ (24.9 бөлімін қ).

Б.9.4 `valarray` класы

`valarray` стандартты класының объектісі – бұл сандардың бір өлшемді жиымы; басқаша айтқанда, ол жиымдар үшін арифметикалық операциялар (24 тараудағы `Matrix` класына ұқсас) және де қиындылар (slices) мен индекстер бойынша қадамдар (strides) қарастырады.

Б.9.5 Жалпыланған сандық алгоритмдер

`<numeric>` бөліміндегі бұл алгоритмдер сандық мәндер тізбегімен орындалатын қалыпты операциялардың жалпы нұсқаларын қамтамасыз етеді.

Сандық алгоритмдер	
`x = accumulate(b,e,i)`	`x` – бұл `i` мен `[b:e)` тізбегі элементтерінің қосындысы.
`x = accumulate(b,e,i,f)`	Жинақтау, мұнда қосындылау орнына `f` функциясы орындалады.
`x = inner_product(b,e,b2,i)`	`x` – `[b:e)` және `[b2:b2+(e-b))` тізбектерінің скалярлық көбейтіндісі, яғни `[b:e)` тізбегіндегі барлық `p1` элементтері үшін `i` мен `(*p1)*(*p2)` сандарының және `[b2:b2+(e-b))` тізбегіндегі барлық сәйкес `p2` элементтерінің қосындысы.
`x = inner_product(b,e,b2,i, f,f2)`	`inner_product`, бірақ `+` және `*` операторларының орнына сәйкесінше `f` және `f2` функциялары орындалады.
`p=partial_sum(b,e,out)`	`[out:p)` тізбегінің `i` элементі `[b:e)` тізбегіндегі `0..i` элементтерінің қосындысы болып табылады.
`p=partial_sum(b,e,out,f)`	`partial_sum`, мұнда `+` операторының орнына `f` функциясы орындалады.
`p=adjacent_difference(b,e,out)`	`[out:p)` тізбегінің `i` элементі `i>1` үшін `*(b+i)-*(b+i-1)` мәніне тең; егер `e-b>0` болса, онда `*out` мәні `*b`-ға тең болады.
`p=adjacent_difference(b,e,out, f)`	`adjacent_difference`, мұнда `-` операторының орнына `f` функциясы орындалады.

Б.10 C тілінің стандартты кітапханасы функциялары

C тілінің стандартты кітапханасы C++ тілінің кітапханасына аздаған өзгертулермен енгізілген. Онда салыстырмалы түрде функциялардың шағын саны қарастырылған, олардың пайдалылығы әртүрлі пәндік салаларда, әсіресе төменгі деңгейдегі программалауда көп жылғы қолдану тәжірибесімен расталған. C тілінің кітапханасы бірнеше санаттарға бөлінген.

- C тілі стиліндегі енгізу-шығару.
- C тілі стиліндегі тіркестер.

- Компьютер жадын басқару.
- Күн-ай мерзімі (дата) және уақыт.
- Қалғандары.

С тілінің кітапханасының бұл кітапта көрсетілгеннен гөрі әжептеуір көптеген функциялары бар; оқырмандарға С тілінен жақсы оқулықтарды, мысалы, Kernighan, Ritchie, *The C Programming Language* (K&R) кітабын пайдалануды ұсынамыз.

Б.10.1 Файлдар

<stdio> тақырыбында сипатталған енгізу-шығару жүйесі файлдарға негізделген. Файлға нұсқауыш (FILE*) файлға да және стандартты **stdin**, **srdout** және **stderr** енгізу-шығару ағымындарына да қатысты болуы мүмкін. Стандартты ағымдарға келісім бойынша қол жеткізуге болады; қалған файлдар тікелей түрде ашылуы тиіс.

Файлды ашу және жабу	
`f=fopen(s,m)`	**m** режимінде **s** атты файл үшін файлдық ағымды ашады.
`x=fclose(f)`	**f** файлдық ағымды жабады; әрекет табысты болса, 0 қайтарады.

Режим – бұл файл қалай ашылатынын анықтайтын бір немесе бірнеше директивасы бар тіркес.

Файлдар режимі	
`"r"`	Оқу үшін
`"w"`	Жазу үшін (алдыңғы мәлімет жойылады)
`"a"`	Қосу үшін (мәліметтер соңына қосылып жазылады)
`"r+"`	Оқу және жазу үшін
`"w+"`	Оқу және жазу үшін (алдыңғы мәлімет жойылады)
`"b"`	Бинарлық; бір немесе бірнеше режимдермен бірге аралас пайдаланылады

Нақты операциялық жүйеде мүмкіндіктер көбірек (көбінесе дәл солай да) болады. Кейбір режимдер араласуы да мүмкін, мысалы, **fopen (foo, rb)** нұсқауы **f((oo** файлын бинарлық режимде оқу үшін ашуға тырысады. **Stdio** және **iostream** кітапханасындағы ағымдарға арналған енгізу-шығару режимдері бірдей болуы тиіс (Б.7.1 бөлімді қ.).

Б.10.2 printf() функциялары топтамасы

С тілінің стандартты кітапханасында ең кең таралған функциялар болып енгізу-шығару функциялары саналады. Дегенмен де, `iostream` кітапханасын қолдануды ұсынамыз, өйткені ол типтер тұрғысынан қарағанда, қауіпсіз және кеңейтілуді де қолдайды. Форматталған `printf()` шығару функциясы өте кең қолданылады (C++ программаларында да қолданылады) және басқа программалау тілдерінде де имитацияланады.

`printf()` функциясы	
`n=printf(fmt,args)`	`stdout` ағымына `fmt` форматтық тіркесін сәйкес орындарына `args` аргументін кірістіре отырып шығарады
`n=fprintf(f,fmt,args)`	`f` файлына `fmt` форматтық тіркесін сәйкес орындарына `args` аргументін кірістіре отырып шығарады
`n=sprintf(s,fmt,args)`	`stdout` С-тіркесіне `fmt` форматтық тіркесін сәйкес орындарына `args` аргументін кірістіре отырып шығарады

Әрбір нұсқадағы `n` саны – бұл жазылған символдар саны, ал табыссыз болған жағдайда – теріс сан болады. Негізінде, `printf()` функциясы қайтаратын мән практикалық тұрғыдан алғанда, әрқашанда есепке алынып қолданылмайды.

`printf()` функциясының жариялануы келесі түрде болады:

```
int printf(const char* format ...);
```

Басқаша айтқанда, бұл функция С тілі стиліндегі тіркес (көбінесе тіркестік литерал) алады, онан кейін кез келген типтегі аргументтердің кез келген санынан тұратын тізім орналасады. Осы қосымша аргументтердің мағынасы форматтық тіркестегі түрлендіру спецификаторлары арқылы, мысалы, `%c` (символ шығару) және `%d` (бүтін сан шығару) түрінде беріледі. Мысал қарастырайық.

```
int x = 5;
const char* p = "asdf";
printf("x мәні '%d' санына тең, ал s '%s' мәніне тең\n",x,s);
```

`%` таңбасынан кейін тұрған символ аргументті өңдеуді басқарады. Бірінші `%` таңбасы бірінші қосымша аргументке (бұл мысалдағы `%d` спецификаторы `x` айнымалысына қолданылады), екінші `%` таңбасы екінші қосымша аргументке (бұл мысалдағы `%s` спецификаторы `s` айнымалысына қолданылады) қатысты болады, т.с.с. Жекелеп айтсақ, жоғарыдағы `printf()` функциясының шақырылуы келесідей нәтиже береді:

```
x мәні '5' санына тең, ал s 'asdf' мәніне тең
```

Сонан соң келесі жолға көшу орындалады.

Негізінде, `%` түрлендіру директивасы мен ол қолданылатын типтің сәйкестігін тексеру мүмкін емес. Мысал қарастырайық.

```
printf("x мәні '%s' санына тең, ал s '%d' мәніне тең\n",x,s);
                                                    // ой!
```

Түрлендіру спецификаторлары тобы біршама көлемді және олар үлкен икемділікті (және де қате жіберудің де көптеген мүмкіндіктерін) қамтамасыз етеді. `%` символынан соң төменде көрсетілген спецификаторлардың бірі орналаса алады:

- `-` Міндетті емес таңба, түрлендірілген мәнді өрістің сол жақ шетіне туралауды білдіреді.
- `+` Міндетті емес таңба, таңбалы типі бар мән алдында әрқашанда `+` немесе `-` таңбасы тұратынын білдіреді.
- `0` Міндетті емес таңба, сандық мәнді туралау үшін алдыңғы нөлдер қолданылатынын білдіреді. Егер формат спецификаторында `-` таңбасы немесе дәлдік көрсетілсе, онда `0` есепке алынбайды.
- `#` Міндетті емес таңба, жылжымалы нүктелі мән ондық нүкте арқылы шығарылатынын білдіреді, тіпті егер санның бөлшек бөлігі тек нөлдерден тұрмаса да соңғы нөлдер шығарылатынын, сегіздік сандар алдында `0` жазылатынын, ал оналтылық сандар алдына `0x` немесе `0X` жазылатынын білдіреді.
- `d` Өріс енін беретін міндетті емес таңба. Егер түрлендірілген мән өріс енінен аз символдардан тұратын болса, өріс енін түгелдей толтыру үшін, ол сол жақтан босорындармен (немесе оң жақтан, егер туралау индикаторы сол жақ шеттен көрсетілсе) толтырылады. Егер өріс ені нөлден басталатын болса, онда мәндерді толықтыру үшін, босорындар орнына нөлдер пайдаланылатын болады.
- `.` Міндетті емес таңба, өріс ені мен келесі цифрлар тіркесі арасындағы ажыратқыш рөлін атқарады.
- *dd* Міндетті емес цифрлар тіркесі, ол дәлдікті береді, яғни `e` және `f` түрлендірулері үшін, ондық нүктеден кейінгі цифрлар санын тағайындайды немесе тіркесте шығаруға болатын символдардың ең үлкен саны.
- `*` Өріс ені немесе дәлдік, ол цифрлар тіркесі емес, `*` символымен беріле алады. Мұндағы жағдайда өріс ені немесе дәлдік бүтін сандық аргумент пен беріледі.
- `h` Міндетті емес символ (`h` әрпі), ол одан кейінгі `d, o, x` немесе `u` спецификаторлары `short int` типіндегі аргументке сәйкес келетінін көрсетеді.

l Міндетті емес символ (**l** әрпі), ол одан кейінгі **d, o, x** немесе **u** спецификаторлары **short int** типіндегі аргументке сәйкес келетінін көрсетеді.

L Міндетті емес символ (**L** әрпі), ол одан кейінгі **e, E, g, G** немесе **f** спецификаторлары **long double** типіндегі аргументке сәйкес келетінін көрсетеді.

% **%** символы баспаға шығатынын көрсетеді. Аргументтер қолданылмайды.

c Қолданылатын түрлендіру типін беретін символ:

d Ондық санға түрлендірілген бүтінсандық аргумент.

i Ондық санға түрлендірілген бүтінсандық аргумент.

o Сегіздік санға түрлендірілген бүтінсандық аргумент.

x Оналтылық санға түрлендірілген бүтінсандық аргумент.

X Оналтылық санға түрлендірілген бүтінсандық аргумент.

f Ондық санға **[–]*ddd.ddd*** түрлендірілген **float** немесе **double** типіндегі аргумент. Ондық нүктеден кейінгі әріптер саны ***d*** аргумент дәлдігін береді. Қажет болса, сан дөңгелектенеді. Егер дәлдік көрсетілмесе, баспаға алты цифр шығарылады; егер дәлдік тікелей түрде **0** символы арқылы берілсе, ал **#** символы көрсетілмесе, онда цифрлар да, ондық нүкте де шығарылмайды.

e Ондық санға ғылыми форматта **[–]*d.ddde+dd*** немесе **[–]*d.ddde–dd*** түрлендірілген **float** немесе **double** типіндегі аргумент, мұндағы нүкте алдында бір цифр тұрады, ал ондық нүктеден кейінгі цифрлар саны аргумент дәлдігіне тең. Қажет болса, сан дөңгелектенеді. Егер дәлдік көрсетілмесе, баспаға алты цифр шығарылады; егер дәлдік тікелей түрде **0** символы арқылы берілсе, ал **#** символы көрсетілмесе, онда цифрлар да, ондық нүкте де шығарылмайды.

E **e** спецификаторы тәрізді жұмыс істейді, бірақ дәреже көрсеткішін шығару үшін жоғарғы регистрдегі **E** әрпі қолданылады.

g **float** немесе **double** типіндегі аргумент **d, e** немесе **f** стилінде шығарылады, бұл форматтардың қайсысы ең үлкен дәлдікті ең аз таңбалар санымен беруіне байланысты жұмыс істейді.

G **g** спецификаторы тәрізді жұмыс істейді, бірақ дәреже көрсеткішін шығару үшін жоғарғы регистрдегі **E** әрпі қолданылады.

c Баспаға символдық аргумент шығарылады. Нөлдік символдар есепке алынбайды.

s Аргумент тіркес (символға нұсқауыш) болып табылады, тіркестегі символдар нөлдік символ кездескенше немесе дәлдікке тең символдар саны шығарылғанша, баспаға шығарыла береді.

p Аргумент нұсқауыш болып табылады. Оның баспаға шығарылуы тілдің жүзеге асырылу ерекшеліктеріне байланысты болып келеді.

- u `unsigned int` типіндегі аргумент ондық санға түрлендіріледі.
- n `printf()`, `fprintf()` және `sprintf()` функциялары арқылы осыған дейін шығарылған символдар саны `int` типіндегі айнымалыға жазылады, ол айнымалыға `int` типіндегі аргументпен байланысқан нұсқауыш сілтеме жасап тұруы тиіс.

Өрістің нөлдік немесе өте кіші мәні ешқашанда шығарылатын мәнді қысқартпайды; шығару тіркесін нөлдермен немесе босорындармен толтыру тек өрістің берілген ені нақты қолданылатын еннен артық болған кезде ғана орындалады.

С тілінде C++ тіліндегі мағынадағы қолданушы типтері жоқ болғандықтан, онда `complex`, `vector` немесе `string` кластары үшін шығару форматын анықтау мүмкіндігі жоқ.

С тілінде стандартты `stdout` шығару ағымы `cout` ағымына сәйкес келеді. Стандартты `stdin` енгізу ағымы С тілінің `cin` ағымына сәйкес келеді. Стандартты `stderr` қателер туралы мәлімдемелер ағымы С тілінің `cerr` ағымына сәйкес келеді. Бұл С және C++ тілдерінің стандартты енгізу-шығару ағымдарының бір-біріне сәйкестігі өте жақын, сондықтан С және C++ тілі стильдеріндегі енгізу-шығару ағымдары тек бір ғана буферді пайдаланып жұмыс істей алады. Мысалы, мәлімет шығарудың бір ғана ағымын жасау үшін, `cout` және `stdout` объектілерімен (мұндай жағдай С және C++ тілдерін араластыра отырып жазған кодтарда жиі кездеседі) орындалатын операциялар комбинациясын қатар пайдалануға болады. Мұндай икемділік шығындарды талап етеді. Жоғарырақ өнімділікке қол жеткізу үшін бір ағыммен жұмыс істеу кезінде `stdio` және `iostream` кітапханаларынан алынған ағымдарды қолданатын операцияларды араластырмау керек. Оның орнына енгізу-шығарудың алғашқы операциясын орындау алдында `ios_base::sync_with_stdio(false)` функциясын шақырыңыз. `stdio` кітапханасында `scanf()` функциясы анықталған, ол `printf()` функциясына ұқсас енгізу операциясы болып табылады. Мысал қарастырайық.

```
int x;
char s[buf_size];
int i = scanf("x мәні '%d' санына тең,
               ал s '%s' мәніне тең\n",&x,s);
```

Мұнда `scanf()` функциясы бүтін санды `x` айнымалысына және ажыратқыш болып саналмайтын символдар тізбегін `s` жиымына оқуға тырысып жатыр. Форматты емес символдар, олар енгізу тіркесінде болуы тиіс деп көрсетіп тұр. Мысал қарастырайық.

```
x мәні '123' санына тең, ал s 'string 'мәніне тең\n"
```

Программа `x` айнымалысына 123 санын және `s` жиымына соңына `0` жазылған `"string"` тіркесін енгізеді. Егер `scanf()` функциясын шақыру жұмысты дұрыс

аяқтаса, нәтижелік мән (алдыңғы шақыруда `i`) меншіктелген аргумент-нұсқауыштар санына (бұл мысалда ол сан 2-ге тең) тең болады; кері жағдайда ол `EOF` мәніне тең болады. Бұл енгізуді индикациялау тәсілі қателерге төтеп бере алмайды (мысалы, егер сіз енгізу тізбегінде `"string"` тіркесінен соң босорын қоюды ұмытып кетсеңіз, не болады?). `scanf()` функциясының барлық аргументтері нұсқауыштар болуы тиіс. Біз мүмкіндігінше, бұл функцияны қолданбауды ұсынамыз.

Егер біз `stdio` кітапханасын пайдалануға мәжбүр болсақ, мәліметтерді қалай енгіземіз? Кең таралған жауаптардың бірі мынадай: "Стандартты `gets()` функциясын пайдаланыңыз".

```
// өте қауіпті код:
char s[buf_size];
char* p = gets(s);        // жолды s жиымына оқиды
```

Егер `p=gets(s)` шақырылса, онда жаңа жолға көшу символы кездескенше немесе файл соңына жеткенше, `s` жиымына символдар оқылады. Мұнда `s` тіркесінің соңына ең соңғы символдан кейін `0` қойылады. Егер файл соңы анықталса немесе қате шықса, онда `p` нұсқауышы `NULL` (яғни 0) болып орнатылады; қарсы жағдайда ол `s`-ке тең болып орнатылады. Ешқашанда `gets(s)` функциясын немесе оның эквивалентін `(scanf("%s",s))` пайдаланбаңыз! Өткен жылдарда вирус жазушылар осылардың осал жақтарын біліп алды: енгізу буферін аса толтыратын енгізу тіркесін туындата отырып, олар программаларға кіріп, компьютерлерге шабуыл жасауды үйреніп алды. `sprintf()` функциясы да осындай буфердің аса толып кетуіне байланысты мәселелерден жапа шегіп жүр.

`stdio` кітапханасында символдарды оқитын және жаза алатын қарапайым әрі пайдалы функциялар бар.

`stdio` кітапханасындағы символдарды енгізу функциялары	
`x=getc(st)`	`st` енгізу ағымынан символ енгізеді; символдың бүтінсандық мәнін қайтарады; егер файл соңы кездессе немесе қате шықса, онда `x==EOF`
`x=putc(c,st)`	`st` шығару ағымына `c` символын жазады; жазылған символдың бүтінсандық мәнін қайтарады; егер қате шықса, онда `x==EOF`
`x=getchar()`	`stdin` ағымынан символ оқиды; символдың бүтінсандық мәнін қайтарады; егер файл соңы кездессе немесе қате шықса, онда `x==EOF`
`x=putchar(c)`	`stdout` ағымына `c` символын жазады; символдың бүтінсандық мәнін қайтарады; егер қате шықса, онда `x==EOF`
`x=ungetc(c,st)`	`st` енгізу ағымына `c` символын қайтарады; символдың бүтінсандық мәнін қайтарады; егер қате шықса, онда `x==EOF`

Бұл функциялардың нәтижесі болып `int` типіндегі сан (`char` типіндегі айнымалы емес немесе `EOF` макросы емес) есептелетініне назар аударыңыз. С тіліндегі программада қалыпты енгізу циклын қарастырайық.

```
int ch;   /* бірақ char ch емес; */
while ((ch=getchar())!=EOF) { /* кез келген бір әрекеттер */ }
```

Ағымға тізбекті түрде екі `ungetch()` шақыруды қолданбаңыз. Мұндай әрекеттің нәтижесін болжауға болмайды, сондықтан программа да ауысымды болмайды.

Біз `stdio` кітапханасындағы барлық функцияларды сипаттаған жоқпыз, толығырақ ақпаратты С тілінен жақсы оқулықтардан, мысалы, `K&R` кітабынан табуға болады.

Б.10.3 С тілі стиліндегі тіркестер

С тілі стиліндегі тіркестер нөлмен аяқталатын `char` типіндегі элементтер жиымы болып табылады. Бұл тіркестер `<cstring>` (немесе `<string.h>`; ескерту: бірақ `<string>` емес) тақырыптарында сипатталған функциялармен өңделеді. Бұл функциялар С тілі стиліндегі тіркестермен `char*` нұсқауыштары (`const char*` нұсқауыштары тек оқуға арналған жады ұяларына сілтеме жасайды) арқылы жұмыс істейді.

С тілі стиліндегі тіркестермен орындалатын операциялар	
`x=strlen(s)`	Символдарды (соңғы нөлді есепке ала отырып) есептейді
`p=strcpy(s,s2)`	`s2` тіркесін `s` тіркесіне көшіреді; `[s:s+n)` және `[s2:s2+n)` диапазондары қабаттасып кетпеуі тиіс; `p=s;` соңғы нөл көшіріледі.
`p=strcat(s,s2)`	`s2` тіркесін `s` тіркесінің соңына көшіреді; `p=s;` соңғы нөл көшіріледі.
`x=strcmp(s, s2)`	Лексикографикалық тәртіпте салыстыру: егер `s<s2` болса, онда `x` – теріс сан; егер `s==s2` болса, онда `x==0;` егер `s>s2` болса, онда `x` – оң сан
`p=strncpy(s,s2,n)`	`strcpy;` `n` символдан артық емес; соңғы нөл көшірілгенде, қате шығуы мүмкін; `p=s`
`p=strncat(s,s2,n)`	`strcat;` `n` символдан артық емес; соңғы нөл көшірілгенде, қате шығуы мүмкін; `p=s`
`x=strncmp(s,s2,n)`	`strcmp;` `n` символдан артық емес
`p=strchr(s,c)`	`p` нұсқауышын `s` тіркесіндегі бірінші `c` символына орнатады
`p=strrchr(s,c)`	`p` нұсқауышын `s` тіркесіндегі соңғы `c` символына орнатады
`p=strstr(s,s2)`	`p` нұсқауышын `s` тіркесіндегі бірінші `c` символына орнатады, сол символдан `s2` тіркесіне тең ішкі тіркес басталуы тиіс

C тілі стиліндегі тіркестермен орындалатын операциялар *(жалғасы)*	
`p=strpbrk(s,s2)`	`p` нұсқауышын `s` тіркесіндегі бірінші символға орнатады, сол символ `s2` тіркесінде болуы тиіс
`x=atof(s)`	`s` тіркесінен `double` типті санды шығарып алады
`x=atoi(s)`	`s` тіркесінен `int` типті санды шығарып алады
`x=atol(s)`	`s` тіркесінен `long int` типті санды шығарып алады
`x=strtod(s,p)`	`s` тіркесінен `double` типті санды шығарып алады; `p` нұсқауышын `double` типіндегі саннан кейінгі бірінші символға орнатады
`x=strtol(s,p)`	`s` тіркесінен `long int` типті санды шығарып алады; `p` нұсқауышын `long` типіндегі саннан кейінгі бірінші символға орнатады
`x=strtoul(s,p)`	`s` тіркесінен `unsigned long` типті санды шығарып алады; `p` нұсқауышын `long` типіндегі саннан кейінгі бірінші символға орнатады

Мынаған назар аударыңыз: C++ тілінде `strchr()` және `strstr()` функциялары типтер қауіпсіздігін (олар, C тіліндегі сияқты `const char*` типін `char*` типіне түрлендіре алмайды) қамтамасыз ету үшін екі-екіден кездеседі; 27.5 бөлімін қ.

Символдарды шығарып алу функциялары санның сәйкесінше форматталған бейнеленуін іздеуде тіркесті C тілі стилінде қарап шығады, мысалы, `"124"` және `"1.4"`. Егер мұндай бейнелену табылмаса, шығарып алу функциясы `0` қайтарады. Мысал қарастырайық.

```
int x = atoi("fortytwo");      /* x 0-ге тең болып шығады */
```

Б.10.4 Жады

Компьютер жадын басқару функциялары `void*` типті (`const void*` нұсқауыштары тек оқуға арналған жады ұяларына сілтеме жасайды) нұсқауыштар арқылы "жалаң жадыда" (типі белгісіз күйде) жұмыс істейді.

C тілі стиліндегі жады басқару функциялары	
`q=memcpy(p,p2,n)`	`p2` нұсқауышы адрестейтін жады аймағынан `p` нұсқауышы (`strcpy` функциясы тәрізді) адрестейтін жады аймағына `n` байтты көшіреді; `[p:p+n)` және `[p2:p2+n)` диапазондары қабаттаспауы тиіс: `q=p`
`q=memmove(p,p2,n)`	`p2` нұсқауышы адрестейтін жады аймағынан `p` нұсқауышы адрестейтін жады аймағына `n` байтты көшіреді; `q=p`

C тілі стиліндегі жады басқару функциялары *(жалғасы)*	
`x=memcmp(p,p2,n)`	`p2` нұсқауышы адрестейтін жады аймағындағы `n` байтты `p` нұсқауышы адрестейтін жады аймағындағы соған эквивалентті `n` байтпен салыстырады; (`strcmp` функциясы сияқты)
`q=memchr(p,c,n)`	`c` символын (`unsigned char` типіне түрлендірілген) `p[0]..p[n-1]` диапазонынан табады да, сол элементке `q` нұсқауышын орнатады; егер `c` символы табылмаса, онда `q=0` болады
`q=memset(p,c,n)`	`c` символын (`unsigned char` типіне түрлендірілген) `p[0]..p[n-1]` диапазонының әрбір ұясына көшіріп жазады; `q=0`
`p=calloc(n,s)`	Бос жады аймағындағы нөлмен инициалданған `n*s` байтты ерекшелейді; егер `n*s` байтты ерекшелеу мүмкін болмаса, онда `p=0`
`p=malloc(s)`	Бос жады аймағындағы инициалданбаған `s` байтты ерекшелейді; егер `s` байтты ерекшелеу мүмкін болмаса, онда `p=0`
`q=realloc(p,s)`	Бос жады аймағындағы `s` байтты ерекшелейді; `p` нұсқауышы `malloc()` немесе `calloc()` функциясының нәтижесі болуы тиіс; егер мүмкін болмаса, `p` нұсқауышы сілтеме жасап тұрған жады аймағын қайта пайдаланады; егер бұл мүмкін болмаса, `p` нұсқауышы адрестейтін жады аймағының барлық байттарын жаңа жады аймағына көшіреді; егер `q` байтты ерекшелеу мүмкін болмаса, онда `q=0`
`free(p)`	`p` нұсқауышы адрестейтін жады аймағын босатады; `p` нұсқауышы `malloc()`, `calloc()` немесе `realloc()` функцияларының нәтижесі болуы тиіс

`malloc()` функциялары және соған ұқсағандары конструкторларды шақырмайды, `free()` фукциясы деструкторларды шақырмайды. Бұл функцияларды конструкторлары мен деструкторлары бар типтерге қолданбаңыз. Бұған қоса, `memset()` функциясы да конструкторы бар типтерге қолданылмауы тиіс.

Атаулары `mem` әріптерінен басталатын функциялар `<cstring>` тақырыбында сипатталған, ал жады аймағын ерекшелейтін функциялар – `<cstdlib>` тақырыбында сипатталған.

27.5.2 бөлімін қ.

Б.10.5 Күн-ай мерзімі және уақыт

`<ctime>` тақырыбында күн-ай мерзімі (дата) және уақытпен байланысқан бірнеше типтер мен функцияларды табуға болады.

Күн-ай мерзімімен және уақытпен байланысқан типтер	
`clock_t`	Қысқа уақыт интервалдарын (бірнеше минуттарға дейін) сақтауға арналған арифметикалық тип
`time_t`	Ұзақ уақыт интервалдарын (жүздеген жылдарға дейін) сақтауға арналған арифметикалық тип
`tm`	Күн-ай мерзімі мен уақытты (1900 жылдан бастап) сақтауға арналған құрылым

`tm` құрылымы шамамен былай анықталады:

```
struct tm {
   int tm_sec;  // минуттың секунды [0:61];
                // 60 пен 61 кәбисә секундтар
   int tm_min;  // сағаттың минуты [0,59]
   int tm_hour; // күннің сағаты [0,23]
   int tm_mday; // айдың күні [1,31]
   int tm_mon;  //  жылдың айы [0,11];
                // 0 - қаңтар (ескерту: [1:12] емес)
   int tm_year; // 1900 ж. бастап жыл;
                // 0 - 1900 ж., 102 - 2002 ж.
   int tm_wday; // жексенбіден басталған күндер [0,6]; 0 - жексенбі
   int tm_yday; // 1 қаңтардан кейінгі күндер [0,365]; 0 - 1 қаңтар
   int tm_isdst;// жазғы кез сағаттары
};
```

Күн-ай мерзімі және уақытпен жұмыс істейтін функциялар:

```
   clock_t clock();
   // программа басталғаннан кейінгі таймер тактыларының саны

   time_t time(time_t* pt);      // ағымдағы календарлық уақыт
   double difftime(time_t t2, time_t t1);  // t2-t1 секундтармен

   tm* localtime(const time_t* pt); // *pt үшін жергілікті уақыт
   tm* gmtime(const time_t* pt);
   // Гринвич бойынша уақыт (GMT) tm *pt, немесе 0 үшін

   time_t mktime(tm* ptm);  // *ptm немесе time_t(-1) үшін time_t

   char* asctime(const tm* ptm);
   // C-тіркесі түрінде *ptm бейнелеу
   char* ctime(const time_t* t) { return asctime(localtime(t)); }
```

`asctime()` функциясын шақыру нәтижесінің мысалы:

```
"Sun Sep 16 01:03:52 1973\n".
```

`(Do_something())` функциясы жұмысының уақытын өлшеу үшін `clock` функциясын пайдалану мысалын қарастырайық.

```
int main(int argc, char* argv[])
{
    int n = atoi(argv[1]);

    clock_t t1 = clock();      // есептеу басы
    if (t1 == clock_t(- 1)) { // clock_t(-1) деген
    //"clock() жұмыс істемейді" дегенді білдіреді
          cerr << "кешіріңіз, таймер жұмыс істемейді\n";
          exit(1);
}

    for (int i = 0; i<n; i++) do_something(); // уақыт циклі

    clock_t t2 = clock();        // есептеу соңы
    if (t2 == clock_t(- 1)) {
          cerr << "кешіріңіз, таймер толып кетті\n";
          exit(2);
    }
    cout << "do_something() " << n << " жұмыс істеді "
         << double(t2-t1)/CLOCKS_PER_SEC << " секунд"
         << " (өлшеу дәлдігі: " << CLOCKS_PER_SEC
         << " of a секунд)\n";
}
```

Бөлу алдында `double(t2-t1)` өрнегін тікелей түрлендіру қажет, өйткені `clock_t` бүтін сан болуы мүмкін. `clock()` функциясы қайтаратын `t1` және `t2` мәндері үшін `double(t2-t1)/CLOCKS_PER_SEC` шамасы екі шақыру арасындағы секундпен берілген уақыттың ең жақсы жүйелік жуықтауы болып табылады.

Егер `clock()` функциясы процессормен сүйемелденбейтін болса немесе уақыт интервалы өте үлкен болып жатса, онда `clock()` функциясы `clock_t(-1)` мәнін қайтарады.

Б.10.6 Басқа функциялар

`<cstdlib>` тақырыбында келесі функциялар анықталған:

Stdlib кітапханасының басқа функциялары	
`abort()`	Программа жұмысын "апатты" түрде аяқтайды
`exit(n)`	Программа жұмысын `n` мәнімен аяқтайды; `n==0` шарты табысты аяқталуды білдіреді
`system(s)`	С тілі тіркесі түрінде бейнеленген команданы орындайды (жүйеге тәуелді болады)
`qsort(b,n,s,cmp)`	`b` нұсқауышынан басталып, `n` элементтер тұратын `s` өлшемді жиымды реттейді, салыстыру үшін `cmp` функциясын пайдаланады.
`bsearch(k,b,n, s,cmp)`	`b` нұсқауышынан басталып, `n` элементтер тұратын `s` өлшемді жиымдағы `k` аргументін іздейді, салыстыру үшін `cmp` функциясын пайдаланады.
`d=rand()`	`d` – `[0:RAND_MAX]` диапазонындағы кездейсоқ сан
`srand(d)`	Бастапқы мән ретінде `d` санын пайдаланып, кездейсоқ сандар тізбегін бастайды

`qsort()` және `bsearch()` функциялары пайдаланатын салыстыру `(cmp)` функциясы келесідей типте болуы тиіс:

```
int (*cmp)(const void* p, const void* q);
```

Басқаша айтқанда, сұрыптау функциясына реттелетін элементтердің типтері белгісіз: ол жиымды байттар тізбегі ретінде қарастырады да, келесі шарттарды қанағаттандыратын бүтін санды қайтарады:

- егер `*p` `*q` -дан аз болатын болса, ол теріс болады;
- егер `*p` мен `*q` тең болатын болса, ол нөлге тең болады;
- егер `*p` `*q`-дан артық болатын болса, ол нөлден үлкен болады;

`exit()` және `abort()` функциялары деструкторларды шақырмайтынын айта кетейік. Егер статикалық объектілер үшін және автоматты түрде (А.4.2 бөлімін қ.) құрылған объектілер үшін деструкторлар шақырғыңыз келсе, аластама туындатыңыз.

Стандартты кітапханадағы функциялар жайлы толығырақ ақпаратты **K&R** кітабынан немесе C++ тілі туралы дұрысырақ басқа анықтамалықтардан табуға болады.

Б.11 Басқа кітапханалар

Стандартты кітапхана мүмкіндіктерін зерттей отырып, сіз, әрине, өзіңізге пайдалы бір заттар бар ештеңе таба алмайсыз. Программалаушылар алдында тұрған

міндеттермен және қол жетімді кітапханалардың орасан зор сандарымен салыстырғанда, C++ тілінің стандартты кітапханасы онша үлкен болып саналмайды. Келесі міндеттерді шешуге арналған көптеген кітапханалар бар:

- Графикалық қолданушы интерфейстері.
- Күрделі математикалық есептеулер.
- Мәліметтер базасына қол жеткізу.
- Желілерде жұмыс істеу.
- XML.
- Күн-ай мерзімі мен уақыт.
- Файлдармен жұмыс істейтін жүйелер.
- Үшөлшемді графика.
- Анимация.
- Тағы басқалары.

Дегенмен бұл кітапханалар стандарт бөлігі болып табылмайды. Сіз оларды Интернеттер таба аласыз немесе өз достарыңыз бен әріптестеріңізден сұрап алуыңызға болады. Тек стандартты кітапхана бөлігі болып табылатын кітапханалар ғана пайдалы болады деп ойлауға болмайды.

Visual Studio ортасымен жұмысты бастау

"Ғалам біз бейнелейтіндей ғана түсініксіз емес, ол біз ойға алатындайдан да түсініксіз"

- Дж. Б.С. Холдейн (J.B.S. Haldane)

Бұл қосымшада сіз программаға кіріп, оны компиляциядан өткізгенге және орындауға жібергенге дейінгі Microsoft Visual Studio C++ жұмыс жасау ортасының көмегімен атқарылатын қадамдар сипатталған.

B.1 Программаны іске қосу

Программаны іске қосу үшін, сізге файлдарды қандай да бір жолмен бір жерге жинақтап алу керек болады (егер олар бір-біріне сілтеме жасап тұрса, мысалы, бастапқы файл тақырыптық файлға, олар бірін-бірі тауып алатындай жағдайда болуы тиіс). Сонан кейін компиляторды және байланыс редакторын (егер тағы бір нәрселер жасау талап етілмесе, ол программаны C++ тілінің стандартты кітапханасымен байланыстыруға мүмкіндік береді) шақырып алып, программаны іске қосу (орындау) керек. Бұл мәселені шешудің бірнеше жолы бар, бірақ әртүрлі операциялық жүйелерде (мысалы, Windows және Linux) әртүрлі келісімдер қабылданып, әртүрлі құралдар жиыны қарастырылған. Дегенмен, кітапта келтірілген мысалдардың барлығын да кең таралған құралдардың бірін пайдаланып, барлық негізгі жүйелерде орындауға болады. Бұл қосымшада осыны кең таралған жүйелердің бірі – Microsoft Visual Studio ортасында қалай жасау керектігі көрсетілген.

Біз өзіміз де, кейбір мысалдарды орындау барысында жаңа және қызықты жүйемен жұмыс істеу алдында кез келген программалаушының бойынан кездесетін таңырқау сезімінде болдық. Мұндай кездерде көмек сұрау керек. Бірақ та көмек алар кезде, сіздің кеңесшіңіз мұны қалай жасау керек екендігін сізге үйретсін, оны сіз үшін өзі жасап шықпасын.

B.2 Visual Studio жұмыс ортасын орнату (инсталляция)

Visual Studio – бұл Windows операциялық жүйесі үшін жасалған интерактивті программа құру ортасы (IDE — interactive development environment). Егер ол сіздің компьютеріңізде орнатылмаған болса, оны сатып алып, қосымшасында келтірілген нұсқауларды орындаңыз немесе тегін таратылатын Visual C++ Express нұсқасын

www.microsoft.com/express/download веб-парағынан жүктеп орнатып алу керек. Мұнда келтірілген сипаттама Visual Studio 2005 нұсқасына сәйкес келеді. Қалған нұсқалар да одан көп алшақ кете қоймайды.

В.3 Программаны құру және іске қосу

Программаны құру және іске қосу келесі қадамдардан тұрады:

1. Жаңа жоба жасау.
2. Жобаға C++ тіліндегі бастапқы файлды қосу.
3. Бастапқы кодты енгізу.
4. Атқарылатын файлды жасау.
5. Программаны орындау.
6. Программаны сақтау.

В.3.1 Жаңа жоба жасау

Visual Studio ортасында "жоба" болып программаны (*қосымша* деп те аталады) құру және оны Windows операциялық жүйесінде орындау ісіне қатынасатын файлдар тобы саналады.

1. Microsoft Visual Studio 2005 пиктограммасын шертіп немесе **Start=>Programs=> Microsoft Visual Studio 2005=>Microsoft Visual Studio 2005** командасын орындап, Visual C++ ортасын ашыңыз.
2. **File** менюін ашып, **New** командасын таңдаңыз да, **Visual C++** опциясын шертіңіз.
3. **Project Types** ішкі бетіндегі **Visual C++** қосқышын орнатыңыз.
4. **Templates** бөліміндегі **Win32 Console Application** қосқышын орнатыңыз.
5. **Name** редакциялау терезесіне жобаңыздың атын, мысалы, **Hello, World!** енгізіңіз.
6. Жобаңыз орналасатын каталогты таңдаңыз. Келісім бойынша мынадай жол **C:\Documents and SettingsWour nameWly DocumentsWisual Studio 2005 Projects** ұсынылады.
7. **OK** батырмасын шертіңіз.

8. Осыдан кейін `Win32 Application Wizard` қосымшасының шебері терезесі ашылуы тиіс.
9. Сұқбат (диалог) терезесінің сол жағындағы **Application Settings** пунктін таңдаңыз.
10. **Additional Options** бөлімінде тұрып, **Empty Project** қосқышын орнатыңыз.
11. **Finish** батырмасын шертіңіз. Енді сіздің консольдік жобаңыз үшін компилятордың барлық орнатулары инициалданған болып саналады.

B.3.2 `std_lib_facilities.h` тақырыптық файлын пайдаланыңыз

Сіздің бірінші программаңыз үшін біз тиянақты түрде **`std_lib_facilities.h`** тақырыптық файлын пайдалануды ұсынамыз, оны мына веб-парақтан көшіріп алуға болады: *www.stroustrup.eom/programming/std_lib_facilities.h*. Оны С.3.1 бөлімінің 6 қадамында таңдап алынған каталогқа көшіріп алыңыз. (Ескерту. Ол файлды HTML-файл түрінде емес, мәтіндік түрде сақтаңыз). Бұл файлды пайдалану үшін өз программаңызға

```
#include " ../../std_lib_facilities.h"
```

жолын кірістіріңіз. `"../../"` символдары компиляторға бұл файл мынадай каталогқа жазылғанын мәлімдейді:

```
C:\Documents and Settings\Your name\My Documents\Visual Studio 2005 Projects,
```

мұны басқа жобалар да тауып ала алады, ал егер сіз оны өз бастапқы файлыңыздың қасына жазсаңыз, онда оны әрбір жаңа жоба каталогына көшіріп жазып отыру керек болады.

B.3.3 Жобаға C++ тіліндегі бастапқы файлды қосу

Сіздің программаңыз аз дегенде бір бастапқы файлдан тұруы тиіс (бірақ көбінесе программалар бірнеше файлдардан тұрады).

1. Меню жолындағы **Add New Item** пиктограммасын шертіңіз (әдетте сол жақтан екінші). Нәтижесінде **Add New Item** сұқбат терезесі ашылады. **Visual C++** санатындағы (категориясындағы) **Code** пунктін таңдаңыз.

2. Шаблондар терезесінен **C++ File (.cpp)** пунктін таңдаңыз. Редакциялау терезесінде өз прогаммаңыздың атын (**Hello, World!**) теріңіз де, **Add** батырмасын шертіңіз.

Сонымен, сіз бос бастапқы файл жасадыңыз. Енді сіз өз программаңыздың мәтінін теруге дайынсыз.

В.3.4 Бастапқы кодты енгізу

Бұл пунктте сіз бастапқы кодты осы ортада тере отырып енгізе аласыз немесе басқа жерден (деректер көзінен) көшіріп алып, енгізуіңізге де болады.

В.3.5 Атқарылатын файлды жасау

Егер сіз өз программаңыздың бастапқы коды мәтінін дұрыс тергеніңізге сенімді болсаңыз, **Build** менюіне кіріп, **Build Selection** командасын орындаңыз немесе осы терезенің жоғарғы жағының оң жақ шетіндегі пиктограммалар тізімінен үшбұрышты пиктограмманы шертіңіз. Жұмыс жасау ортасы сіздің кодыңызды компиляциядан өткізіп, оның байланыстарын да редакциялауға кіріседі. Егер бұл процесс табысты аяқталса, сіз Output терезесінде мынадай мәлімдеме аласыз:

```
Build: 1 succeeded,0 failed,0 up-to-date,0 skipped
```

Кері жағдайда **Output** терезесіне кеткен қателер туралы көптеген мәлімдемелер шығады. Программа қателерін жөндеп түзетіңіз де, **Build Solution** командасын орындаңыз.

Егер сіз үшбұрышты пиктограмманы таңдаған болсаңыз, программада қате болмаса, ол автоматты түрде орындала бастайды. Ал егер сіз **Build Solution** менюі пунктін пайдалансаңыз, онда программаны тікелей орындауға С.3.6 бөлімінде көрсетілгендей түрде жібере аласыз.

В.3.6 Программаны орындау

Барлық қателерді жөндегеннен кейін, **Debug** менюінің **Start Without Debugging** пунктін таңдап алып, программаны орындаңыз.

B.3.7 Программаны сақтау

File менюінде тұрып, **Save All** пунктін шертіңіз. Егер сіз мұны ұмытып, программаны орындау ортасын жабуға тырыссаңыз, ол сізге бұл туралы ескерту жасайды.

B.4 Ары қарай ше...

Программаны орындау ортасының көптеген шексіз қасиеттері мен мүмкіндіктері бар. Ол туралы қам жемеңіз, әйтпесе шатасып қалуыңыз да ғажап емес. Егер сіздің жобаңыз өзін түсініксіз етіп көрсетсе, тәжірибелі досыңыздан көмек сұраңыз немесе жаңа жобаны басынан бастап қайта құрыңыз. Уақыт өте келе біртіндеп оның қасиеттері мен мүмкіндіктеріне көз жеткізіп, тәжірибе жасай беріңіз.

FLTK кітапханасын орнату (инсталляция)

"Егер код және комментарийлер бір-біріне қайшы келсе, онда мүмкін, екеуі де дұрыс емес болар".

- Норм Шрайер (Norm Schreyer)

Бұл қосымшада FLTK графикалық кітапханасының байланыстарын қалай жүктеу, орнату (инсталляция) және түзету (редакциялау) керек екендігі көрсетілген.

Г.1 Кіріспе

Біз FLTK (Fast Light Tool Kit) ("фултик" болып оқылады) кітапханасын өз графикамызды бейнелеу негізі ретінде және графикалық қолданушы интерфейсін жасауға байланысты мәселелерді шешу үшін таңдап алдық, өйткені ол оңай ауысады, салыстырмалы түрде қарапайым және де жеңіл орнатылатын болып табылады. Біз FLTK кітапханасын Microsoft Visual Studio ортасына қалай орнату керек екендігін көрсетеміз, өйткені осы мәселе біздің көптеген студенттерімізді қызықтырады және ең күрделі қиындықтар туғызады. Егер сіз басқа бір жүйені пайдаланатын болсаңыз, онда соның жүктелетін файлдар (Г.3 бөлімі) орналасқан басты каталогынан сол жүйеге байланысты нұсқауды іздеңіз.

Егер сіз ISO C++ стандарты бөлігі болып саналмайтын кітапхананы пайдаланатын болсаңыз, онда сізге оны жүктеп алып орнатқан соң, өз кодыңызда дұрыс пайдалана білуіңіз керек. Бұл мәселе онша қарапайым болмайды, сондықтан FLTK кітапханасын орнату – жай ғана мәселе емес, өйткені тіпті ең жақсы кітапхананың өзін қойып орнату, егер сіз оны бұрын орындамаған болсаңыз, көптеген қиындықтар туғызады. Бұрын мұны жасаған адамдардан кеңес сұраудан ұялмаңыз, бірақ өз жұмысыңызды оларға тапсырмаңыз, тек солардан үйреніңіз.

Нақты жұмыс кезінде кездесетін файлдар мен процедуралар біз айтқаннан гөрі аздап басқаша да бола беретінін айта кетейік. Мысалы, FLTK кітапханасының жаңа нұсқасы шығуы мүмкін немесе сіз Visual Studio нұсқасын өзгертуіңіз ықтимал немесе тіпті басқа бір ортаға ауысып та кетерсіз.

Г.2 FLTK кітапханасын жүктеу

Бір нәрсе жасаудың алдында сіздің компьютеріңізде бұрын FLTK кітапханасы (Г.5 бөлімін қ.) орнатылмаған ба екен, соны тексеріп шығыңыз. Егер олай болмаса, кітапхана файлдарын жүктеңіз.

1. *http:f ltk.org* веб-парағына кіріңіз. (Егер ол орындалмай қалса, оларды осы кітапқа арналған веб-сайттан да көшіріп алуыңызға болады (*www.stroustrup.com/Programming/FLTK*).

2. Навигациялық менюдегі **Download** батырмасын шертіңіз.
3. Сонда суырылып шығатын менюден **FLTK 1.1.x** пунктін таңдап алып, **Show Download Locations** батырмасын шертіңіз.
4. Файлдарды қайдан жүктейтініңізді таңдап, **.zip** кеңейтілуі бар файлды жүктеп шығыңыз.

Сіз алған файл .zip форматында жазылған. Бұл файлдарды желі бойынша тасымалдауға арналған архивтің форматы. Файлдарды архивтен шығарып, қалыпты күйге келтіру үшін сізге арнайы программа керек, мысалы, Windows жүйесінде бұған WinZip және 7-Zip программалары сәйкес келеді.

Г.3 FLTK кітапханасын орнату

Нұсқауларды орындау кезінде екі мәселенің бірі туындауы мүмкін: біздің кітап шыққаннан бері бір нәрселер өзгерген (бұл болып тұрады) немесе сіз терминологияны дұрыс түсінбейсіз (мұндай жағдайда біз еш көмек бере алмаймыз; кешіріңіз). Соңғы жағдайда, досыңызды шақырыңыз, ол бәрін де түсіндіреді.

1. Жүктелген файлды қалпына келтіріп (архивтен шығарып), негізгі **fltk-1.1.?** каталогын ашыңыз. C++ жүйесі каталогында (мысалы, **vc2005** немесе **vcnet**) **fltk.dsw** файлын ашыңыз. Егер сізден барлық ескі жобаларды жаңартуды растауды сұраса, **Yes to All** деп жауап беріңіз.
2. **Build** менюінен **Build Solution** командасын таңдаңыз. Бұл бірнеше минут уақыт алуы мүмкін. Бастапқы код статикалық кітапханаларға (static link libraries) компиляциядан өтеді, сондықтан сізге FLTK кітапханасының бастапқы кодын әрбір жаңа жоба ашқан сайын компиляциядан өткізу керек емес. Процесс аяқталған соң, **Visual Studio** ортасын жабыңыз.
3. **FLTK** кітапханасының негізгі каталогындағы **lib** ішкі каталогын ашыңыз. **C:\Program Files\Microsoft Visual Studio\Vc\lib** каталогына **README.lib** файлынан басқа (олар жетеу болуы тиіс), **.lib** кеңейтілуі бар барлық файлдарды көшіріп жазыңыз (жай жылжыта салмаңыз).
4. FLTK кітапханасының негізгі каталогына қайтып оралыңыз да, **FL** ішкі каталогына **C:\Program Files\Microsoft Visual Studio\Vc\include** каталогын көшіріңіз.

Сарапшылар сізге файлдарды **C:\Program Files\Microsoft Visual Studio\Vc\lib және C:\Program Files\ Microsoft Visual Studio\Vc\include** каталогтарына көшіргеннен гөрі кітапхананы қайта

орнатқан жеңіл болар еді деп айтуы мүмкін. Олардыкі дұрыс, бірақ біз **Visual Studio** ортасы бойынша сарапшы болғалы жатқан жоқпыз ғой. Егер сарапшылар сізге тағы да жөн айтқысы келсе, олардан жақсы балама тәсіл көрсетуді сұраңыз.

Г.4 FLTK кітапханасын Visual Studio ортасында пайдалану

1. Қалыпты процедураға бір өзгерту енгізе отырып, Visual Studio ортасында жаңа жоба құрыңыз: жоба типін таңдай отырып, "Console application" емес, "Win32 project" опциясын таңдаңыз. "empty project" жасап жатқаныңызға сенімді болыңыз; әйтпесе кері жағдайда "software wizard" программасы сіздің жобаңызға бірсыпыра артық кодтарды қосып жібереді, оларды сіз түсіне де алмайсыз және пайдалануыңыз да екіталай.
2. Visual Studio ортасында тұрып, бас менюдегі **Project** командасын таңдаңыз да, суырылып шыққан менюден **Properties** командасын орындаңыз.
3. **Properties** терезесінің сол жақ менюіндегі **Linker** пиктограммасын шертіңіз. Сонда ашылған ішкі менюден **Input** командасын таңдаңыз. Оң жақтағы **Dependencies** редакциялау өрісіне келесі мәтінді енгізіңіз:

   ```
   fltkd.lib wsock32.1ib comctl32.1ib fltkjpegd.lib fltkimagesd.lib;
   ```

   (Келесі қадам керексіз де болып қалуы мүмкін, өйткені қазіргі кезде ол келісім бойынша да орындалады). **Ignore Specific Library** редакциялау өрісіне келесі мәтінді енгізіңіз:

   ```
   libcd.lib;
   ```

4. Бұл қадам керексіз де болып қалуы мүмкін, өйткені қазіргі кезде /MDd опциясы келісім бойынша іске қосылады. Басқа ішкі менюді ашу үшін **Properties** терезесінің сол жақ менюінде C/C++ командасын таңдаңыз. Ішкі менюді ашып, **Code Generation** командасын таңдаңыз. Оң жақ менюдегі **Runtime Library** опциясын **Multi-threaded Debug DLL (/MDd)** опциясына өзгертіңіз. **Properties** терезесін жабу үшін, **OK** батырмасын шертіңіз.

Г.5. Егер барлығы дерлік жұмыс істемесе, қалай тесттен өткіземіз

Жаңа жобада .**cpp** кеңейтілуімен жаңа файл құрыңыз да, келесі кодты енгізіңіз. Ол компиляциядан кінаратсыз өтуі тиіс.

```
#include <FL/Fl.h>
#include <FL/Fl_Box.h>
#include <FL/Fl_Window.h>

int main()
{
    Fl_Window window(200, 200, "Window title");
    Fl_Box box(0,0,200,200,"Hey, I mean. Hello, World!");
    window.show() ;
    return Fl::run();
}
```

Егер мұның бір нәрсесі жұмыс істемесе, келесіні орындаңыз.

- Егер сіз компилятордың **.lib** кеңейтілуі бар файлды табу мүмкін емес деген мәлімдемесін алсаңыз, онда, мүмкін, сіз кітапхананы орнату кезінде бір амалды дұрыс орындамаған боларсыз. Өз компьютеріңіздегі кітапханалық файлдарды сақтау үшін адрес (жол) көрсетілген 3 пунктті мұқият тексеріп шығыңыз.
- Егер сіз компилятордың **.h** кеңейтілуі бар файлды ашу мүмкін емес деген мәлімдемесін алсаңыз, онда сіз программаларды орнату кезінде қате жібергенсіз. Сіздің компьютеріңіздегі тақырыптық файлдарды (**.h**) сақтауға арналған адрес көрсетілген 3 пунктті мұқият тексеріп шығыңыз.
- Егер сіз байланыс редакторының шешілмеген сыртқы сілтемелер туралы айтатын мәлімдемесін алсаңыз, онда мәселе жоба қасиеттерінде жатыр.

Егер біздің кеңестеріміз сізге көмектесе алмаса, досыңызды шақырыңыз.

Графикалық қолданушы интерфейсін іске асыру

"Сіз не істеп жатқаныңызды ақыры түсінген сәтте, бәрі де дұрыс болады"

- Билл Фэйрбэнк (Bill Fairbank)

Бұл қосымшада кері шақыруларды және де **`Window`**, **`Widget`** және **`Vector_ref`** кластарын іске асыру келтірілген. Біз 16-тарауда оқырмандардан нұсқауыштар мен келтіру операторлары туралы білуді талап етпегенбіз, сондықтан толығырақ түсініктемелерді қосымшаға шығардық.

Д.1 Кері шақыруларды іске асыру

Кері шақырулар келесідей түрде іске асырылған:

```
void Simple_window::cb_next(Address, Address addr)
    // Simple_window::next() функциясын pw адресінде
    // орналасқан терезе үшін шақыру
{
    reference_to<Simple_window>(addr).next();
}
```

Сіз 17-тарауды оқып шыққандықтан, сізге `Address` аргументінің типі `void*` болуы керек екендігі түсінікті шығар. Және де, әрине, `reference_to <Simple_window> (addr)` функциясы белгілі бір тәсілмен `void*` типі бар `addr` нұсқауышындағы `Simple_window` класының объектісіне сілтеме жасауы тиіс екені де көрініп тұр. Бірақ, егер сіздің программалаудан тәжірибеңіз болмаса, онда сізге 17-тарауды оқып шыққанша, ешнәрсе де "түсінікті" және "әрине" де бола қоймас, сондықтан адрестерді пайдалануды да толығырақ қарастырайық.

А.17-бөлімінде сипатталғандай, C++ тілі типтің атын көрсететін тәсіл ұсынады. Мысал қарастырайық.

```
typedef void* Address; // Address – бұл void* типінің синонимі
```

Бұл `void*` орнына `Address` атын қолдануға болатындығын көрсетеді. Мұндағы жағдайда `Address` атын қолдана отырып, біз адресті беріп отырғанымызды білдіргіміз келеді және де `void*` дегеніміз – нұсқауыш типінің аты, ал ол әзірше белгісіз, сондықтан оны жасыра тұруға тура келіп тұр.

Сонымен, `cb_next()` функциясы аргумент ретінде `addr` атты `void*` типіндегі нұсқауышты қабылдап алады да, әйтеуір бір тәсілмен оны бірден `Simple_window&` сілтемесіне түрлендіреді:

```
reference_to<Simple_window>(addr)
```

reference_to функциясы шаблондық функция болып табылады (А.13-бөлімі).

```
template<class W>W& reference_to(Address pw)
    // адресті W класының объектісіне сілтеме ретінде қабылдайды
{
    return *static_cast<W*>(pw);
}
```

Мұнда біз **void*** типін **Simple_window&** типіне келтіру ретінде жұмыс істейтін операцияларды өздігімізден жазу үшін шаблондық функцияны пайдаландық. Мұндай **static_cast** типін келтіру әрекеті 17.8 бөлімінде сипатталған болатын.

Компилятордың **addr** аргументі **Simple_window** класының объектісіне сілтеме жасайтындығы жайлы біздің болжамдарымызды тексеруге мүмкіндігі жоқ, бірақ тіл ережелері компилятор бұл мәселеде программалаушыға сенім артуы керек деп талап етеді. Қуанышқа орай, біздікі дұрыс болып шықты. Ол туралы FLTK жүйесінің біз оған берген нұсқауышты ол бізге қайтадан қайтаратыны куәлік етеді. FLTK жүйесіне нұсқауышты бере отырып, біз оның типін білгендіктен, **reference_to** функциясын "оны қайтарып алуға" пайдалануға болады. Мұның бәрі аздап шатасқан түрде болып отыр, олар тексеруді де өтпейді және төменгі деңгейдегі программалау үшін де қалыпты жағдай емес.

Simple_window класының объектісіне сілтеме алып, біз оны **Simple_window** класының функция-мүшесін шақыруға пайдалана аламыз. Мысал қарастырайық (16.3-бөлім).

```
void Simple_window::cb_next(Address,Address pw)
    // pw адресі бойынша орналасқан терезе үшін
    // Simple_window: :next ()  функциясын шақыру
{
    reference_to<Simple_window>(pw).next();
}
```

Біз жай ғана қарапайым **next()** функция-мүшесін шақыруға қажетті типтерді үйлестіру үшін, әжептеуір күрделі **cb_next()** кері шақыру функциясын пайдаландық.

Д.2 Widget класын іске асыру

Біздің **Widget** интерфейстік класы төмендегідей түрде болады:

```
class Widget {
    // Widget класы – бұл Fl_widget класының дескрипторы,
    // ол Fl_widget класының өзі емес;
```

```
    // біз өз интерфейстік кластарымызды
    // FLTK-мен араластырмауға тырысамыз

public:
    Widget(Point xy,int w,int h,const string& s,Callback cb)
        :loc(xy),width(w),height(h),label(s),do_it(cb)
{   }

    virtual ~Widget () { } // деструктор

    virtual void move(int dx,int dy)
    { hide(); pw->position(loc.x+=dx, loc.y+=dy); show();}

    virtual void hide() { pw->hide(); }
    virtual void show() { pw->show(); }

    virtual void attach(Window&) = 0;
    // Widget класының әрбір объектісі, кем дегенде,
    // тереземен орындалатын бір әрекетті анықтайды

    Point loc;
    int width;
    int height;
    string label;
    Callback do_it;

protected:
    Window* own;    // Widget класының әрбір объектісі
                    // Window класының объектісіне жатады
    Fl_Widget* pw; // Widget класының әрбір объектісі
                    // "өзінің" Fl_Widget класын біледі
};
```

Біздің **Widget** класымыз FLTK кітапханасындағы "өз" компоненті мен сонымен байланысты **Window** класын ғана қадағалайтынына назар аударыңыз. Оған қоса, бұл үшін бізге нұсқауыштар керек, өйткені **Widget** класының объектісі өзі жұмыс істеп тұрған уақыт кезеңінде әртүрлі **Window** класының объектілерімен байланыста болуы мүмкін. Бұл үшін сілтеме немесе атаулы объект жеткіліксіз болар еді. (Неге олай екенін түсіндіріңіз).

Widget класы объектісінің өз орны (**loc**), төртбұрышты формасы (**width** және **height**) және де белгісі (**label**) бар. Тағы да оның кері шақыру функциясы (**do_it**) да бар, яғни ол экрандағы **Widget** класының объектісі бейнесін өз кодының фрагментімен байланыстырады. Енді **move()**, **show()**, **hide()** және **attach()** операцияларының мағынасы түсінікті болған шығар.

Widget класы аяқталмаған тәрізді болып көрінеді. Ол қолданушы жиі көруге тиіс емес жүзеге асыру класы сияқты жобаланған болатын. Оны қайта жасап шығу керек. Біз барлық осы ашық мүшелер мен "түсінікті болуы тиіс" операциялардың астыртын қиындықтары да бар деп пайымдаймыз.

Widget класының виртуалды функциясы бар және ол базалық класс ретінде де қолданыла алады, сондықтан оның виртуалды деструкторы да қарастырылған (17.5.2-бөлімді қ.).

Д.3 Window класын іске асыру

Нұсқауышты қашан пайдалануға болады, ал сілтемені ше? Біз бұл сұрақты жалпы түрде 8.5.6 бөлімінде талқылаған болатынбыз. Мұнда біз кейбір программалаушылардың нұсқауыштарды жақсы көретінін, біз әртүрлі объектерге әртүрлі уақыт кезеңдерінде сілтеме жасағымыз келсе, нұсқауыштар бізге керек болатынын айта кетейік.

Бұған дейін біз графикалық кітапханадағы басты класты – **Window** класын жасырын ұстап келдік. Оның басты себебі – ол нұсқауыштарды пайдаланады, ал оның FLTK кітапханасы арқылы іске асуы бос жады аймағын пайдалануға алып келеді. Бұл класс **Window.h** тақырыптық файлында былай сипатталған:

```
class Window : public Fl_Window {
public:
    // жүйеге жады аймағынан орын таңдауға мүмкіндік береді:
    Window(int w, int h, const string& title);
    // xy нүктесіндегі сол жақ жоғарғы бұрыш:
    Window(Point xy, int w, int h, const string& title);

    virtual ~Window() { }

    int x_max() const { return w; }
    int y_max() const { return h; }

    void resize(int ww,int hh) { w=ww,h=hh; size(ww,hh); }

    void set_label(const string& s) { label(s.c_str()); }

    void attach(Shape& s) { shapes.push_back(&s); }
    void attach(Widget&);

    void detach(Shape& s);  // w элементін фигурадан өшіреді
    void detach(Widget& w); // w элементін терезеден өшіреді
                            // (кері шақыруларды алып тастайды)
```

```
        void put_on_top(Shape& p); // p объектісін барлық
                                   // фигуралардың үстіне қояды
    protected:
        void draw();
    private:
        vector<Shape*> shapes; // фигуралар тереземенбайланысады
        int w,h;               // терезе көлемі

        void init();
    };
```

Сонымен, біз `attach()` функциясын пайдалана отырып, фигураны тереземенбайланыстырғанда, біз `Shape` класының объектілерінде сақтаймыз, сондықтан `Window` класының объектісі сәйкес фигураны сызып сала алады. Біздің кейіннен фигураны `detach()` функциясы арқылы ажыратып алуымызға болатындықтан, бізге нұсқауыш керек. Негізінде, байланыстырылған фигура өз кодына жатады; біз жай ғана `Window` класының объектісіне оған сілтеме береміз. `Window::attach()` функциясы кейіннен сақтау үшін, өз аргументін нұсқауышқа түрлендіреді. Жоғарыда көрсетілгендей, `attach()` функциясы қарапайым болып табылады да, `detach()` функциясы аздап күрделілеу болып келеді. `Window.cpp` файлын ашқанда, біз төмендегіні көреміз:

```
    void Window::detach(Shape& s)
        // бірінші болып соңғы қосылған фигура
        // жойылуы тиіс екенін анықтайды
    {
        for (unsigned int i = shapes.size(); 0<i; --i)
          if (shapes[i-1]==&s)
              shapes.erase(&shapes[i-1]);
    }
```

`erase()` функция-мүшесі вектордың мөлшерін бірге кеміте (20.7.1-бөлім) отырып, ондағы бір мәнді жояды (өшіреді). `Window` класы базалық класс ретінде қолданылады, сондықтан оның виртуальды деструкторы бар (17.5.2-бөлім).

Д.4 Vector_ref класын іске асыру

Негізінде, `Vector_ref` класы сілтемелер векторын бейнелейді. Біз оны сілтемелермен немесе нұсқауыштармен инициалдай аламыз:

- Егер объект `Vector_ref` класының объектісіне сілтеме арқылы берілсе, онда ол оның өмірлік (жұмыс істеу) мерзімін басқаратын (мысалы, объект

– бұл белгілі бір көріну аймағында орналасқан айнымалы) шақыру функциясына жатады деп саналады.

- Егер объект `Vector_ref` класының объектісіне нұсқауыш арқылы берілетін болса, онда ол компьютер жадында new операторы арқылы орналасқан болып саналады да, оның жою жауапкершілігі `vector_ref` класына жүктеледі.

Элемент `Vector_ref` класының объектісінде, объект түрінде емес, сілтеме семантикасы бар нұсқауыш түрінде сақталады. Мысалы, `Vector_ref<Shape>` класы векторына `Circle` класы объектісін қию (қима) қауіпсіздігін болдырмай, орналастыруға болады.

```
template<class T> class Vector_ref {
    vector<T*> v;
    vector<T*> owned;
public:
    Vector_ref() {}
    Vector_ref(T* a, T* b = 0, T* c = 0, T* d = 0);

    ~Vector_ref() { for (int i=0; i<owned.size(); ++i)
                    delete owned[i]; }

    void push_back(T& s) { v.push_back(&s); }
    void push_back(T* p)
                { v.push_back(p); owned.push_back(p); }

    T& operator[](int i) { return *v[i]; }

    const T& operator[](int i) const { return *v[i]; }

    int size() const { return v.size(); }
};
```

`Vector_ref` класының деструкторы оған нұсқауыш ретінде берілген әрбір объектіні жояды.

Д.5. Мысал: Widget класы объектілерімен іс-әрекеттер орындау

Бұл аяқталған программа. Ол `Widget/Window` кластарының көптеген қасиеттерін көрсетеді. Біз оған комментарилердің ең аз көлемін ғана қостық. Өкінішке орай, программаларға осылай жеткіліксіз түрде комментарий беру – кең

таралған құбылыстардың бірі. Осы программаны орындап, оның қалай жұмыс істейтіндігін түсіндіруге тырысыңыз:

```
#include "../GUI.h"
using namespace Graph_lib;

class W7 : public Window {
//батырманың жылжи алатындығын көрсететін төрт тәсілді бейнелеу:
// көрсету/жасыру, орнын ауыстыру, жаңасын жасау және қосу/ажырату

public:
    W7(int h, int w, const string& t);

    Button* p1;             // көрсету/жасыру
    Button* p2;
    bool sh_left;

    Button* mvp;            // жылжыту
    bool mv_left;

    Button* cdp;            // жасау/жою
    bool cd_left;

    Button* adp1;           // екпінді ету/екпінсіз ету
    Button* adp2;
    bool ad_left;

    void sh();              // әрекеттер
    void mv();
    void cd();
    void ad();

    static void cb_sh(Address, Address addr) // кері шақырулар
        { reference_to<W7>(addr).sh(); }
    static void cb_mv(Address, Address addr)
        { reference_to<W7>(addr).mv(); }
    static void cb_cd(Address, Address addr)
        { reference_to<W7>(addr).cd(); }
    static void cb_ad(Address, Address addr)
        { reference_to<W7>(addr).ad(); }
};
```

Дегенмен, `W7` класының объектісі (нөмірі 17 `Window` класының объектісімен тәжірибе) негізінде алты батырмадан тұрады; оның екеуін ол жасырып тұрады:

```
W7::W7(int h, int w, const string& t)
    :Window(h,w,t),
    sh_left(true),mv_left(true),cd_left(true),
                                    ad_left(true)
{
    p1 = new Button(Point(100,100),50,20,"show",cb_sh);
    p2 = new Button(Point(200,100),50,20,"hide",cb_sh);

    mvp = new Button(Point(100,200),50,20,"move",cb_mv);

    cdp = new Button(Point(100,300),50,20,"create",cb_cd);

    adp1 = new Button(Point(100,400),50,20,
                                    "activate",cb_ad);
    adp2 = new Button(Point(200,400),80,20,
                                     "deactivate",cb_ad);

    attach(*p1);
    attach(*p2);
    attach(*mvp);
    attach(*cdp);
    p2->hide();
    attach(*adp1);
}
```

Бұл класта төрт кері шақыру бар. Оның әрқайсысы басылған батырманың өшірілуімен және оның орнына жаңасы пайда болуымен байқалады. Бірақ бұл төрт түрлі тәсілдермен атқарылады:

```
void W7::sh()  // батырманы жасырады да, келесісін көрсетеді
{
   if (sh_left) {
      p1->hide();
      p2->show();
   }
   else {
      p1->show();
      p2->hide();
   }
   sh_left = !sh_left;
}
```

```
void W7::mv()           // батырманы жылжытады
{
    if (mv_left) {
        mvp->move(100,0);
    }
    else {
        mvp->move(- 100,0);
    }
    mv_left = !mv_left;
}

void W7::cd()       // батырманы жояды да, жаңасын жасайды
{
    cdp->hide();
    delete cdp;
    string lab = "create";
    int x = 100;
    if (cd_left) {
        lab = "delete";
        x = 200;
    }
    cdp = new Button(Point(x,300), 50, 20, lab, cb_cd);
    attach(*cdp);
    cd_left = !cd_left;
}

void W7::ad()       // батырманы  терезеден ажыратады да,
                    // оның ауысуымен байланыс орнатады
{
    if (ad_left) {
        detach(*adp1);
        attach(*adp2);
    }
    else {
        detach(*adp2);
        attach(*adp1);
}
    ad_left = !ad_left;
}

int main()
{
    W7 w(400,500,"mole");
    return gui_main();
}
```

Бұл программа терезе элементтерін қосу мен жоюдың негізгі тәсілдерін бейнелейді, олар элементтердің жоғалып кетуі мен пайда болуы арқылы көрсетіледі.

Глоссарий

"Көбінесе, дәл таңдалып айтылған сөздер мыңдаған суреттерге татиды"

- Жасырын автор

Глоссарий – бұл мәтінде пайдаланылған сөздердің қысқаша түсініктемесі. Төменде программалауды үйренудің әсіресеб алдыңғы кезеңдерінде біз маңызды деп санайтын терминдердің салыстырмалы түрдегі қысқаша сөздігі келтірілген. Әрбір тараудың соңындағы салалық көрсеткіш пен "Терминдер" бөлімі де оқырмандарға осы жағынан көмек бере алады. C++ тілімен тығыз байланысты толығырақ және ауқымды терминдер сөздігін *www.research.att.com/~bs/glossary.html* веб-парағынан тауып алуға болады. Оған қоса веб ортасында өте көптеген арнайы глоссарийлер (сапалары әртүрлі) бар. Терминдердің бірнеше мағыналары (олардың кейбірін біз көрсетіп отырамыз) болатыны да есте болсын, біз көрсеткен терминдердің бірсыпырасының басқа бір сөз тіркестерімен бірге қолданылғанда, мағынасының өзгеріп кететіні де бар; мысалы, біз *абстрактылы* ((abstract) сөзін қазіргі сурет өнеріне, заңгерлік тәжірибеге немесе философияға байланысты сын есім түрінде анықтамаймыз.

RAII қағидасы ("Resource Acquisition Is Initialization"). Көріну аймағы тұжырымдамасына (концепциясына) негізделген ресурстарды басқарудың негізгі технологиясы.

Абстрактылы класс (abstract class). Объектілер жасау үшін тікелей қолдануға болмайтын класс; Көбінесе туынды кластардың интерфейстерін анықтау үшін қолданылады. Егер кластың таза виртуалды функциясы немесе қорғалған конструкторы бар болса, ол абстрактылы болып табылады.

Абстракция (abstraction). Болмыстың сипатталу нақтылықтарын (мысалы, жүзеге асырылу нақтылықтары) ерікті немесе еріксіз түрде есепке алмайтын (жасыратын) түрі; селективті түрдегі білімсіздік.

Адрес (address). Объектіні компьютер жадынан тауып алуға мүмкіндік беретін мән.

Айнымалы (variable). Берілген типтегі атаулы объект; егер инициалданса, мәні болады.

Ақпаратты жасыру (information hiding). Интерфейс пен оны іске асыруды бір-бірінен ажыратуға байланысты әрекет, мұның нәтижесінде іске асыру нақтылықтары қолданушы назарынан тыс қалады да, абстракция туындайды.

Алгоритм (algorithm). Мәселені шешуге арналған процедура немесе формула; Тізбектердің нәтижеге алып келетін шектелген есептеу қадамдары.

Алғышарт (pre-condition). Код фрагментіне, мысалы, функцияға немесе циклге, кірерде орындалуы тиіс шарт.

Алмастыру (override). Аты мен аргументтерінің типтері базалық кластағы виртуалдық функцияның аты мен типтерімен бірдей болып келетін туынды кластағы функцияны анықтау; нәтижесінде бұл функцияны базалық кластың интерфейсі арқылы шақыруға болады.

Альтернативті ат (alias). Объектіні пайдалануға арналған баламалы (альтернативті) тәсіл; көбінесе атау, нұсқауыш немесе сілтеме.

Анықтау (definition). Программада пайдалану үшін барлық қажетті ақпаратты сақтайтын болмысты жариялау. Қарапайым анықтау: жады бөлетін жариялау түрі.

Аппроксимация (approximation). Кемшіліксіздікке немесе идеалға (санға немесе жобаға) жақын бір нәрсе (мысал, сан немесе жоба). Көбінесе аппроксимация қағидалар арасындағы компромисс нәтижесі болып табылады.

Аргумент (argument). Функцияға немесе шаблонға қол жеткізу үшін параметр түрінде берілетін мән.

Асыра жүктелу (overload). Аттары бірдей, бірақ аргументтерінің (операндтарының) типтері әртүрлі болып келетін екі функцияны немесе операторды анықтау.

Атқарылатын код (executable). Компьютерде орындауға дайын программа.

Базалық класс (base class). Кластар иерархиясының базасы ретінде қолданылатын класс. Әдетте, базалық кластың бір немесе бірнеше виртуалдық функциялары болады.

Байланыстар редакторы (linker). Объектілік код файлы мен кітапханаларды атқарылатын модульге біріктіретін программа.

Байт (byte). Компьютерлердің басым бөлігінде адрестеу үшін қолданылатын негізгі бірлік. Көбінесе, байт сегіз биттен тұрады.

Бастапқы код (source code). Басқа программалаушылардың оқуына да жарамды болып келетін программалаушы құрған код.

Бастапқы файл (source file). Бастапқы кодты сақтайтын файл.

Бит (bit). Компьютердегі ақпарат мөлшерін өлшеуге арналған негізгі бірлік. Бит 0 немесе 1 мәнін қабылдай алады.

Бүтін сан (integer). Математикалық тұрғыдан алынған бүтін сан, мысалы, 42 және -99.

Бірлік (unit). 1) Мәнге мағына беретін стандартты өлшем (мысалы, қашықтық үшін км); 2) бүтіннің ажыратылатын (яғни аты бар) бөлігі.

Виртуалдық функция (virtual function). Туынды класта алмастыруға болатын функция-мүше.

Деструктор (destructor). Объектіні жою үшін жанамалы түрде шақырылатын операция (мысалы, көріну аймағының соңында). Әдетте ресурстарды босатады.

Дефект (ақау; bug). Программадағы қате.

Диапазон (range). Бастапқы және соңғы нүктелерін беру арқылы сипаттауға болатын мәндер тізбегі. Мысалы, [0:5) диапазоны 0,1,2,3 және 4 мәндерін алуды білдіреді.

Дөңгелектеу (rounding). Мәнді өзіне жақын, бірақ дәлдігі төмендеу мәнге математикалық ережелерге сәйкес түрлендіру.

Дұрыстық (correctness). Егер программа немесе фрагмент өз спецификацияларына сәйкес келсе, ол дұрыс болып саналады. Өкінішке орай, спецификация толық болмайды немесе қайшылықты болып, қолданушы күткен ақылға сиымды нәрселерге сәйкес келмей жатады. Сонымен, қабылдауға тұратын код жасау үшін, біз кейде жай ғана формальды спецификацияны сақтаудан гөрі басымырақ заттар істеуіміз қажет.

Енгізу (input). Есептеуүшін пайдаланылатын мәндер (мысалы, функция аргументтері немесе пернетақтада терілген символдар.

Есептеу (computation). Әдетте, кірістік ақпаратты алатын және нәтижені қалыптастыратын белгілі бір кодты орындау.

Жалған код (pseudo code). Программалау тілі емес, бейформальді (жасанды емес) белгілеулер арқылы жазылған есептеулерді сипаттау.

Жалпыланған программалау (generic programming). Алгоритмдерді жобалауға және тиімді іске асыруға бағытталған программалау стилі. Жалпыланған алгоритм оның талабына сай келетін кез келген типтегі аргументтермен жұмыс істеуі тиіс. C++ тілінде жалпыланған программалау әдетте, шаблондарды пайдаланады.

Жариялау (объявление; declaration). Типі бар атау спецификациясы.

Жасыру (hiding). Ақпаратқа қол жеткізбеуге арналған әрекет. Мысалы, қабаттасқан (ішкі) көріну аймағынан алынған ат, оны қамтитын (сыртқы) көріну аймағындағы атпен бірдей болған кезде, ол тікелей пайдалану үшін қол жеткізетіндей болмауы тиіс.

Жиым (массив; array). Элементтердің біртекті тізбегі, әдетте, нөмірленген болады, мысалы [0:max).

Жоба (design). Программалық жабдықтаманың өз спецификациясына сәйкес келуі үшін қалай жұмыс істейтінін көрсететін жалпы сипаттама.

Жөндеп түзету (отладка; debugging). Программаның қатесін тауып оны түзету; әдетте, тесттен өткізуге қарағанда, шағындау жүйелік сипатта болады.

Жүзеге (іске) асыру (implementation). 1) Кодты жазу және тесттен өткізуді білдіретін әрекет; 2) программаны жүзеге асыратын код.

Жүйе (system). 1) компьютердегі белгілі бір міндетті атқаруға арналған программа немесе программалар жиыны; 2) "операциялық жүйе" сөздер тіркесінің қысқашасы, яғни программаларды орындаудың базалық ортасы және компьютер құралдары.

Идеал (ideal). Біз қол жеткізуге ұмтылатын тиянақты нұсқа. Әдетте, біз компромисске келісуге мәжбүр боламыз да, сол идеалға тек жақындаумен ғана шектелеміз.

Инвариант (invariant). Программаның берілген нүктесінде (нүктелерінде) әрқашанда орындалатын шарт; әдетте, объектіні немесе циклді қайталау нұсқауына кіру алдында сипаттау үшін қолданылады.

Инициалданбаған (uninitialized). Инициалдауға дейінгі (анықталмаған) объектінің қалып-күйі.

Инициалдау (initialize). Объектіге бірінші (бастапқы) мәнді меншіктеу.

Инкапсуляция (encapsulation). Іске асыру нақтылықтарына рұқсатсыз қол жеткізуден сақтау.

Интерфейс (interface). Код фрагментін (мысалы, функцияны немесе класты) шақыру тәсілін анықтайтын жариялану немесе жарияланулар тобы.

Итератор (iterator). Тізбек элементін анықтайтын объект.

Итерация (iteration). Код фрагментінің қайталанып орындалуы; *рекурсия* ұғымын қараңыз.

Класс (class). Қолданушы анықтаған тип, ол мәлімет-мүшелерден, функция-мүшелерден және тип-мүшелерден тұра алады.

Код (code). Программа немесе программаның бөлігі; ол бастапқы немесе объектілі түрде бола алады.

Компилятор (compiler). Бастапқы кодты объектілік кодқа айналдыратын программа.

Компромисс (trade-off). Жобалау мен іске асырудың бірнеше қағидаларын үйлестіру нәтижесі.

Константа (constant). Өзгертуге (берілген көріну аймағында) болмайтын мән.

Конструктор (constructor). Объектіні инициалдайтын (құрастыратын) операция. Әдетте конструктор инвариантты орнатады және объектілерді пайдалануға қажетті ресурстарды (деструктор босататындарды) жиі сұрайды.

Контейнер (container). Элементтері (басқа объектілер) бар объект.

Көріну аймағы (scope). Болмыс атауына сілтеме жасауға болатын программа мәтіні аймағы (бастапқы код).

Күрделілік (complexity). Дәл анықталуы қиынға түсетін, мәселені шешуді іздеу процесіндегі қиындық мөлшерін көрсететін түсінік немесе сол шешудің өз қасиеті. Кейде *күрделілік* (жай ғана) деп алгоритмді орындауға қажетті операциялар санын бағалауды түсінеді.

Кітапхана (library). Көптеген программалар пайдалана алатын құралдар (абстракциялар) жиынтығы түрінде іске асырылған типтер, функциялар, кластар және т.с.с. тобы.

Қалып-күй (қалып; state). Мәндер жиынтығы.

Қате (error). Программа тәртібіне қатысты (көбінесе талаптар түрінде немесе қолданушыға арналған жетекші құрал ретінде) ақылға сыйымды күтілген жайттар мен программаның негізінде не істейтіні арасындағы сәйкессіздік.

Қима (қиып алу; truncation). Типті басқа типке түрлендіру нәтижесінде ақпаратты жоғалту, ол түрлендірілген мәнді дәл өрнектей алмаудан туындайды.

Қосымша (application). Қолданушылар болмыс ретінде қарастыратын программа немесе программалар топтамасы (коллекциясы).

Құн (cost). Программаны өндіруге немесе оны орындауға байланысты шығындар (мысалы, программалаушының жұмыс істеуге жіберген уақыты, программаның орындалу уақыты немесе жады көлемі). Негізінде, құн жұмыс күрделілігіне тәуелді болуы тиіс.

Литерал (literal). Тікелей санды беретін белгілеу түрі, мысалы, 12 литералы "он екіге" тең бүтін санды береді.

Мәлімет (data). Есептеуде қолданылатын мәндер.

Мән (value). Компьютер жадындағы типке сәйкес тағайындалатын биттер жиыны.

Мүмкіндіктерді қабаттастыру (feature creep). Программаға "болғаны дұрыс" деп қосыла салатын артық функционалдық мүмкіндіктер енгізуге талпыныс.

Нақты класс (concrete class). Объектілерін құруға болатын класс.

Нұсқау (assertion). Программаның осы нүктесінде қандай шарт әрқашанда орындалуы тиіс екендігін орнататын (assert) программаға енгізілген нұсқау.

Нұсқауыш (pointer). 1) Компьютер жадындағы типі бар объектіні идентификациялау (анықтап алу) үшін қолданылатын мән; 2) осындай мәні бар айнымалы.

Объект (object). 1) Берілген типтегі бір мән жазылған инициалданған жады аймағы; 2) жады аймағы.

Объектіге бағытталған программалау (object-oriented programming). Кластар және кластар иерархияларын жобалау мен пайдалануға бағытталған программалау стилі.

Объектілік код (object code). Байланыс редакторы үшін кіріс ақпараты болып саналатын компилятор жұмысының нәтижесі, ол өз кезегінде атқарылатын код жасайды.

Объектілік файл (object file). Объектілік коды бар файл.

Ондық нүктелі сан (floating-point number). Нақты санның компьютерлік аппроксимациясы, мысалы 7.93 және 10.78e-3.

Операция (operation). Белгілі бір әрекетті, мысалы, функцияны немесе операторды, орындайтын нәрсе.

Өзгертілетін (mutable). Өзгертілмейтін объектілерге, константаларға және айнымалыларға қарағанда, өз қалып-күйін өзгерте алатын болмыс.

Пайдалану прецеденті (use case). Тесттен өткізу мен мүмкіндіктерін көрсету үшін программаны пайдаланудың нақты (көбінесе қарапайым) мысалы.

Парадигма (paradigm). Программалаудың немесе жобалау стилінің аздап претенцияланған атауы. Көбінесе барлық қалғандарынан басым түсетін парадигма бар деп (қате пікір) есептейді.

Параметр (parameter). Функция немесе шаблон үшін тікелей кіріс ақпаратын жариялау. Шақыру кезінде функция аргументтерін өз параметрлерінің аттары бойынша пайдалана алады.

Программалау (programming). Есепті шығаруды код түрінде өрнектейтін өнер.

Программалау тілі (programming language). Программаларды өрнектеуге арналған тіл.

Программалық жабдықтама (software). Код фрагменттері мен солармен байланысқан мәліметтер жиынтығы; көбінесе "программа" сөзінің синонимі ретінде қолданылады.

Программалық код (өзімен байланысты мәліметтермен бірге болуы мүмкін). Компьютерде орындалуға толығынан дайын тұрған нәрсе.

Регулярлық өрнек (regular expression). Шаблондарды символдар тіркесі түрінде белгілеу.

Рекурсия (recursion). Функцияның өзін-өзі шақыруы; итерация ұғымын да қараңыз.

Ресурс (resource). Иеленуге болатын және кейіннен ол босатылуы тиіс нәрсе, мысалы, файлдар дескрипторлары, бұғаттау немесе жады аймағы.

Соңғы шарт (post-condition). Код фрагментінен шығарда, мысалы, функциядан немесе циклден, орындалуы тиіс шарт.

Сөз (word). Компьютер жадының негізгі бірлігі, әдетте, бүтін сандарды сақтау үшін қолданылады.

Спецификация (specification). Код фрагменті жасауы тиіс нәрсені сипаттау.

Стандарт (standard). Бір нәрсенің, мысалы, программалау тілінің ресми түрде келісілген анықтамасы.

Стиль (style). Тіл мүмкіндіктерін келісілген түрде пайдалануды қамтамасыз ететін программалау әдістерінің жиыны. Кейде аттарды таңдау ережелеріне және программа мәтінінің сыртқы түріне байланысты өте шектеулі мағынада қолданылады.

Супертип (supertype). Базалық тип; туынды типтің де қасиеттерінің ішкі бөліктері бойында бар тип.

Сілтеме (reference). 1) Компьютер жадындағы типі бар мәннің орнын сипаттайтын мән; 2) осындай мәні бар айнымалы.

Таза виртуалдық функция (pure virtual function). Туынды класта ауыстырылуы тиіс виртуалдық функция.

Тақырып (header). Программа бөліктері арасындағы интерфейстерді бөлуге арналған жариялаулары бар файл.

Талап (requirement). 1) Программаның немесе программа бөлігінің қажетті тәртібін сипаттау; 2) шаблонның немесе функции аргументтерінің күтілетін мүмкіндіктерін сипаттау.

Тесттен өткізу (testing). Программадағы қателерді жүйелі түрде іздеу.

Тип (type). Объектілердің мүмкін болатын мәндер жиынын және солармен орындауға болатын операцияларды анықтайтын нәрсе.

Толып кету (overflow). Өзіне берілген жады аймағында (сыймайтын) сақтауға болмайтын мән жасау.

Туынды класс (derived class). Бір немесе бірнеше базалық кластардың мұрагері болып табылатын класс.

Тізбек (sequence). Тізбекті түрде қарастырып шығуға болатын элементтер жиыны.

Тіркес (string). Символдар тіркесі.

Тіршілік мерзімі (lifetime). Инициалдау сәтінен бастап, объект қажет болмай қалатын кезге дейінгі уақыт аралығы (көріну аймағынан шығу кезінде жойылады немесе программа жұмысы аяқталуына байланысты әрекеті тоқтатылады).

Файл (file). Компьютердің тұрақты жадында ақпарат сақтайтын контейнер.

Функция (function). Программаның әртүрлі бөлігінен екпінді етуге (шақыруға) болатын кодтың атаулы бірлігі; есептеулердің логикалық бірлігі.

Цикл (loop). Қайталанып орындалатын код фрагменті; көбінесе C++ тілінде **for** немесе **while** нұсқаулары арқылы жүзеге асырылады.

Шаблон (template). Бір немесе бірнеше типтермен не мәндермен (компиляциядан өткізу кезеңінде) параметрленген класс немесе функция; жалпыланған программалауды сүйемелдейтін C++ тіліндегі негізгі конструкция.

Шексіз рекурсия (infinite recursion). Шақыруларды сақтауға қажетті компьютер жады таусылғанға дейін ешқашанда аяқталмайтын рекурсия. Практикада мұндай рекурсия ешқашанда шексіз болмайды, ол аппараттық жабдықтаманың қатесі нәтижесінде тоқталады.

Шексіз цикл (infinite loop). Аяқталу шарты ешқашанда орындалмайтын цикл. **Итерация** (iteration) ұғымын қараңыз.

Шығару (output). Есептеу нәтижесінде құрылған мәндер (мысалы, функция жұмысының нәтижесі немесе экранға шығарылған символдар тіркесі).

Ішкі тип (subtype). Туынды тип; базалық типтің барлық қасиеттерін сақтап, қосымша мүмкіндіктерге де ие бола алатын тип.

Әдебиеттер

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition* (usually called "The Dragon Book"). Addison-Wesley, 2007. ISBN 0321547985.

Andrews, Mike, and James A. Whittaker. *How to Break Software: Functional and Security Testing of Web Applications and Web Services*. Addison-Wesley, 2006. ISBN 0321369440.

Austern, Matthew H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999. ISBN 0201309564.

Austern, Matt, *ed. Draft Technical Report on C++ Standard Library Extensions*. ISO/IECPDTR 19768. www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf.

Bergin, Thomas J., and Richard G. Gibson, eds. *History of Programming Languages* — Volume 2. Addison-Wesley, 1996. ISBN 0201895021.

Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall, 2006. ISBN 0131872493.

Boost.org. "*A Repository for Libraries Meant to Work Well with the C++ Standard Library*." www.boost.org.

Cox, Russ. "Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, . . .)." http://swtch.com/~rsc/regexp/regexp1.html.dmoz.org. http://dmoz.org/Computers/Programming/Languages.

Freeman, T. L., and Chris Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992. ISBN 0136515975.

Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0201633612.

Goldthwaite, Lois, *ed. Technical Report on C++ Performance.* ISO/IEC PDTR 18015.
www.research.att.com/~bs/performanceTR.pdf.

Gullberg, Jan. *Mathematics — From the Birth of Numbers*. W. W. Norton, 1996. ISBN 039304002X.

Hailpern, Brent, and Barbara G. Ryder, *eds. Proceedings of the Third ACMSIGPLAN Conference on the History of Programming Languages* (HOPL-III). San Diego, CA, 2007. http://portal.acm.org/ toc.cfm?id=1238844. 534_stroustrup_biblio_bk 11/20/2008 8:30 AM Page 1177

Henricson, Mats, and Erik Nyquist. *Industrial Strength C++: Rules and Recommendations.* Prentice Hall, 1996. ISBN 0131209655.

ISO/IEC 9899:1999. *Programming Languages — C.* The C standard.

ISO/IEC 14882:2003. *Programming Languages — C++.* The C++ standard.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, first edition, 1978; second edition, 1988. ISBN 0131103628.

Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1998. ISBN 0201896842.

Koenig, Andrew, ed. *The C++ Standard*. ISO/IEC 14882:2002. Wiley, 2003. ISBN 0470846747.

Koenig, Andrew, and Barbara E. Moo. *Accelerated C++: Practical Programming by Example.* Addison-Wesley, 2000. ISBN 020170353X.

Langer, Angelika, and Klaus Kreft. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference.* Addison-Wesley, 2000. ISBN 0201183951.

Lippman, Stanley B., Josée Lajoie, and Barbara E. Moo. *The C++ Primer.* Addison-Wesley, 2005. ISBN 0201721481. (Use only the 4th edition.)

Lockheed Martin Corporation. *"Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program.*" Document Number 2RDU00001 Rev C. December 2005. Colloquially known as "JSF++." www.research.att.com/~bs/ JSF-AV-rules.pdf.

Lohr, Steve. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts — The Programmers Who Created the Software Revolution.* Basic Books, 2002. ISBN 9780465042265.

Maddoc, J. boost::regexp documentation. www.boost.org and www.boost.org/doc/libs/1_36_0/libs/regex/doc/html/index.html.

Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library.* Addison-Wesley, 2001. ISBN 0201749629.

Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition.* Addison-Wesley, 2005. ISBN 0321334876.

Musser, David R., Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Second Edition. Addison-Wesley, 2001. ISBN 0201379236.

Programming Research. *High-integrity C++ Coding Standard Manual Version 2.4.* www.programmingresearch.com.

Richards, Martin. *BCPL — The Language and Its Compiler.* Cambridge University Press, 1980. ISBN 0521219655.

Ritchie, Dennis. *"The Development of the C Programming Language." Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. ACM SIGPLAN Notices, Vol.28 No. 3, 1993.

Salus, Peter. *A Quarter Century of UNIX*. Addison-Wesley, 1994. ISBN 0201547775.

Sammet, Jean. *Programming Languages: History and Fundamentals*, Prentice Hall, 1969. ISBN 0137299885.

Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2002. ISBN 0201604647.

Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2003. ISBN 0201795256.

Schwartz, Randal L., Tom Phoenix, and Brian D. Foy: *Learning Perl, Fourth Edition.* O'Reilly, 2005. ISBN 0596101058.

Scott, Michael L. *Programming Language Pragmatics.* Morgan Kaufmann, 2000. ISBN 1558604421.

Sebesta, Robert W. *Concepts of Programming Languages, Sixth Edition.* Addison-Wesley, 2003. ISBN 0321193628.

Shepherd, Simon. "The Tiny Encryption Algorithm (TEA)." www.tayloredge.com/reference/Mathematics/TEA-XTEA.pdf and http://143.53.36.235:8080/tea.htm.

Stepanov, Alexander. www.stepanovpapers.com.

Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions.* SIAM, 1998. ISBN 0898714141.

Stone, Debbie, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation.* Morgan Kaufmann, 2005. ISBN 0120884364.

Stroustrup, Bjarne. *"A History of C++: 1979–1991." Proceedings of the ACM History of Programming Languages Conference (HOPL-2). ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.

Stroustrup, Bjarne. *"Learning Standard C++ as a New Language."* C/C++ Users Journal, May 1999.

Stroustrup, Bjarne. *The C++ Programming Language (Special Edition)*. Addison-Wesley, 2000. ISBN 0201700735.

Stroustrup, Bjarne. "*C and C++: Siblings*"; "*C and C++: A Case for Compatibility*"; and "*C and C++: Case Studies in Compatibility."* The C/C++ Users Journal, July, Aug., and Sept. 2002.

Stroustrup, Bjarne. "*Evolving a Language in and for the Real World: C++ 1991–2006.*" Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III). San Diego, CA, 2007. http://portal.acm.org/toc.cfm?id=1238844.

Stroustrup, Bjarne. Author's home page, www.research.att.com/~bs.

Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2000. ISBN 0201615622.

Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, 2004. ISBN 0321113586.

University of St. Andrews. *The MacTutor History of Mathematics archive.* http://wwwgap.dcs.st-and.ac.uk/~history.

Wexelblat, Richard L., ed. *History of Programming Languages*. Academic Press, 1981. ISBN 0127450408.

Whittaker, James A. *How to Break Software: A Practical Guide to Testing.* Addison-Wesley, 2003. ISBN 0321194330.

Wood, Alistair. *Introduction to Numerical Analysis.* Addison-Wesley, 1999. ISBN 020134291X.

Пәндік көрсеткіш

A

B

C

D

E

F

G

H

I

J

K

L

V

X

А

Б

Н

О

Ө

П

Р

С

Т

У

Ү

Ф

Ц

Ш

Ы

І

Э

Фотосуреттер

14-бет Бьярне Страуструптың фотосуреті, 2005. Дерек көзі: Bjarne Stroustrup.

15-бет Лоуренс «Пит» Петерсеннің фотосуреті, 2006. Дерек көзі: Dept. of Computer Science, Texas A&M Unversity.

26-бет Casio фирмасының цифрлік сағатының суреті. Дерек көзі: www.casio.com.

26-бет Casio фирмасының аналогтық сағатының суреті. Дерек көзі: www.casio.com.

26-бет MAN 12K98ME – кеменің дизельдік қозғалтқышы; MAN Burgmeister & Waine. Дерек көзі: MAN Diesel A/S, Copenhagen, Denmark.

26-бет Emma Maersk; әлемдегі ең ірі контейнер тасушылардың бірі; тіркелу порты – Århus, Denmark. Дерек көзі: Getty Images.

28-бет цифрлық телефон коммутаторы Дерек көзі: Alamy Images.

28-бет музыкалық жүйесі, телефондық функциялары және веб-ке шығу мүмкіндіктері бар Sony-Ericsson W-920 ұялы телефоны. Дерек көзі: www.sonyericsson.com.

29-бет Уолл-Стриттегі Нью-Йорк қор биржасының операциялық залы. Дерек көзі: Alamy Images.

29-бет Интернетті құратын бөліктердің Стивен Айк (Stephen G. Eick) жасаған графикалық бейнесі. Дерек көзі: S. G. Ei k.

30-бет CAT сканері. Дерек көзі: Alamy Images.

30-бет Компьютерлік хирургия. Дерек көзі: Da Vinci Surgical Systems, www.intuitivesurgical.com.

31-бет Әдеттегі компьютерлік жабдықтар (сол жақтағы экран Unix жүйесімен басқарылатын үстелдік жүйемен, ал оң жақтағы экран Windows операциялық жүйесі орнатылған ноутбукпен байланысқан). Дерек көзі: Bjarne Stroustrup.

31-бет Серверлік түйіндегі компьютерлер тағаны. Дерек көзі: Istockphoto.

33-бет Марс бетін зерттеу құрылғысынан көрінісі. Дерек көзі: NASA, www.nasa.gov.

264-бет EDCAS командасы 1949 ж. Ортада Морис Уилкс (Maurice Wilkes), галстуксіз – Дэвид Уилер (David Wheeler). Дерек көзі: The ambridge University Computer Laboratory.

264-бет Дэвид Уилер дәріс оқып тұр (шамамен 1974 ж.). Дерек көзі: The ambridge University Computer Laboratory.

267-бет Джон Бэкус (John Backus, 1996). Copyright: Louis Fabian Bachrach. Компьютерлік эра пионерлерінің фотографиялары топтамасын Christopher Morgan, *Wizards and their wonders: portraits in computing,* ACM Press Press, 1997. ISBN 0-89791-960-2 кітабына қ.

269-бет Грейс Мюррей Хоппер (Grace Murray Hopper). Дерек көзі: Computer History Museum.

269-бет Грейс Мюррей Хоппер (Grace Murray Hopper) қоңызы. Дерек көзі: Computer History Museum.

271-бет Джон Маккарти (John C. McCarty) 1967 ж., Станфорд. Дерек көзі: Stanford Unversity.

271-бет Джон Маккарти 1996 ж. Дерек көзі: Louis Fabian Bachrach.

273-бет Питер Наур (Peter Naur) фотографиясы, Брайан Ранделл (Brian Randell) Мюнхенде 1968 жылы түсірген, мұнда олар компьютерлік ғылымдар дамуына түрткі болған есеп беру құжатын бірге даярлап жатыр. Брайан Ранделлдің рұқсатымен жарияланып отыр.

273-бет Питер Наур, портретті Дуо Дуо Жанг (Duo Duo Zhuang) 1995 жылы маймен жазған. Эрик Фрокьяердің (Erik Frakjaer) рұқсатымен жарияланып отыр.

274-бет Эдсгер Дейкстра (Edsger Dijkstra). Дерек көзі: Wikimedia Commons.

276-бет Никлаус Вирт (Nisklaus Wirth). Дерек көзі: N. Wirth.

276-бет Никлаус Вирт (Nisklaus Wirth). Дерек көзі: N. Wirth.

278-бет Жан Ишбиа (Jean Ichbiah). Дерек көзі: Ada Information learinghouse.

278-бет Леди Лавлейс (Ladu Lovelace), 1838. Ескі фотография. Дерек көзі: Ada Information learinghouse.

280-бет Кристен Нюгард (Kristen Nygaard) және Оле-Иохан Дал (Ole-Johan Dal), шамамен 1968 ж. Дерек көзі: University of Oslo.

281-бет Кристен Нюгард, шамамен 1996 жыл. Дерек көзі: University of Oslo.

231-бет Оле-Йохан Дал, 2002. Дерек көзі: University of Oslo.

283-бет Деннис Ритчи (Dennis M. Rit hie) және Кен Томпсон (Ken Thompson), шамамен 1978 ж. Copyright: AT&T Bell Labs.

283-бет Деннис Ритчи, 1996. Дерек көзі: Louis Fabian Bachrach.

284-бет	Дуг Макилрой (Doug Mcllroy), шамамен 1990 ж. Дерек көзі: Gerard Holzmann.
284-бет	Брайан Керниган (Brian W. Kernigan), шамамен 2004 ж. Дерек көзі: B.W. Kernigan
286-бет	Бьярне Страуструп, 1996. Дерек көзі: Bjarne Stroustrup.
287-бет	Алекс Степанов (Alex Stepanov), 2003. Дерек көзі: Bjarne Stroustrup.
504-бет	AT&T Bell Labs' Murray Hill Research Center, шамамен 1990 ж. Дерек көзі: AT&T Bell Labs.

Бьярне Страуструп

ПРОГРАММАЛАУ.
С++ тілін пайдалану қағидалары мен тәжірибесі.

2-том

Оқулық

Басуға 21.11.2014 ж. қол қойылды. Қағазы офсеттік.
Қаріп түрі «Times». Пішімі 70x100$^{1/16}$. Баспа табағы 47,75.
Таралымы: Мемлекеттік тапсырыс бойынша – 1000 дана.
Тапсырыс № 1213.

Тапсырыс берушінің дайын файлдарынан басылып шықты.

ЖШС РПБК «Дәуір», 050009,
Алматы қаласы, Гагарин д-лы, 93а.
E-mail: rpik-dauir81@mail.ru, zakaz@dauir.kz