Коммерциялық емес
акционерлік
қоғам

АЛМАТЫ

ЭНЕРГЕТИКА

ЖӘНЕ БАЙЛАНЫС

Ақпараттық

жүйелер

кафедрасы

ОБЪЕКТІГЕ БАҒЫТТАЛҒАН ПРОГРАММАЛАУ

5В070300 – Ақпараттық жүйелер, 5В060200 – Информатика мамандықтарының студенттеріне арналған

Алматы, 2016

ҚҰРАСТЫРУШЫ: А.С.Сәрсенбай. «Объектіге бағытталған программалау». 5В070300-Ақпараттық жүйелер, 5В060200 – Информатика мамандықтарының студенттеріне арналған дәрістер жинағы. – Алматы: АЭжБУ, 2016. – 58 б.

Дәрістер жинағы «Объектіге бағытталған программалау» таңдау курсына сәйкес 5В070300 - Ақпараттық жүйелер, 5В060200 - Информатика мамандықтарының студенттеріне арналған. Дәрістер жинағында объектіге бағытталған программалау негіздері, абстрактты тип ретіндегі класс, ағынды кластар, үлгілер (шаблоны), мұрагерлік, кластар иерархиясы, полиморфизм, класс достары, кластарға арналған стандартты операцияларды қайта жүктеу, әрекеттер аймақтары мен атаулар кеңістіктері, файлдық енгізу-шығару кластары қарастырылған.

Библиография –10.

Пікір беруші: т.ғ.к., доцент Кожамбердиев К.О.

«Алматы энергетика және байланыс университеті» коммерциялық емес акционерлік қоғамының 2016 жылғы жоспары бойынша басылады.

Введение

Основное различие между современными лидирующими объектно-ориентированными языками программирования C++ и Java и процедурными языками состоит в том, что первые имеют синтаксис для использования объектно-ориентированных понятий, таких как наследование, инкапсуляция, полиморфизм и так далее

Язык C++ имеет в своих стандартных библиотеках большое количество уже написанного низкоуровневого кода, но главным образом он приспособлен для использования коллекций и низкоуровневых структур данных, таких как стеки и очереди. Обычные задачи программирования, такие как работа с базами данных, изображениями, Интернетом реализованы не в самом языке C++, а в виде дополнительных библиотек операционной системы.

Язык C++ основан на C, который известен как высокопроизводительный язык за счет простоты использования. C++ - это компилирующий язык. Это означает, что когда вы пишите исходный код, вы должны откомпилировать его и скомпоновать, чтобы он стал выполнимым для определенной платформы, и перекомпилирован для выполнения на другой платформе. Так как пользовательский интерфейс отличается на различных платформах, перенос приложения на C++ - основная проблема при переносе программ с одной операционной системы на другую. C++ не предоставляет никакой безопасности. Любая программа на C++ может получать доступ к любой части памяти и к любым ресурсам. C++ не поддерживает сбор мусора, то есть возможность языка автоматически, как в языке Java, очищать ресурсы(память). В C++, выделив блок памяти, вы отвечаете за его освобождение. Язык C++ не предоставляет никаких элементов пользовательского интерфейса, кроме консоли ввода-вывода.

Мазмұны

1 Дәріс №1. Объектіге бағытталған программалау технологиясы. C++ тілі

Дәрістің мақсаты: объектіге бағытталған программалау технологиясы түсініктерімен таныстыру.

Дәрістің мазмұны: класс ұғымы және оның элементтері. C++ тілі ортасының артықшылықтары.

Программалаушылар көптеген жылдар бойы бизнесті жүргізу әдістерін автоматтандыратын программалаларды жазып үлкен жұмыс атқарды. Бизнесте программалар – бизнес операцияларына қызмет етуі талап етеді. Программалаушыларға көмекке қайта пайдаланылатын кодтар (reusable code) есебінен, жұмысты жеңілдететін процедуралық программалау келді. Процедуралық программалау программалар жазуда үлкен өзгеріс болды, бірақ объектіге бағытталған әлемді қайта құру мәселесі шешілмеген күйінде қалды. Бұл мәселе 1980 жылы объектіге бағытталған программалаудың (ОБП) пайда болуымен шешілді. Нақты әлемнің объектілеріне еліктеуді C++ немесе Java сияқты объектіге бағытталған программалау тілдерінде кластарды анықтау арқылы жүзеге асыруға болады. Класс деректер мүшелері мен әдістер мүшелерінен тұрады. Деректер мүшелері (*data members*) кейде өрістер (*fields*) деп аталады және объект атрибуттарын сақтау үшін пайдаланылады. Әдістер мүшелері (*member methods*) кейде функциялық мүшелер (*member functions*) деп аталады және объектінің жәй-күйін анықтайды. Қазіргі уақытқа сай алдыңғы қатарлы C ++ және Java объектіге бағытталған программалау тілдерімен процедуралық тілдердің арасындағы негізгі айырмашылық, біріншісінде мұраға, инкапсуляция, полиморфизм сияқты т.б. объектіге бағытталған тұжырымдамаларды пайдалануға арналған синтаксисі бар екенінде.

C++ тілінің стандартты кітапханаларында көп мөлшерде жазылған төменгі деңгейлі коды бар, бірақ негізінен бұл стек және кезек сияқты төмен деңгейлі деректер құрылымдардың жинақтарын пайдалану үшін бейімделген. Деректер қорымен, суреттермен, интернетпен жұмыс істеу сияқты қарапайым программалау есептері C++ тілінің өзінде емес, операциялық жүйелердің қосымша кітапханалары түрінде іске асырылған.

C++ тілі пайдалануға қарапайымдылығына байланысты өнімділігі жоғары тіл ретінде белгілі C тіліне негізделген. C++ - бұл компиляциялайтын тіл. Бұл, бастапқы кодты жазғанда, ол анықтаған платформа үшін орындалатындай етіп, оны компиляциядан және компоновкадан өткізіп, басқа платформада орындалуы үшін қайта компиляция жасауды білдіреді.

Қолданушы интерфейсі әртүрлі платформаларда ерекшеленетіндіктен, қосымшаны C++-ке көшіру - программаны бір операциялық жүйеден басқасына көшіру барысындағы басты мәселе. C++ ешқандай қауіпсіздікті қамтамасыз етпейді. C++ тілінде программа жадының кез келген бөлігіне және кез келген ресурстарға қол жеткізе алады.

C++ қоқыс жинауды қолдамайды, яғни оның мүмкіндігі Java тіліндегі сияқты ресурстарды (жады) автоматты түрде тазалайды. C++ тілінде жады блогын бөле отырып, сіз оның босауына жауап бересіз. C++ тілі енгізу-шығару консолінен басқа ешқандай қолданушы интерфейсінің элементтерін ұсынбайды. Ол мұнда Windows жүйесіндегі сияқты ешқандай графикалық пайдаланушы интерфейсін қолдаудың жоғын білдіреді. Windows ортасының өзінің C++ те пайдалануға болатын жеке программалық интерфейсі бар (*application programming interface*, API). C ++ тілі көпжақты мұрагерлік қолдау көрсетеді, ол дегеніміз бір кластың бір мезгілде бірнеше базалық кластары болуы мүмкін дегенді білдіреді.

C++ шаблондар (*templates*) аталымды механизмін ұсынады. Шаблондар печенье пісіретін формалар сияқты кластар мен функцияларды анықтау барысында, жаңа кластар мен функцияларды құру үшін қолданылады.

C++ тілінің түйінді сөздері программа объектілерінің жады кластарымен, типтердің аттарымен, көріну спецификациясымен, модификаторлармен және операторлармен байланысты. C++ тілі операциясының белгілері өрнектерді қалыптастыру және әрі қарай есептеуді қамтамасыз етеді. Мәнді алу үшін өрнек қағида болады. Операцияның бір белгісі әртүрлі өрнектерде қолданылу мүмкін және контекстке байланысты әртүрлі интерпретациялануы мүмкін. Әрине, барлық операциялардың өз рангісі болып, приоритеттер иерархиясын құрап, ассоциативтік (оңнан солға немесе солдан оңға) ережеге бағынады.

C++ тілінің операторлары мен функциялары өз кезегінде классикалық алгоритмдік тілдердің құрылысын мұра еткен C тілімен толықтай дерлік сәйкес келіп, шамалы жетілдірілген. Операторлар әдеттегідей іс-әрекеттерді және осы іс-әрекеттердің программада орындалу логикасын (ретін) анықтайды. C++ тілінде әртүрлі функциялардың атауы бірдей болуы мүмкін. Бұл жағдайда әртүрлі функциялар ретінде бір-бірінен формальді параметрлер санымен немесе типімен ерекшеленетін функцияларды түсінеміз. Тек қана кері қайтарылатын мәннің типі бойынша айырмашылықтар рұқсат етілмейді. *Inline* модификаторы бар функциялар алмастыру функциялары деп аталады. Ереже бойынша, бұл барлық жұмысы жалғыз оператормен атқарылатын және оның көмегімен программада бірнеше рет қолданылатын кодты жүзеге асыратын функциялар.

C++ тілінде бар және C тілінде жоқ кейбір мүмкіндіктерді қарастырайық:

- C++ тілінде комментарияларды - // белгісі арқылы енгізуге болады;
- C++ тілінде енгізу-шығарудың қосымша жабдықтары бар;
- функцияны қайта жүктеу - бірнеше функциялар үшін бірдей атауларды пайдалану. Бір функцияға кейде әр типті объектілермен ұқсас әрекеттерді орындау талап етіледі;

-программада динамикалық жады облысын таңдап және жоюға мүмкіндік беретін *new* және *delete* операциялары;

-көріну аймағын кеңейту операциясы.

Кластар және оның әдістері (ОБП түйінді ұғымдары):

- конструкторлар мен деструкторлар кластар объектілерін құратын және жоятын арнайы әдістер болып табылады;

- кластың мүшесі болмайтын, бірақ ондағы сипатталған айнымалылар мен әдістерге қол жеткізуге мүмкіндігі бар достық функциялары;

- қайта жүктеу операциялары – тек стандартты типтегі деректерге ғана емес, әртүрлі кластардың объектілеріне де қолдануға мүмкіндік беретін тілдің көптеген негізі операцияларының мағынасын өзгерту мүмкіндігі;

- туынды кластар. Осы класс объектілері барлық ашық айнымалылар мен түбірлік (родительский) класс әдістерін мұра етеді, бірақ оған қоса өзінің жеке айнымалылары мен әдістері болуы мүмкін.

- полиморфизм - виртуалды функциялар жабдықтары арқылы жүзеге асырылады;

- хабарламалар көмегімен объектілерді басқаруға болады. Объектіге хабарламаны жіберу функцияны шақыруды еске түсіреді.

Бақылау сұрақтары:

1) Компьютерлік жүйелерді құру барысында объектіге бағытталған программалауды қолдану неліктен маңызды?
2) Мұралықтың объектіге бағытталған программадағы орнын түсіндіріңіз.
3) ОБП күрделі компьютерлік жүйелерді қандай жолмен қолдауға көмек береді?
4) Адам объектісінен мұра етілетін Университет объектілерін сипаттаңыз.

2 Дәріс №2. Деректер типінің сілтемелері

Дәрістің мақсаты: объектіге сілтемені қолданудың әдістерін ұғындыру.

Дәрістің мазмұны: сілтемені анықтау және оны қолдану әдістері. Нұсқауыштан айырмашылығын көрсету. Функцияға сілтемені пайдалану.

C++ тілінде нұсқауышқа туыс элемент – сілтеме (reference) бар. Сілтеме барлық жағдайда жасырын нұсқауыш болып, оны кезкелген мақсат үшін айнымалының тағы бір аты ретінде қолдануға болады. Сілтемені үш түрлі әдіспен қолдануға болады. Біріншіден, сілтемені функцияға жіберуге болады. Екіншіден, оны функциядан қайтарып алуға болады. Ақырында, тәуелсіз сілтемені құруға болады. Сілтеменің ең маңызды қолданылуы – бұл оның функция параметрі ретінде берілуі.

Сілтемені анықтау үшін & белгісі пайдаланылады, егер ол келесі контексте қолданылатын болса: *type* & сілтеме_ аты инициализатор;

Сол жақтағы мәні болатын инициализатор ретінде болуы міндетті, яғни жадыда орын алатын объектінің атауы болуы керек. Инициализацияны

анықтаудан кейін сілтеменің мәні сол объектінің адресі болады. Мысалы: *int* k=2; *int*& ref=k; сілтемені анықтауда‘&’ символы типтің бөлігі болып табылмайды, яғни ref *int* типті және дәл осылай программада қабылдануы керек.

Функциональді түрде сілтеме өзін-өзі қарапайым типті сияқты айнымалы ретінде ұстайды. Нұсқауыш үшін керек болатындай, сілтеме «қарайтын» жады бөлігінің мәнін иемдену үшін қайта атауды нақты орындау қажет емес.

Сілтеме айнымалы не көрсеткішке ұқсас толық құқылы объект емес. Инициализациядан кейін сілтеменің мәнін өзгертуге болмайды, ол әрқашан инициализациямен байланысқан жады бөлігіне «қарайды». Бір де бір операция сілтемеге әсер етпей, онымен байланысқан объектіге қатысты болады. Мұны сілтеменің негізгі қасиеті деп есептеуге болады. Сонымен, сілтеме толықтай бастапқы объект атымен бірдей.

Анықталған болсын:

```
int ar[]={1,2,3,4};      //ar - массив
int *p=ar;               //p - нұсқауыш
int & ref=ar[0];         //ref – массивтің бірінші элементіне сілтеме
int*& rp=ar;             //нұсқауышқа сілтеме (массив атына)
```

Келтірілген мысалдан сілтемелер мен нұсқауыштарға келесі теңдіктер орындалады:

p==&ref, *p==ref, rp==ar; ref== ar[0].

Сілтемеге Sizeof операциясын қолдану нәтижесі оның өлшемі емес, онымен аталған объектінің өлшемі болып табылады.

Сілтеме шын объект болмаған соң, сілтемені анықтауда және қолдануда шектеулер бар. Біріншіден, сілтеме *void* типті бола алмайды. Сілтемені *new* операциясы көмегімен құруға болмайды, яғни сілтемеге жаңа жады бөлігін бөлуге болмайды. Басқа сілтемелерге сілтеме анықталмаған. Сілтемелерге нұсқауыштар жоқ және сілтеме массивін құру мүмкін емес.

Сілтеменің мәні өзгермегенімен, бір объект сілтемелер және нұсқауыштардың кез келген санымен адрестелуі мүмкін:

```
int k=7;          // k айнымалысы анықталған
int &r1=k;      // r1сілтемесі  k-мен байланысты
int *p=&r1;   // p нұсқауышы k-ні адрестейды
int &r2=r1;    // r2 сілтемесі k-ны көрсетеді
```

Енді k айнымалысының мәніне төрт әдіспен жетуге болады:

- k атауымен;
- r1 сілтемесі;

- *p нұсқауышы және r2 сілтемесі арқылы.

Сілтеме айнымалыға қарағанда, сілтеме өзімен байланысқан тұрақты объектінің мәнін өзгертуге мүмкіндік бермейді. Мысалы:

```
int k=5;
const int &ref=k;        //ref – тұрақтыға сілтеме
```

Осы анықтамалардан:

```
k =0;        // қалыпты меншіктеу операторы
ref =0;     // қате, себебі сілтеме тұрақтыға жарияланған
```

Сілтемелерді анықтауда оларды міндетті түрде атаулау (инициализация) талап етіледі. Алайда, сілтемелердің сипаттамасында оның болуы міндетті емес, бірақ та тыйым салынбайды.

Осындай сілтемелердің сипаттамаларына мыналар қатысты:

- сыртқы сілтемелер сипаттамасы (*extern* ерекшелегішімен (спецификаторымен));
- класс компоненттеріне сілтемелер сипаттамасы;
- функцияның формальды параметрлерін ерекшелеу;
- функция қайтаратын мән типінің сипатталуы.

С++ тіліне сілтемелерді енгізудің негізгі себебі (тек қана С++ үшін емес), функциялармен параметрлер аппараты арқылы алмасу тиімділігін және функцияны шақыруды сол тұрғыдағы мән ретінде мақсатты қолдану мүмкіндігін арттыру қажеттілігінен болды. Мысалы:

```
int f1(int &k)              // функция бүтін санды қайтарады
{return k;}
int &f2(int &k)          // функция k айнымалысына сілтемені қайтарады
{return k;}
```

Екі функцияда k енгізілетін параметрінің сандық мәніне сәйкес келетін мәнді қайтарады. Екінші жағдай үшін шақырудың келесі түрі мүмкін:

```
int main(void)
{int n=3;
f2(n)=7;          // n  жетіге тең болады
return 0;
            }
```

Функцияға нұсқауышқа ұқсас функцияға сілтеме форматы анықталады: функция_типі (&сілтеме_аты) (функция_параметрлерінің_ерекшеліктері) =атаулаушы өрнек; Мысалы:

```
int func(float, int);                    // функция прототипі
int (&ref)  (float, int)=func;        // сілтеменің анықталуы
```

Функция атының жақшасыз (және параметрлерсіз) қолданылуы функция адресі ретінде қабылданады.

Функцияға сілтеме функцияның негізгі атының барлық құқықтарына ие, яғни оның синонимі (псевдонимі) болып табылады. Функцияға сілтеме (нұсқауышқа сілтеме) мәнін өзгерту мүмкін емес, сондықтан функцияға нұсқауыштар сілтемеге қарағанда үлкен ауқымға ие. Келесі программа функцияны шақыруды негізгі аты бойынша, нұсқауышпен және сілтеме бойынша орындауды суреттейді:

```
#include <iostream.h>
void func(char lit)
{cout<<endl<<lit;}
int main (void)
{void (*pf)(char);              // pf - функцияға нұсқауыш
void(&ref)(char)=func;       // ref - функцияға сілтеме
func('A');                        // атымен шақыру
pf=func;                         // нұсқауышқа функция адресі меншіктеледі
(*pf)('B');                     // нұсқауыш көмегімен адресі бойынша шақыру
ref('C');                       // функцияны сілтеме арқылы шақыру
return 0;
}
```

Функциямен сілтемені қайтару бірнеше функцияларға бірнеше қайтара әрекеттесуді ұйымдастыруға мүмкіндік береді. Осы әрекеттесу нәтижесінде бір объектіні бірнеше қайтара әр түрлі ережемен өзгертуге болады.

Бақылау сұрақтары:

1) Сілтеме дегеніміз не?
2) Сілтеме параметрлерінің қандай артықшылығы мен кемшіліктері бар?
3) Объектіні сілтеме арқылы функцияға бергенде оның көшірмесі құрыла ма?
4) Сілтеме параметрлерін қолданған кезде адрес аргументі қалай жіберіледі?
5) Тәуелсіз сілтеме деген не?

3 Дәріс №3. Класс абстрактылы тип ретінде

Дәрістің мақсаты: класс ұғымын толық түсіндіру.

Дәрістің мазмұны: «Клиент-сервер» технологиясы бойынша класс әдістерін анықтау. Конструктор мағынасы. Деструктор мағынасы.

Деректер абстракциясына негізделетін программаларды жүзеге асыру және құрастыру, жобалау принциптерінің жиынтығы, шешілмелі есептердің ерекшеліктері мен түсініктерін анағұрлым толық көрсететін деректердің жаңа типтерін құруды қарастырады. C++ тілінде программалаушының өзіндік деректер типтерін енгізуге және олармен кластар көмегімен орындалатын операцияларды анықтауға мүмкіндігі болады.

Класс - бұл бар типтер негізіндегі туынды құрылымдық (структурированный) тип. Класты келесідей ең оңай әдіс арқылы анықтауға болады:

Класс_кілті класс_аты {компоненттер тізімі};

мұнда класс_кілті - class, struct, union қызметші сөздерінің бірі;

класс_аты - еркін таңдалатын идентификатор;

компоненттер тізімі - класқа қатысты функциялар мен типтелген деректердің сипаттаушылары мен анықтамалары. Компоненттер обьект күйін анықтайды және класс объектісінің тәртібін шартқа негіздейді. Класс компоненттері деректер, функциялар, кластар, санауыштар, биттік өрістер, достық функциялар мен типтер атаулары бола алады. Мысалы:

```
class Classid
{ int k;                // к, х, у жабық мүшелер - үнсіздік бойынша деректер
double x,y;
public: char lit, ch;     //  lit,ch ашық мүшелер - деректер
int f3(int,int);          //  кластың ашық әдістері
int Get x (void)          //  кластың орнына қою әдісі
{ return x; }
private:
void f1 (void);         // жабық мүшелер - класс әдістері
int f2 (int);
};
```

Қатынау спецификаторлары public: және private: қатынау спецификаторлары класс мүшелеріне өтуге мүмкін беретін режимді анықтайды: private - класс мүшелерін жабыққа айналдырады, ал public - ашық. Кластың жабық мүшелеріне тек берілген класс мүшелерінің қатынауға рұқсаты бар, сонымен қатар функциялардың – класс достары. Кластың ашық мүшелеріне кез келген функциялардың қатынауға мүмкіншіліктері бар. Олар бар болып табылатын бағдарлама мен класс объектілерінің байланысы үшін арналған.

Класты жобалап отырып, оның қай мүшесін ашық, ал қайсысын жабық етуді мұқият ойластырған жөн. Көптеген жағдайда класс анықтамасы блокта локализацияланбаған және класс атауының іс-әрекет облысы бүкіл файл болып табылады.

Жай кластық әдістерін анықтау класс анықтамасының ішінде болуы мүмкін және мұндай жағдайда олар автоматты түрде орын ауыстырғандар

болып табылады. Орын ауыстыратын функцияларды егер функция қарапайым және қысқа болса, қолданған жөн, мысалы, Getx ( ) функциясы.

Әдетте, қауіпсіздікті жоғарылату мақсатымен кластың мүше-берілгендерін жалпыға бірдей рұқсат етілмейтін етеді және еркін бағдарлама ортасынан олармен тікелей байланысуға болмайды. Бірақ оған қарамастан олармен кластың барлығына рұқсат етілген әдістерін қолданып ерікті бағдарламалық ортадан да жұмыс істеуге болады (public:). Жоғарыда қарастырылған мысал үшін келесі әдіс қолданылады:

```
int Getx (void)
{ return x; }
```

жабық мүше мәнін берілген x қайтарады.

Арнайы түрде мұндай әдістердің «жүдеу» денелерін тиімділікті жоғарылату үшін орын ауыстырушы етіп класс анықтамасының ішінде енгізеді. Класс анықтамасынан тыс орналастыратын әдістерді анықтаған жағдайда әдістер тақырыбына inline спецификаторы қосылады.

«Клиент-сервер» технологиясы бойынша класс әдістерін анықтау класс анықтамасынан тыс орналастырған жөн. Көру облысының «::» рұқсат ету операторы арқылы компиляторға берілген анықталатын әдіс қандай класқа қатысты екендігін хабарлайды, мысалы:

```
int Classid:: f2 (int x)
{ әдіс денесі }
```

Класс анықтамасы берілген класс объектілерін құрмайды. Объектілер оларды анықтау жолымен ғана құрады, мысалы:

```
Classid obj1, obj2, obj Array [10];
```

Класс объектісінің өлшемі кластың статикалық емес мүше берілгендерінің өлшемдер қосындысымен жадыда анықталады. Класс әдістері класс объектісі үшін бөлінген жады облысының орнын алмайды.

Бірнеше файлдардан тұратын бағдарламалық жобалармен жұмыс жасағанда класс анықтамасы берілген класс объектілері қолданатын немесе оның әдістері анықталатан файлдарда болуға міндетті. Сондықтан класс анықтамасын арнайы түрде ол қажет болып табылатын файлдарда #include дерективасы көмегімен қосылатын тақырыптық файлға (хедер-файл) орналастырылады. Егер класс әдісінің анықтамасы анықталудан тыс орналасса, онда ол міндетті түрде өзі қолданылатын файлдарда анықталған болуы керек. Мұндай әдістің анықтамасы класс анықтамасымен бірге тақырыптық файлда болуы керек.

Белгілі бір класс объектісінің ашық мүшелеріне қатынауға рұқсат алуды тура «.» және жанама « → » таңдау операторлары көмегімен жүзеге асырылады.

C++ тілінде объектілерді жою мен инициализациялау жұмыстарын орындайтын, обьектілердің жойылуы мен құрылуы кезінде автоматты түрде шақырылатын, кластың арнайы әдістері сәйкесінше конструкторлар мен деструкторлар деп аталады. Класта бірнеше конструкторларды құруға болады. Әрбір конструктордың өзінің басқалардан айырмашылығы бар параметрлері болады. Параметрлері жоқ конструктор үнсіздік конструкторы. Деструктор әрқашан жалғыз және параметрлері болмайды.

Тек ашық мүшелері бар және конструкторы жоқ класс объектісі мәндер тізімі арқылы жай құрылымдық айнымалы сияқты инициализациялануы мүмкін.

Егер класс конструкторларының біреуі де ашық функция – класс мүшесі болып табылмаса, онда мұндай кластың мүшелері құрыла алмайды. Мұндай кластар басқа кластар (мұрагерлік) үшін базалық кластар (родительские) болып табылады.

Класс объектісін құрған кезде ол үшін сәйкес келетін конструктор автоматты түрде шақырылады. Объекттің деректер мүшесі (объект айнымалыларының) инициализациясы конструктордың орындалуы сияқты инициализациясы кезінде де орындала алады.

Конструктор параметрі өз класы бола алмайды, бірақ көшіру конструкторы сияқты оған жасалған сілтеме болуы мүмкін.

Кластың деректер компонентінің жалғыз данасы болуы үшін және әрбір жаңа кластың объектісінің құрылуы кезінде тираждалмауы үшін ол класты статикалық сияқты анықтау керек, яғни Static атрибуты болуы керек.

Кластың статикалық компоненттік функциялары нақты обьект атауы жоқ кластың Static берілгендеріне қатынасуды қамтамасыз етеді. Кластың Static функциялары функциялардың жай (статикалық емес) компоненттік барлық негізгі ерекшеліктерін сақтайды. Класқа қатысты функция нақты объекттің деректерін өңдеу үшін шақырған кезде бұл функцияға автоматты және айқын емес функция шақырылған объект көрсеткіші беріледі. Бұл нұсқауыштың бекітілген *this* аты бар және программалаушы үшін кластың әрбір функциясы келесі түрде анықталған:

класс_аты*const this=өңделетін_обьекттің_адресі;

this нұсқауышын қолдану тиімділігі көрінетін жағдайлар:

- берілген класс компонентінің аты класс функциясының формальды параметрінің атымен сәйкес келгенде;
- класс функциясының денесінде айқын түрде бұл функция шақырылған объект адресін көрсету қажет болғанда;

- функция параметрі ретінде бұл кластың функция атын жазған кезде объекттің нақты атына жол берілмегенде (керек объектіге мұндай функция сілтемесін немесе нұсқауышын берудің орнына).

Бақылау сұрақтары:

1) Белгілі бір кластың екі объектісін анықтағанда, деректер мүшелері өздерінің әртүрлі (бірдей) мәндеріне ие бола ала ма?
2) Кластың қандай деректер мүшелерін қандай жолмен иниализациялау керек?
3) this нұсқаушысы нені білдіреді?

4 Дәріс №4. Кластар достары. Кластарға арналған стандартты операцияларды қайта жүктеу

Дәрістің мақсаты: класс достары және достық функциясы ұғымдарын беру.

Дәрістің мазмұны: достық функциясының ерекшеліктері. Көшіріп алу конструкторы. Беттік және тереңдетілген көшіріп алу. Кластарға арналған операциялардың қайта жүктелуі.

Класс мүшесі болмауына қарамастан кейбір жағдайларда функция кластың жабық мүшелеріне тиесілікке (доступ) ие болғаны жөн. Егер функция класс досы немесе мүшесі болып табылатын класты friend қызметші сөзі көмегімен жарияласа, осы әрекетті C++ ортасы орындауға мүмкіндік береді,.

Достық функциялары мен кластарды қолданудың бірқатар қызықты ерекшеліктерін көрсететін мысалды қарастырайық:

```
#include <iostream.h>
class Class1;                          // класты алдын ала жариялау
class Class2;
{ public : void ff (Class1 &ref);
};
class Class1
{ private : int x;              // достық  функциялары мен кластарға арналған
//жол беру спецификаторының мәні жоқ
friend int  main (void);           // main – достық функциясы бола алады
friend void f (Class1 &ref);
friend class Class3;
friend void Class2 :: ff (Class1 &ref);
};      //Class1 анықтамасының соңы
class Class3                   // Class3 – Class1 класына дос
{ public : void func (Class1 &ref)
{ ref.x=9;
```

```
return;
}
};
void Class2 :: ff (Class1 &ref)
{ ref.x=11;
return;
}
//f – Class1 класына дос қарапайым сыртқы функция
void f (Class1 &ref)
{ ref.x=7;
return;
}
int main (void)
{Class1 obj1;
obj1.x=5;
cout << “obj.1x=” << obj1.x << endl;  //x=5
f(obj1);
cout<<“obj1.x=” << obj1.x <<endl;   //x=7
Class3 obj3;
obj3.func(obj1)
cout<< “obj1.x=” << obj1.x <<endl;   //x=9
Class2 obj2;
Obj2.ff(obj1);
cout<< “obj1.x=” <<obj1.x<<endl;   //x=11
 return 0;
}    // main соңы
```

Достық функциясының ерекшеліктері:

- достық функция класс объектісіне автоматты түрде this нұсқауышын ала алмайды, сондықтан оның шақыруында міндетті түрде аргумент ретінде оған жұмыс істеуге қажет класс объектісін көрсету керек;
- консольды қосымшаның басты функциясы класқа дос болуы мүмкін;
- қажетсіз достық функциялары мен кластарын қолдануға ұмтылмау керек, себебі бұл инкапсуляция концепциясына кері болып табылады.

Кластарға арналған операциялардың қайта жүктелуі (перезагрузка).

C++ тілінің көптеген @ операциялары қолданушы типтерінің объектілерімен жұмыс істеу үшін қайта жүктелуі мүмкін. Мұндай мүмкіндіктер объект-кластар үшін де болады. @ операциясының қайта жүктелуі operator @ фунция-операциясы көмегімен жүзеге асырылады, оны класс мүшесі немесе достық функциясына айналдыру (қажет емес) немесе қарапайым (глобальды) функцияға айналдыруға болады.

Соңғы екі жағдайда функция-оператор ең болмаса бір сілтеме немесе нұсқауыш, класс типі бар аргументті қабылдау керек. Сыртқы функция -

операциясы үшін сәйкес класс мүшелерінің тиесілігін (доступность) ескерген жөн.

Функция - оператор атауы operator қызметші сөзінен, және одан кейін орналасқан қайта анықталатын операция белгісінен тұрады.

Операцияларды қайта жүктеу типті беру, функцияны шақыру ережелері ISO/IES14882 халықаралық стандартында берілген.

Көшіріп алу конструкторы. Беттік және тереңдетілген көшіріп алу

Класс құрамына арнаулы түр конструкторы – көшіріп алу конструкторы жатады. Көшіріп алу конструкторының жалғыз параметрі ретінде, осы класс объектісіне сілтемені қолданады:

```
Classid : Classid (const Classid & obj)
{ конструктор денесі }
```

Көшіріп алу конструкторы жаңа обьект класс типімен бар обьектіні көшіріп алу жолымен құрылған кезде шақырылады:

- сол типтің басқа бар объектімен инициализациясы және класс типі бар жаңа объект анықтаған кезде;
- мән бойынша класс типі бар параметр - объект класс әдісіне берілген кезде;
- return операторы арқылы класс типі бар объект мәнін класс әдісінен қайтарылған кезде.

Егер программалаушы бір де бір көшіріп алу конструкторын құрмаса, онда компилятор автоматты түрде қалыпты көшіріп алу конструкторын құрады. Мұндай конструктор кластың мәліметтер өрістерінің элемент бойынша көшірілуін орындайды (беттік көшіріп алу). Егер өрістердің ең болмаса біреуі динамикалық жадының белгілі бір облысына нұсқауыш болып табылмаса, онда қалыпты конструкторды қолдану программаның дұрыс емес, яғни бұрыс жұмысына әкеледі. Осы жағдайда қалыпты көшіріп алу конструкторының орнына өзінің көшіріп алу конструкторын жобалап алған жөн:

- беттік көшіруді орындайтын конструктор (нұсқауыштың объектіге көшірілуі орындалады, ал объект көшірмесі динамикалық жадыда құрылады);
- тереңдетілген көшіріп алуы бар конструктор (динамикалық жадыда объект көшірмесі құрылады).

Қорытындысында мынаны қайталайық, егер класта динамикалық жадымен жұмыс орындалмаса және конструктор қажет болмаса, онда қалыпты көшіріп алу конструкторын қолдану көмегімен көшіріп алу конструкторын анықтау керек емес. Осындай кластарды қолданатын бағдарламалар дұрыс жұмыс істейді.

Мүмкіндігінше мән бойынша, класс типі бар параметрді кластың әдісіне және класс типі бар объекттің мәнін емес, сілтемесін оған класс әдісіне

қайтарудан аулақ болған жөн. Бұл жағдайда көшіріп алу конструкторы қажет емес, бұл қосымшаның шапшаңдығының жоғарылауына әкеледі.

Көрсетілген мақсатқа жету үшін келесі ережелерді қолданған жеткілікті:

- мән бойынша класс типі бар параметрді, класс әдісіне берудің орнына бұл параметрді сілтемемен берген жөн, ал сәйкес аргументтің модификациясын тоқтату үшін функция параметрін const модификаторымен қамтамасыз ету керек;
- класс типі бар объектілер үшін арифметикалық операцияларды қайта жүктегенде қайта жүктеу әдісін объектке қайтаратындай етіп жобалау керек (оны әрқашанда орындауға болады).

Бақылау сұрақтары:

1) C++ тілінің операцияларын қайта жүктеу операциялары қандай?
2) Көшіріп алу конструкторы қашан шақырылады?
3) Бірігіп көшу операторларының префиксті және постфиксті қайта жүктеуін қалай айыруға болады?
4) short int типті операндалар үшін қосынды операциясын қайта жүктеуге бола ма?

5 Дәріс №5. Үлгілер (шаблоны)

Дәрістің мақсаты: көп қолданылатын және қарсылыққа тұрақты программаларды құру мүмкіндіктері.

Дәрістің мазмұны: функция-үлгілер. Туыстық функциялар. Туыстық кластар.

Жоғары деңгейлі C++ тілінің келесідей екі негізгі сипаттамасы болып мыналар табылады: үлгілер (templates) (шаблоны) және ерекше жағдайлардың өңделуі (exception handling). Бұл сипаттамалар қазіргі уақыттағы барлық компиляторлардан қолдау табады және программалауда анағұрлым қызықты екі мақсатқа жетуге: көп рет қолданылатын және қарсылыққа төзімді программаларды құруға мүмкіндік береді.

Үлгілер көмегімен туыстық функцияларды (general functions) және туыстық кластарды (general classes) құруға болады. Туыстық функцияда немесе класта, класс немесе функция жұмыс істейтін мәліметтер типі параметр ретінде беріледі. Бұл бір класты немесе функцияны функция немесе кластың жаңа нұсқауын қолданбай-ақ деректердің әртүрлі бірнеше типтерін қолдануға мүмкіндік береді. Сөйтіп, үлгілер көп рет қолданылатын программаларды құруға мүмкіндік береді.

Көптеген алгоритмдер логикалық жағынан бірдей деректер типінен тәуелсіз, мысалы, сұрыптау алгоритмдері. Туыстық функцияны құрудың арқасында деректер типінен тәуелсіз алгоритм мәнісін анықтауға болады.

Туыстық функция template шешуші сөзі арқылы құрылады. Функция-үлгі анықтамасының типтік формасы:

```
template  < class T >  мәнді_қайтару
        функция_аты (параметрлер_тізімі)
        {функция денесі}
```

Мұндағы T - берілгендер типінің жалған аты, оны компилятор функцияның нақты нұсқасын құрғанда берілгендердің шын типінің атымен автоматты түрде ауыстырады.

Мағынасы ұқсас функциялар бар болғанда түрлі деректер типтері үшін екі шаманың ең кішісін табу үшін MyMin атты жалғыз үлгілік функцияны анықтауға болады:

```
template < class T > T MyMin (T x, T y)
{ if (x<=y) return x; else return y;}
```

Ол потенциалды түрде белгісіз T типімен де жұмыс істей алады.

Функциялар үлгілерінің келесі тақырыптары синтаксистік жағынан дұрыс болып табылады:

```
template< class T > T Fun1(T x, T y, int z)
template<class T > double Fun2 (Tx, Ty)
template<class T, class W > W Fun3(Tx,Wy,bool z).
```

1) Үлгінің формальды параметрлерінің анықтауыштары (T,W) ең болмаса формальды параметрлер тізімінде бір рет болса да болу керек.

2) Үлгілік функцияның денесі ішінде үлгінің формальды параметрлері локальды айнымалыларды анықтау операторларында, типті беру операторларында және т. б. өзге түрлі деректердің нақты типтерімен бірдей қолданылады, яғни жұмыс істеуде деректер типтерінің толық құқылы атаулары сияқты.

3) Кәдімгі функциялар сияқты үлгілік функцияларға да прототиптерді жазуға болады, сонымен қатар оларға inline, static спецификаторларын да жазуға болады (бұл спецификаторлар үлгінің формальды параметрлер тізімінен кейін және функциямен қайтарылатын мән типіне дейін орналасу керек).

4) Функция үлгілерін анықтау мәтіні шақыру орындалатын файлдарда немесе тақырыптық файлдарда (h-файлдарда) орналастырылу керек.

5) Үлгілік функция анықталу алдында template кілттік сөзімен нұсқаулық орналасу керек.

6) Функция үлгілерінің механизмін дәлелсіз қолданудың ешқандай қажеті жоқ! Біріншіден, функция үлгілерінің деректердің абстрактты типтерімен байланысының пайдасы бар. Үлгілер механизмі глобальды

функциялар үшін емес кластар мен олардың әдістерінің үлгілері үшін ерекше пайдалы!

Туыстық кластар.

Туыстық функцияларын толықтыру үшін туыстық кластарды да анықтауға болады. Осы кластың объектілерін құру кезінде барлық қажетті алгоритмдер анықталып, өңделетін деректердің нақты типтері параметрлері ретінде кейінірек беріледі.

Туыстық кластар, класс құрамында жұмыстың жалпы логикасын қамтығанда пайдалы. Мысалы, бүтіндер кезегін жүзеге асыратын алгоритм символдар кезегімен де жұмыс істейді. Сонымен пошталық адрестердің байланысқан тізімін жүзеге асыратын механизм автомобильдерге арналған қосалқы бөлшектердің байланысқан тізімін де қолдайтын болады. Туыстық класс арқылы түрлі берілгендер типтеріне арналған байланысқан тізімдер кезегін жүзеге асыратын класты құруға мүмкіндік береді. Компилятор берілген объектіні құруда, объектінің тип негізінде дұрыс типін автоматты басқарады.

Класс үлгісінің сипатталу синтаксисі келесі түрде болады:

- template < үлгі_параметрлерін_сипаттау > класс_анықтамасы
- Параметрлер ретінде стандарттыға ұқсас қолданушымен анықталатын типтер үлгілер мен типтелген конструктор қолданылады.

Бір байланысты тізімді жүзеге асыратын жай туыстық класты құрайық:

```
#include< iostream.h >
template< class T > class List
{ T data;
List*next;
public:
List(T d);                          //  класс конструкторы
Void add(List*node)
{node    next=this; next=0;}
List*getnext() {return next;}
T getdata(){return data;}
}
template< classT >List<T>::List(Td)
{data=d; next=0;}
int main(void)
{List< char > start('a');           //start-обьект
List< char >*p,*last;
//тізімді құру
last=&start;
for(int i=1; i<26;i++)
{p=newList<char> ('a'+i);
p->add(last); last=p;
```

```
}
//тізімді шығару
p=&start;
while(p)
{cout<<p->getdata();
p=p->getnext();
}
return 0;
}
```

Көрініп тұрғандай, туыстық класты жариялау туыстық функция жариялануына ұқсас. Тізімде сақталынатын берілгендер типі класты жариялағанда туыстыққа айналады, бірақ ол деректердің нақты типін беретін объект жарияланбай көрінбейді. Объектті құрғанда көрсетілген мәліметтер типін өзгерту жолымен тізімде сақталған мәліметтер типін өзгертуге болады.

Кейбір қорытындыларды келтірейік және үлгілерді сипаттаудың негізгі ережелерін атап өтейік:

1) Класс үлгісінің ішінде тип параметрі тип спецификациясы қолданылатын кез келген жерде қолданылады.

2) Үлгі параметрінің әрекет ету облысы – үлгі параметрінің сипаттау нүктесінен класс үлгісінің соңына дейін.

3) Класс үлгісінің әдістері автоматты түрде функция үлгілеріне айналады. Егер әдіс класс үлгісінен тыс сипатталса, онда әдіс тақырыбы келесі құрылымға ие болу керек:

```
template<үлгі_параметрін_сипаттау>
типті_қайтару класс_аты  < үлгі_параметрлері>::
```

Үлгі параметрлерін сипаттау әдіс тақырыбын сандық және позициялық түрінде сақтай отырып, класс үлгісіне сәйкеес келу керек.

4) Локальды кластар үлгілерді өз элементтері ретінде қолдана алмайды.

5) Класс үлгілерінің әдістері виртуальды бола алмайды.

6) Класс үлгілеріне статикалық элементтер, достық функциялар мен кластар ене алады.

7) Класс үлгісінің ішінде достық функциялардың үлгілерін анықтауға болмайды.

8) Класс үлгілері үлгіліктерден және жай кластардан туынды бола алады, сонымен қатар үлгілік және жай кластар үшін базалық болып табыла алады.

Бақылау сұрақтары:

1) Класс үлгілерін анықтау мүмкіндігі не?

2) Үлгілік функцияға арналған машиналық кодтың шын генерациясы қалай және қашан жүргізіледі?

3) Үлгілік функцияны шақырғанда нақтыландыру қалай орындалады?

4) Белгілі бір функционалдық үлгінің қай функциясы маманданған деп аталады?

5) Кезекті жүзеге асыратын туыстық класты құрыңыз және көрсетіңіз.

6 Дәріс №6. Мұрагерлік

Дәрістің мақсаты: кластар мұрагерлігін түсіндіру.

Дәрістің мазмұны: кластың қорғалған мүшелері. Конструкторлар, деструкторлар және мұрагерлік.

Мұрагерлік – C++ тіліндегі объектіге бағытталған программалаудың басты механизмдерінің бірі. Оның көмегімен жалпыдан жекеге ауыса отырып өте күрделі кластарды құруға болады, сонымен қатар қорытынды кластардан айрықшаланатын жаңа кластарды өсіруге болады.

Жаңа класты жобалай отырып, алдын-ала оның объектілері қандай анағұрлым жалпы ерекшеліктерге ие болу керектігін анықтау және ұқсас дайын кластың бар жоқтығын тексеру қажет. Басқаша айтқанда, басында қайта өндіретін класс жоспарын «ірі көріністермен» құрған жөн, ал кейін бір уақытта жаңа қасиеттерді ала отырып, тәртіп пен қасиеттерді (яғни мүше деректер мен класс әдістері) мұраға алатын жаңа кластарды құрылған класс негізінде құра отырып, рет-ретімен бөлшектенуге көшеміз.

Түрлі класс объектілері мен кластардың өздері алдын-ала құрастырған кластар иерархиясына сай келетін объектілер иерархиясы құрылған кезде мұрагерлік қатынасында бола алады.

Бір класс екінші класс мұрагері болғанда жазбаның келесі негізгі пішіні қолданылады:

{ class_ класс_ аты_ as базалық_ класс_ аты _ туынды
{ туынды класының анықтамасы

Мұндағы as – рұқсат ету спецификаторы (access specifier) базалық класс элементтері (base class) туынды класқа (derived class) қалай мұрагер болатынын анықтайды.

Егер as public болса, онда барлық ашық мүшелері туындысы да ашық болып қалады. Егер as private шешуші сөзі болса, онда базалық кластың барлық мүшелері туынды класта жабыққа айналады. Екі жағдайда да базалық кластың барлық private мүшелері туынды класта қалай мұрагер болуына тәуелсіз, жабық және рұқсатсыз болып қалады.

Егер as private болса, онда базалық кластың ашық мүшелері туынды класта жабық болуына қарамастан олар туынды кластың функция мүшелері үшін рұқсат етілген болып қалатынын ескерген жөн.

Техникалық жағынан рұқсат ету спецификаторы міндетті емес. Егер ол көрсетілмеген туынды класс class шешуші сөзімен анықталса, онда базалық класс struct қызметші сөзімен үнсіздік бойынша ашық түрде мұра болады.

Түсініктілік үшін рұқсат ету спецификаторын нақты түрде беру дұрыс.

Public спецификаторымен мұрагерлік мысалын көрсетейік:

```
class Base
{ int x;
public:
void setx (int n) {x=h;}
void showx( ) { cout « x «endl ;}
};
 class Derived: public Base
{ int y;
public:
void setx (int n) {x=h;}
void showx( ) { cout « x «endl ;}
};
 int main (void)
{ Derived obj;
   obj.setx (10);
   obj.sety (20);
   obj showx( );
   obj showy( );
return 0;
}
```

Base класы ашық болып, мұраға қалдырылатындықтан оның ашық функциялары туынды класқа ашық болып қалады және сондықтан да программаның кез келген бөлігінен рұқсат етілген болады. Бұл функцияларды main ( ) функциясынан сәйкесінше дұрыс шақыру керек.

Базалық кластың мүшелері рұқсат етілмеген болады, сондықтан туынды класс ішінде x айнымалысына тікелей рұқсат алуға талаптану әрекеті дұрыс емес. Ал базалық кластың ашық функциясы арқылы бұл мүмкін болады.

Егер класс сияқты жабық мұраға қалса, онда базалық кластың барлық мүше деректері мен туынды класта жабық болады және одан тыс қана рұқсат етілген болады. (Derived типті берілген объекттер үшін олар жабық болады).

Private спецификаторын қолдана отырып мұра ету кезінде базалық кластың ашық мүшелері туынды класта жабық болады.

```
# include < iostream.h >
```

```
class Base
{ int x;
public:
void setx (int n) {x=h;}
void showx( ) { cout « x «endl ;}
};
class Derived: public Base
{ int y;
public:
void setxy(int n, int m) {set(n); y=m;}
void showxy( ) {showx( ) ; cout « y «endl ;}
};
 int main (void)
{ Derived obj;
obj.setx (10);
obj.sety (20);
obj showx( );
obj showy( );
return 0;
}
```

Бұл жағдайда show x ( ) және set x ( ) функциялары туынды класс ішінде рұқсат етілген болады, бұл өте дұрыс, себебі олар бұл кластың жабық мүшелері болып табылады.

Кластың қорғалған мүшелері.

Қажет болғанда, базалық класс мүшелері жабық болып қалуы үшін, туынды класс қолжетімді болуы мүмкін. Осы идеяны іске асыруда C++ те protected: спецификаторы қарастырылған. protected: спецификаторының қолжетімділігі private: спецификаторынының жалғыз ерекшелігі - базалық кластың қорғалған мүшелері осы базалық кластың барлық туынды кластар мүшелеріне қолжетімділігіне. Базалықтан тыс немесе туынды кластардың қорғалған мүшелері қолжетімсіз.

Protected спецификаторы кез келген жерінде, әдетте кластың private мүшелерінен кейін public – класс мүшелерінің алдында орналасуы мүмкін.

Базалық класс туынды кластың ашық (public) сияқты мұрасы болғанда базалық кластың жабық мүшесі туынды кластың қорғалған мүшесіне айналады. Базалық класс жабық (private) сияқты мұра болғанда, онда базалық кластың қорғалған мүшесі туынды кластың жабық мүшесіне айналады.

Сөйтіп мынаны ескерген жөн, туынды клас объектілері үшін мыналар рұқсат етілген: базалық кластың объектілері үшін ашық мүше деректері мен барлық ашық әдістері рұқсат етілген:

- класс мүшелері болып табылмайтын функциялар;
- глобальды айнымалылар.

Туынды класс ішінде базалық кластың базалық қорғалған мүшелері рұқсат етілген.

Базалық класс тізімінде туынды класты жариялауға public модификациялары қолданылады: protected: private: олардың көмегімен базалық кластың мүшелері рұқсат етілмеуі мүмкін.

Бұл C++ тілінің маңызды функциясы болып, объекттіге бағытталған программалаудың жақсы стилінің белгісі табылады.

Базалық кластың қорғалған мүшелері туынды кластың қорғалған мүшелері болады және main функциясы ішінде олар рұқсат етілген болады.

Protected рұқсат ету спецификаторын құрылыммен бірге де қолдануға болады.

Конструкторлар, деструкторлар және мұрагерлік.

Егер базалық және туынды кластардың конструкторлары мен деструкторлар бар болса, онда конструкторлар мұрагерлік тәртіппен, ал деструктор кері тәртіпте орындалады.

Базалық класта инициализация бірінші болып орындалуы, ал туынды класс деструкторы объект жойылғанға дейін шақырылуы керек.

Базалық және туынды кластардың барлық қажет аргументтері туынды класс объектілері құрылғанда туынды класс конструкторына беріледі. Кейін туынды класс конструкторын жариялаудың кеңейтілген формасын қолдана отырып, сәйкес аргументтер әрі қарай базалық класқа беріледі. Аргументтерді туындыдан базалық класқа берілу синтаксисі төменде көрсетілген:

туынды_класс_конструкторы (аргументер_тізімі): базалық_класс (аргументтер_ тізімі)

Базалық және туынды кластар үшін бірдей элементтерді қолдану рұқсат етілген. Сонымен туынды класс үшін базалық аргументтерді елемеуге және тікелей оларды базалық класқа беру рұқсат етілген.

Бақылау сұрақтары:

1) Базалық және туынды кластар әдістерінің бірдей аттары болуы мүмкін бе?

2) Туынды кластың болашақ ұрпағына базалық кластың әдістері мен деректеріне мұра бола ала ма?

3) Three класы Two класымен түзілген болсын, ал Two класы One класымен түзіледі. Бұл әдістің қай түрін Three класы алады?

4) Базалық класта public түрінде сипатталып өткен әдісті туынды класта private: түрде сипаттауға бола ма?

5) Protected қызметші сөзі қандай мақсатта қолданылады?

7 Дәріс №7. Кластар иерархиясы

Дәрістің мақсаты: кластардың иерархиялық құрылымын түсіндіру.

Дәрістің мазмұны: базалық класс. Туынды класс. Виртуалды базалық кластар.

Туынды класс бір базалық кластан артық класқа мұра ете алатын екі әдіс бар. Біріншіден туынды класс көп деңгейлі кластар иерархиясын құра отырып, базалық ретінде басқа туынды класс үшін қолдануға болады. Екіншіден, туынды класс бір базалық кластан артық класты тікелей мұра ете алады. Класс туындыға базалық класс ретінде қолданылса, өз кезегінде басқа туынды класс үшін базалық болса, онда барлық үш кластың конструкторлары мұрагерлік ретімен шақырылады. Деструкторлар кері тәртіппен шақырылады.

Егер туынды класс бірнеше базалық кластың мұрасына тікелей ие болса, онда мынадай кеңейтілу қолданылады:

```
class туынды _класс _аты:as базалық_ класс_аты1;
as базалық_класс_аты N
{ туынды класс денесі }
```

Мұндай жағдайда конструкторлар туынды кластың хабарламасында берілген тәртіп бойынша солдан оңға қарай орындалады. Егер осындай мұраға ие болу әдісі кезінде базалық кластардың конструкторларына аргументтерді беру қажет болса, туынды класс оларды туынды класс конструкторларының хабарламасының кеңейтілген формасын қолдана отырып береді:

```
туынды_класс_конструкторы (аргументтер_тізімі);
базалық_класс1_аты (аргументтер_тізімі);
базалық_класс 2_аты (аргументтер_ тізімі);
....., базалық_класс N_аты (аргументтер_тізімі)
{туынды класс конструкторының денесі }
```

Егер туынды класс кластар иерархиясының мұрасына ие болса, әрбір туынды класс тізбекті алдыңғы базалық класқа барлық қажет аргументтерді береді. Бұл ережені орындамау программаны компиляциялау кезінде қателік туғызады.

Қызметші сөз class көмегімен анықталған класс базалық сияқты туынды бола алады. Мұндай жағдайда үнсіздік бойынша класс мүшелері жабық болып табылады.

Қызметші сөз Struct көмегімен анықталған кластар базалық сияқты туынды да бола алады, және үнсіздік бойынша класс мүшелері ашық болып табылады.

Қызметші сөз union бар класс белгілі бір класқа қатысты базалық та, туынды да бола алмайды. Туынды класс кезіндегі жағдайларды түсіндіретін бағдарламаны қарастырайық:

- басқадан туынды класс мұрасына ие болады;
- екі базалық кластың мұрасына тікелей ие болады;

- бірнеше базалық кластың мұрасына тікелей ие болады.

```
# include <iostream.h>
class Base1
{ int a;
public: Base1(int x) {a=x; }
 int geta() { return a; }
} ;
class Der1:public Base1
{ int b;
public: Der1(int x,int y):Base1(y)
{ b=x;}
int get b() {return b;}
};
class Der2 :public Der1
{ int c;
public:Der2 (int x,int y,int z):Der1(y,z)
{ c=x;}
void show()
{ cout«geta ()« ′ ′«getb()«′ ′ «getc()«endl;}
};
int main (void)
{ Der2 obj(1,2,3);
obj.show( ) ;
cout « obj.geta ( ) «′ ′« obj.getb( ) « endl ;
return 0;}
```

Базалық кластың ашық мүшелерінің мұрасына ие болғанда олар туынды кластың ашық мүшелеріне айналады. Сондықтан, егер geta Base1 класына мұрагерленсе, онда get a ( ) функциясы Der1 класының ашық мүшесіне және кейін Der2 класының ашық мүшесіне айналады.

Туынды класс тікелей екі базалық класқа мұрагерленетін алдыңғы бағдарламаның қайта өндірілген нұсқасын қарастырайық:

```
# include <iostream.h>
class Base1
{ int a;
public: Base1(int x) {a=x; }
int geta () { return a; }
} ;
 class Base2
{ int b;
public: Base2 (int x) {b=x;}
int getb () {return b;}
```

```
};
class Der :public Base1, public Base2
{int c;
public: Der(int x,int y,int z):Base1(z),Base2(y)
{ c=x;}
void show ()
{ cout «geta () « ′ ′«getb() «′ ′ «getc()«endl;}
};
int main (void)
{Der obj (1,2,3);
obj.show ( ) ;
return 0;}
```

Келесі бағдарламада туынды класс тікелей бірнеше базалық класқа мұрагерленетін кезде конструкторлар мен деструкторлар шақырылатын тәртіпті көрсетейік:

```
# include <iostream.h>
class Base1
{ int a;
public: Base1() { cout «  Base1 класс конструкторының жұмысы\ n"; }
~ Base1() { cout « " Base1 класс деструкторының  жұмысы\ n"; }
int x) {a=x; }
int geta () { return a; }
} ;
class Base2
{ int b;
public: Base2 () { cout «" Base2 класс конструкторының жұмысы\ n"; }
~ Base 2(){ cout «" Base2 класс деструкторының жұмысы\ n"; }
};
class Der :public Base1, public Base2
public:Der() { cout«" Der класс конструкторының жұмысы\ n"; }
~Der () { cout «" Der класс деструкторының жұмысы\ n"; }
};
int main (void)
{ Der obj;
return 0;}
```

Бірнеше базалық кластар тікелей мұрагерленсе, конструкторлар тізіммен берілетін тәртіпте солдан оңға қарай шақырылады, ал конструкторлар кері тәртіпте, сондықтан бағдарлама мынаны экранға шығарады:

- Base1 класының конструкторының жұмысы;
- Base2 класының конструкторының жұмысы;

- Der класының конструкторының жұмысы;
- Der класының деструкторының жұмысы;
- Base2 класының деструкторының жұмысы;
- Base1 класының деструкторының жұмысы.

Виртуалды базалық кластар.

Туынды класс бір базалық класқа бір реттен артық жанама түрде мұрагерленсе және базалық класс барлық туынды кластар үшін виртуалды сияқты мұрагерленсе туынды класс объектісіндегі базалық кластың екі көшірмесінің пайда болуын болдырмауға болады. Мұндай мұрагерлік базалық класқа мұрагерленетін кез келген келесі туынды класта базалық кластың екі (немесе одан да көп) көшірмелерінің пайда болуына жол бермейді. Мұндай жағдайда базалық класқа өту жолының спецификаторының алдына virtual шешуші сөзін қою керек.

С++ тілінің синтаксисі туынды класқа базалық класты тікелей бірнеше рет беруге тыйым салады:

```
Class A {…. };
Class B: A,A{….};          // қате
```

Сонымен бірге базалық класты жанама түрде беруге тыйым салынады:

```
Class A {….};
Class B: public A {….};              // қате
Class C: public A, public B {….}     // қате
```

«Қарапайым» мұрагерлік кезінде туынды класс объектісінің құрамына базалық кластың ішкі объектісі кіреді, ал виртуалды мұрагерлік кезінде туынды класс объектісінің құрамына виртуалды базалық кластың ішкі объектісіне жасырын көрсеткіші енеді, оны компилятор виртуалды базалық кластан мұрагерленген мүшелерге жол беруге арналған объектімен жұмыс барысында айқынсыз қолданады.

```
Class A {…. };
Class B: virtual  public  A {….};
Class C: virtual public A, public B          // осылай жазуға  болады
{….};
```

Мысалы, Der3 класында Base класының екі көшірмесін болдырмайтын виртуалды базалық класс қолданылады:

```
# include<iostrem.h>
class Base
{public : int i ; };
```

```
class Der1 : vertual public Base
{public: int j ; };
class Der2 : vertual public Base
{public: int k ; };
class Der3 :  publicDer1, public Der2
{public: int f( ){return i*j*k;}} ;
int main(void)
Der3 obj ;
```

//Бір мағынасыздық болмайды, себебі Base класының тек бір көшірмесі ұсынылған.

```
obj. i=10; obj.j=3; obj.k=5;
cout « ″ нәтежиесі″ « obj .f( ) « endl;
return 0;
}
```

Бақылау сұрақтары:

1) Кластар иерархиясы қалай анықтауға болады?
2) Объектер иерархиясында хабарламаларды өңдеудің сұлбасы қандай?
3) Егер базалық кластың мүшелері жаңа қайта туынды класта анықталған атаулардың мұрасына ие болса, онда оларға туынды кластан рұқсат беріле ме?
4) Рұқсат етудің қандай спецификаторларын объектілер мұрагерлік иерархиясында хабарламаларды өңдегенде қолдана алады?
5) Қандай жағдайларда виртуалды класс механизмдеріне қатынас жасалады?

8 Дәріс №8. Полиморфизм

Дәрістің мақсаты: кластың үш қасиетінің бірі полиморфизмды түсіндіру.

Дәрістің мазмұны: кластардың виртуалды әдістері. Виртуалды деструкторлар.

Объектіге бағытталған программалаудың (ОБП) үшінші принципі полиморфизм («көп пішімдік») болып табылады. Полиморфизм (операцияларды, функцияларды қайта жүктеу, кластар мен функциялардың мүшелері) операцияның, кластың немесе функцияның бір аттас деректердің түрлі типтері үшін қолданылады.

C++ тілінде полиморфизм екі формаға ие болады. Статикалық байланыстыру механизмі көмегімен жүзеге асырылатын функциялар мен

операцияларды қайта жүктеу. Оған кластар мен функциялардың үлгілері қатысады.

Статикалық байланыстыру операция, кластың, функцияның нақты данасын анықтау бағдарлама компиляциясының сатысында орындалады(осы байланыстыру) .

Виртуалды функцияларды қолдану, бұл динамикалық байланыстыру механизмі арқылы жүзеге асырылады. Бұл форма ОБП негізгі принциптерінің бірі сияқты полиморфизмнің бір формасы болып табылады.

Динамикалық байланыстыру статикалықпен салыстырғанда виртуалды функцияның нақты данасын анықтау компиляция сатысында емес, программа орындаған кезде орындалатындығын білдіреді.

Кластардың виртуалды әдістері.

Компилятордан қандай да бір класс әдісін шақырғанда шақырылатын класс атын компиляция сатысында оған сәйкес машиналық кодпен байланыстырудың екі әдісі бар.

Бірақ егер класс әдісін шақыру үшін аты бір және бірдей аргументтер жиыны бар базалық және туынды кластары бар класқа деген көрсеткіш қолданылса (сигнатурасы бірдей), онда компилятор берілетін объект қай класқа қатысты екендігін және ол үшін қандай әдісті шақыру керектігін анықтай алмайды.

Бұл коллизияны шешу үшін C++ тілі виртуалды функцияларды қолдану мүмкіндігін ескереді. Егер қандай да бір класта virtual сияқты сипатталған әдіс бар болса, онда мұндай класқа виртуалды функциялар кестесіне жасырын мүше-көрсеткіш қосылады, сонымен қатар компиляция кезінде емес, программа жұмысы кезінде деректер тибқнің объектісі үшін сәйкес келетін виртуалды әдісті таңдауға мүмкіндік беретін арнайы кодты басқарады (кейін әдісті біріктіру және динамикалық түрде әдісті байланыстыру).

Виртуалды функция (virtual fuction) базалық кластың ішінде жарияланады және туынды класта қайта анықталады. Негізінен, виртуалды функция полиморфизм негізінде жатқан «бір интерфейс», «көптеген әдістер» ойларын жүзеге асырады. Виртуалды функция базалық кластың ішінде осы функцияның интерфейсі түрін анықтайды. Әрбір виртуалды қайта анықталуы туынды класта оның спецификациясымен байланысты жүзеге асырылуын анықтайды. Сөйтіп қайта анықтау нақты әдісті құрады.

Егер екі немесе одан да көп түрлі кластар виртуалды функция құрамына жататын базалықтан туындылар болып табылса, онда, егер базалық кластың нұсқауышы осы туынды кластардың түрлі объектілеріне сілтеме жасаса, виртуалды функциялардың түрлі нұсқаларын орындайды. Себебі компилятор нұсқауыш сілтеме жасайтын объект типіне негізделе отырып виртуалды функция нұсқасын анықтайды. Бұл үдеріс динамикалық полиморфизм принциптерін жүзеге асыру болып табылады.

Виртуалды функциялардың иерархиялық тәртібі бар программа мысалын қарастырайық:

```
# include<iostream.h>
class Base
{public : int i ;
Base (int x) {i=x;}
Vertual void func ( )
{cout «″ базалық кластың func ( ) функциясын орындау″;
cout « i « endl ;}
};
class Der1 :  public Base
{public: Der1 (int x): Base (x) {}
void func( )
{cout″ Der  кластың func ( ) функциясын орындау: ″;
cout «  i*i « endl;}
};
class Der2 :  public Base
{public: Der2 (int x): Base (x) {}
func ( ) функциясы ауыстырылмайды
};
int main(void)
Base *p; Base obj (10); Der1 d1_obj(10); Der2 d2_obj(10);
p=& obj;
p- func( )                          // базалық  кластың func ( ) функциясы
p=&d1_obj;
p- func( ) ;                        // Der1 туынды класының func ( ) функциясы
p- func( )                          // базалық  кластың func ( ) функциясы
return 0 ;
}
```

Егер виртуалды функция туынды класта қайта анықталмаған болса, онда базалық кластан оның нұсқасы қолданылады. Объект көрсеткіші арқылы адрестелетін тип виртуалды функциямен ауыстырылатын белгілі бір нұсқасын шақыруды анықтайды. Бұған қарсылық ретінде виртуалды емес нұсқауыш арқылы шақырылатын интерпретация нұсқауыш типіне ғана тәуелді.

Бірқатар виртуалды әдістермен жұмыс істеудің негізгі ережелерін қарастырайық:

1) Виртуалды болып кез келген функциялар емес, тек қандай да бір клатың компоненттік функциялары табылады.

2) Функция виртуалды болып анықталған соң, оның туынды класта қайта анықталуы (сигнатурасы бірдей) бұл класта жаңа виртуалды функцияны құрады, сонымен виртуал спецификаторы енді қолданылмауы мүмкін.

3) Туынды класта аты мен сигнатурасы бірдей функцияны, бірақ базалық кластың виртуалды функциясымен салыстырғанда нәтижесінің басқа типімен анықтауға болмайды.

4) Егер виртуалды әдіс туынды класта қайта анықталмаса, онда осы туынды класс объектісі үшін оны шақырған кезде ол анықталған базалық кластың иерархиясы бойынша жақын виртуалды сәйкес әдіске қатынасатын болуы керек.

5) Туынды класс мүшесі базалық кластың виртуалды әдіс атымен сай келетін әдіс аты, бірақ сигнатурасы басқа болуы мүмкін. Онда ол басқа виртуалды емес әдіс болады.

6) Виртуалды әдістер мұрагерленеді, яғни олар туынды класта қайта анықтау айырмашылығы бар әрекеттерді беру керек болған кезде ғана талап етіледі. Қайта анықтау кезінде рұқсат ету құқықтарын өзгертуге болмайды.

7) Егер виртуалды әдіс туынды класта қайта анықталған болса, онда осы класс объектілері көріну облысына (:: операциясы) рұқсат ету операциясы көмегімен базалық кластың виртуалды әдісіне өту рұқсатын ала алады.

Виртуалды деструкторлар.

Класс конструкторы виртуалды болуы мүмкін емес, себебі ол типі белгілі объектті құрған кезде ғана шақырылады. Деструктор виртуалды бола алады. Оның қажеттілігі базалық класқа нұсқауышпен адрестелетін объекттің бұзылуымен байланысты. Егер бұл нұсқауыш өзінің жеке деструкторы бар туынды класс объектісіне сілтеме жасаса, онда көрсетілген объекттердің корректті бұзылу мәселесі виртуалды әдістерге ұқсас виртуалды деструкторлармен шешіледі. Егер осындай жағдайда деструктор виртуалды сияқты жарияланса, онда барлығы дұрыс орындалып, сәйкес туынды кластың деструкторы шақырылатын болады. Кейін туынды класс деструкторы автоматты түрде базалық класс деструкторын шақырады және белгіленген объект тұтастай жойылады. Одан мына ереже шығады: егер класта виртуалды әдістер жарияланса, онда деструктор да виртуалды болу керек.

Бақылау сұрақтары:

1) Туынды класс объект үшін қандай да бір класс әдісін шақыруға бола ма?
2) Егер жұмыс көрсеткіштері қолданумен жүргізілсе, онда кластар иерархиясында виртуалды әдістер не үшін керек?
3) Қайта жүктелген әдіс құрамына үнсіздік бойынша берілген параметрлер ене ала ма?
4) Қандай жағдайда класс әдістерін виртуалды жасау керек емес?
5) Қандай түрде C++ тілінде полиморфизм жүзеге асырылады?

9 Дәріс №9. Абстрактылы кластар

Дәрістің мақсаты: абстрактты кластар түсінігін беру.

Дәрістің мазмұны: Локальды кластар. Класс құрылымы бойынша жасалатын ұсыныстар.

Ереже бойынша базалық класта жарияланған виртуалды функция ешқандай маңызды әдістерді орындамайды. Бұл әдеттегі жағдай, себебі жеке түрде базалық класта деректердің аяқталған типі анықталмайды. Оның орнына тек мүше функциясының базалық жиыны мен жетпейтіндердің бәрін туынды класс анықтайтын айнымалылар болады. Базалық кластың виртуалды функциясында негізгі әрекет болмаған кезде осы базалықтан кез келген туынды класта мұндай функция міндетті түрде қайта анықталған болуы керек. Оны C++ тілінде жүзеге асыру үшін таза виртуалды (pure virtual function) деп аталатындар қолдау табады.

Таза виртуалды функциялар абстрактты әдістер базалық класта анықталмайды. Оған тек қана осы функциялардың прототиптері енеді. Таза виртуалды функция үшін осындай негізгі форма қолданылады:

аты _типі (параметрлер_ тізімі) = 0

Функцияларды нөлге теңестіру компиляторға базалық класта осы функцияның денесі жоқ екендігін хабарлайды. Мұндай жағдайда ол әрбір туынды класта міндетті түрде ауыстырылып отыруы керек. Әйтпесе компиляция кезінде қате туындайды.

Ең болмаса бір абстрактты әдісі бар класс абстрактты деп аталады.

Абстрактылы класс басқа кластар үшін базалық ретінде қызмет ете алады және абстрактылы класс объектісін құру мүмкін емес. Абстрактылы кластан туынды кластар, абстрактты әдістер анықталуы керек немесе абстрактты сияқты қайта жариялануы керек.

Абстрактты класс бар деп есептейік:

```
Class Base
{  protected :virtual void f(char)=0;
void func(int);
};
```

Base абстрактты класс негізінде туынды класты түрінде құруға болады:

```
Class Der1:public Base
{….void f (char);   };
Class Der2:public Base
{….void func  (int);   };
```

Der1 класында f ( ) абстрактты әдісі осы типтің нақты виртуалды функциясымен ауыстырылған. Base: : func ( ) функциясы Der1 класымен мұрагерленеді және оның әдістері мен қатынауын мұрагерленеді. Der1 класы абстрактты емес. Der2 класында Base:: func ( ) функциясы қайта анықталған,

ал Base:: f () виртуалды функциясы мұрагерленген. Сонымен бірге класс Der2 абстрактылы болады және базалық ретінде ғана қолданылады.

Әрбір клас сияқты абстрактты класс анықталған конструкторға ие болуы мүмкін. Конструктордан кластар әдістерін шақыруы мүмкін, бірақ кез келген тура немесе тура емес таза виртуалды функцияларға қатынасы бағдарламаны орындаған кездегі қателерге әкеледі.

Кейін нақтылауға ұсынылатын пәндік облыстың жалпы түсініктерін беру үшін абстрактылы кластар механизмі өндірілген. Бұл жалпы түсініктерді әдетте тікелей қолдану мүмкін емес, бірақ құруға жарамды жеке туынды кластарды құруға болады. Мысалы, абстрактты «Фигура» класынан «Үшбұрыш», «Шеңбер» және тағы басқа кластарды құруға болады.

Абстрактылы әдістер кластармен жұмыс істеуге арналған анағұрлым маңызды ережелерді құрайық:

1) Әдіс параметрінің типі, мәнді қайтаратын әдіс сияқты абстрактылы класс бола алмайды.

2) Егер абстрактылы базалық класқа оның инициализациясы үшін уақытша объектті құру талап етілмесе, онда көрсеткіш құруға, сонымен қатар осындай класқа жасалатын сілтемеге рұқсат етілген (және ол жиі қолданылады).

3) Абстрактылы класс әдістері осы кластың абстрактылы әдістерін шақыра алады. Осындай жағдайда объект типіне сәйкес келетін туынды класта анықталған әдіс шақырылатын болады.

4) Егер абстрактылы кластан туынды класта барлық абстрактылы әдіс функциялары шақырылса, онда туынды класс соған қоса абстрактылы болып табылады.

5) Абстрактылы кластан кез келген туынды класс одан абстрактылы әдістерін мұрагерленеді. Туынды кластың объектісін құру мүмкіндігін алу үшін онда барлық абстрактылы әдістерді қайта анықтау керек.

6) Абстрактылы класс конструкторлар мен деструкторларға ие бола алады.

Абстрактылы кластың конструкторы туынды класс объектілерін құрған кезде, ал деструктор олардың қатынауы кезінде шақырылатын болады. Базалық класс деструкторы туынды кластарда анықталған «ішкі объектілер» бұзылғаннан кейін шақырылады. Сондықтан абстрактылы базалық класс деструкторы өз класының абстрактылы әдістерін шақырмау керек, себебі мұндай шақыру бағдарламаны орындағанда қатеге әкеледі.

Локальды кластар.

Класс блоктың ішінде берілуі мүмкін, мысалы функция анықтамасының ішінде. Мұндай класс локальды деп аталады. Класс локализациясы кластың анықталу облысынан тыс оның компаненттеріне байланысты қатынамауды тұжырымдайды (ол анықталған немесе сипатталған функция немесе блоктың денесінен тыс).

Локальды класс статикалық деректерге ие бола алмайды, себебі локальды кластың компоненттері класс мәтінінен тыс анықталуы мүмкін емес.

Локальды кластың ішінде оның айналасындағы облысынан тек типтер атауларын, статикалық (static), сыртқы (extern) айнымалыларды, сыртқы функциялар мен санауыш элементтерін қолдануға рұқсат етілген. Локальды кластардың компоннетті функциялары тек енгізілген (inline) болуы мүмкін.

Класс құрылымы бойынша жасалатын ұсыныстар.

Әдетте класс қолданушы типі сияқты құрамына жасырын (private) мүше - деректердің саны және келесі әдістері кірмейді:

- класс объектілерін инициализациялаудың әдістерін анықтайтын конструкторлар және қажет болғанда динамикалық жадыны резирверлеуді қамтамасыз ететін конструкторлар;
- көшіріп алу конструкторы (егер класс динамикалық жады үшін қолданылса қажет болады);
- деструктор (егер класс динамикалық жады үшін қолданылса қажет болады);
- класс қасиеттерін жүзеге асыратын әдістер жиыны (бұл кезде кластың private – мүше – деректерінің мәндерін қайтаратын әдістер өрістер мәндерін өзгерте алмайтындығын көрсететін const модификаторымен сипатталуы керек);
- класта қажет етілетін объекттерді салыстыруға, өзіне алуға, арифметикалық және басқа әрекеттерді орындауға мүмкіндік беретін операциялар жиыны;
- қателер туралы хабарламалар үшін қолданылатын ерекшеліктерді өңдеу құрылғылары.

Класты жобалаған кезде келесі сұрақтар өте маңызды болып табылады.

Кластың мүше - деректер құрылымын және олардың тиесілігін қалай анықтайды? Класс әдістері мен тиесілігін қалай анықтайды?

Егер берілген класс әдістермен ғана қолданылса, онда оны жапқан жөн болады. Егер берілген класс пен оның туындыларында қатынау қажет болса, онда оны қорғау керек. Мүше - деректерді ешқашан ашпаған жөн, себебі класс әлсіз болып қалады, ал бұл программалық кодты сенімсіз етеді.

Класс пен қолданушы кластың интерфейсті әдістері мен деструкторлары, конструкуторларының барлық түрлері көмегімен байланыса алады, сондықтан оларды кез келген программалық ортадан рұқсат етілген болуын жасау керек. Көмекші интерфейсті емес, егер олар тек осы класта, немесе қорғау (protected), егер олар осы және одан да туынды кластарда керек болса әдістерді жабық (private) ету керек.

Бақылау сұрақтары:

1) Виртуалды функция дегеніміз не?
2) Қандай функциялар виртуалды бола алмайды?
3) Қалай виртуалды функциялар динамикалық полиморфизмді жүзеге асыруға көмектеседі?
4) Абстрактылы класс дегеніміз не?
5) Полиморфты класс дегеніміз не?

10 Дәріс №10. Әрекеттер аймақтары мен атаулар кеңістіктері

Дәрістің мақсаты: әрекеттер аймақтары мен атаулар кеңістіктерін ашып көрсету.

Дәрістің мазмұны: блок. Функция прототипі. Функция. Файл. Класс. Атаулар кеңістігі.

Әрбір программалық объекттің оның түрі мен орнын анықтайтын әрекет ететін аймағы мен өміршең уақыты бар. Әрекеттер аймағының келесідей түрлері бар:

- блок;
- функция прототипі;
- функция;
- файл;
- программалық жобаның барлық файлдарын қамтитын шеңберіндегі файлдар тобы (әрекеттердің глобальды аймағы);
- класс;
- атаулар кеңістігі (әрекеттердің глобалды аймағының бөлігі).

Программалық объекттің әрекеттер аймағының барлық бес категориясын қысқаша қарастырып өтейік.

Блок. Блок ішінде анықталған объект локалды болып саналады. Осындай обьекттің әрекеттесуінің аймағы анықталу нүктесінде анықталады және блок аяғында аяқталады. Сақтау класы обьектісінің (автоматты) өмір сүру уақыты оның анықталу мерзімінен басталады және блок жұмысы біткеннен кейін аяқталады. Static (статикалық) спецификаторы бар блок объектісі өз мәнін блоктың аяқталуынан кейін сақтайды, ал оның өмір сүру уақыты бағдарламаның орындалу уақытымен сәйкес келеді.

Функция прототипі. Функцияның прототиптер (хабарламалар) параметрлер тізімінде көрсетілген идентификаторлардың әрекеттер аймағы ретінде тек функция прототипі бар. Сондықтан функция параметрлерінің туынды идентификаторларын қолдануға болады және олардың мүлдем жіберіп қоюға болады.

Функция. Функция блогында анықталған бағдарламалық объектілер қалыпты блокта сияқты әрекеттер аймағы және өмір уақытына иеленеді. Мән бойынша берілетін функция параметрлері әрекеттер аймағы ретінде бүкіл функциясы бар және өмір уақыты - функциясының орындалу уақыты.

Сілтеме бойынша берілетін функция параметрлері функция шақырылуына сәйкес аргументтермен анықталатын өмір уақытымен әрекеттер аймағына, әрине, функция блогы да кірмейді.

Файл. Атаулар кеңістігі немесе класс, функция, кез келген блоктан тыс static сақтау класының сипаттауышын қолдану көмегімен анықталған программалық объект анықтау нүктесінде басталатын және файл соңында

аяқталатын әрекеттер аймағына ие болып табылады. Егер құрамында идентификаторы бірдей болып табылатын қайта анықталған программалық объект болмаса, онда әрекеттер аймағына енгізілген (ішкі) блоктар қосылады. Егер енгізілген блокта бірдей идентификатор, қайта анықталған объект бар болса, онда бұл жағдайда сыртқы объект енгізілген блокта көрінбейді. Оған, егер глобальды болса (атаулар кеңістігіне немесе кластан, блоктан тыс анықталған), «::» көріну аймағына өтуді рұқсат ету операциясының көмегімен қатынасуға болады. Мұндай объекттің өмір уақыты максималды және программаны орындау уақытына сәйкес келеді.

Программалық жобаның (әрекеттердің глобалды аймағы) барлық файлдарын шегіне қосатын файлдар тобы. Extern (сыртқы) класының сақтау класының сипаттаушысын қолдану көмегімен басқа файлдардағы және жарияланған, атаулар кеңістігі немесе класс, функция, блоктан тыс жобаның файлдарының біреуінде анықталған программалық объекттің әрбір осындай файлдарын жариялау немесе анықтау нүктесінде басталатын және файл соңында аяқталатын әрекеттер аймағына ие. Егер құрамында бірдей идентификаторы бар программалық объекттің өмір уақыты максималды және оны орындау уақытымен сай келсе, онда әрекеттер аймағына енгізілген (ішкі) блоктары қосылады.

Класс. Статикалық класс мүшелерінен басқа обьект кластарының мүшелері әрекеттер аймағын иеленеді. Бұл олар класс ішінде ғана көрінетінін білдіреді. Объект кластар мүшелерінің өмір уақыты – объект класын құру мерзімінен бастап оның бұзылу мерзіміне дейінгі уақытта анықталады. Берілген класс құрамын сипаттау, кәдімгі статикалық айнымалылар мен функциялардан айырмашылығы сияқты көп файлдық жобаның барлық файлдарында қолжетімді деректердің статикалық мүшелері және әдістер глобальды болып табылады.

Деректердің статикалық мүшелері кәдімгі глобалды айнымалылар сияқты жоба файлдарының біреуінде міндетті түрде анықталуы тиіс. Деректердің статикалық мүшелері және функция-мүшелер бір объекттіні құрғанға дейін де рұқсат етілген болады. Оларға «::» операторы арқылы қатынасуға болады.

Атаулар кеңістігі. C++ тілі name space операторы арқылы атаулардың әрекеттер аймағын глобаьлды бөлігі сияқты нақты түрде беруге мүмкіндік береді. Әрбір әрекеттер аймағында атаулар кеңістігі деп аталатындарды айырады. Атаулар кеңістігі – идентификатор әмбебап болып табылатын аймақ. Түрлі атаулар кеңістігінде идентификаторлар бірдей болуы мүмкін, себебіі сілтемелер шешуші бағдарламадағы идентификатор контексті бойынша жүзеге асырылады, мысалы:

```
struct Node
{  int Node;
int I;
} Node;
```

Бұл жағдайда қарсылықтар жоқ, себебі тип атаулары, құрылым айнымалылары мен өрістері атаулардың түрлі кеңістіктеріне қатысты болады.

С++ тілінде атаулар кеңістігінің төрт түрі анықталған, олардың әрбіреуінің шектерінде идентификаторлар әмбебап болуы керек:

- қолданушымен анықталған типтер, функциялар, айнымалылар (объекттер) идентификаторларға қатысты атаулар кеңістігі және әрекеттердің бір аймақтық шектеріндегі константалардың аталып өтуі. Функциялар идентификатоларынан басқаларының барлығы енгізілген блоктарда қайта анықталуы мүмкін;
- бірлестіктер, кластар, құрылымдар, атап өтулер типтерінің атауларын түзетін кеңістік. Бұл атаулар кеңістігінде әрбір осындай идентификатор бір әрекеттер аймағындағы шектерде әмбебап болуы керек;
- атаулардың бөлек атаулары әрбір кластың мүшелерін құрады. Класс мүшесінің аты класс ішінде әмбебап болуы керек, бірақ басқа кластардың мүшелерінің атауларымен бірдей болуы мүмкін;
- белгілер бөлек атаулар кеңістігін түзеді.

Атаулар кеңістігі. Атаулар кеңістігі (аталған аймақ) анықтамалар, хабарламаларды логикалық түрде топтастыру үшін және оларға қатынауды шектеу үшін қызмет етеді. Бағдарлама өлшемі неғұрлым үлкен болған сайын соғұрлым аталған аймақтарды қолдану өзекті. Түрлі атаулар кеңістігінің көмегімен атаулар қақтығысы мен сәйкес келу мүмкіндіктерінсіз басқа бағдарламалаушымен жазылған кодтан бір бағдарламалаушымен жазылған кодты анықтауға болады.

Атаулар кеңістігінің жариялануы (аталған аймақ) келесі форматтарға ие:

Namespace [аймақ _ аты] {/ *анықтамасы және хабарламасы */….}

Атаулардың бір кеңістігі бір рет жариялануы мүмкін емес, сонымен келешек хабарламалар алдыңғылардың кеңейтулері ретінде қарастырылады. Сөйтіп, атаулар кеңістігі бір файлдың шектерінен тыс өзгеруі және хабарлануы мүмкін.

```
Namespace demo
{int I=1;                    // объектті анықтау
int k=0;                     // объектті анықтау
void func 1(int);           // функция прототипі
void func2(int r){…}  // функция анықтамасы
}
// demo атаулар кеңістігі
namespace demo
{
// int I=2 ;          дұрыс емес - екі анықтама
```

```
void func1(double);        // дұрыс – функция прототипі (қайта жүктеу)
void func2 (int);          // дұрыс - функция прототипі
}
```

Атаулар кеңістігінің хабарламасына тек хабарламаларды логикалық түрде орналастыру болады, ал кейіннен олар аймақ аты мен «**::**» көріну аймағының қатынау операторы көмегімен анықталады, мысалы:

```
void demo:: func1 (int n) {…}
```

Мұндай әдіс интерфейстің бөлінуі мен жүзеге асырылуын қамтамасыз етеді (сөйтіп, атаулар кеңісігінің жаңа атын хабарлауға болмайды).

Егер атау жиі өз кеңістігінен тыс қолданылса, онда оны using операторы арқылы мүмкін етуге болады:

```
Using demo :: i ;
```

Одан кейін аймақтың нақты белгіленуінсіз (i) атауы қолданылуы мүмкін. Егер қандай да бір аймақтан барлық атауларды мүмкін ету қажет болса, using namespace операторы қолданылады:

```
using namespace  demo ;
```

using және using namespace операторларын басқа аймақтан анықтамалар мен хабарламалардың мүмкін болуын жасау үшін аталған аймақтың хабарламасының ішінде де қолданылады.

Бақылау сұрақтары:

1) Атаулар кеңістігін қолдану міндетті ме?
2) Using және using namespace қолданудың арасында қандай айырмашылық бар?
3) Атаулардың аталған кеңістіктері дегеніміз не және олар не үшін керек?
4) using қызметші сөзін қолданбай атаулар кеңістігінде хабарланған идентификаторларды қолдануға бола ма?
5) std атауларының стандартты кеңістігі дегеніміз не?

11 Дәріс №11. Стандартты кітапхана құрылғыларымен C++ тіліндегі енгізу-шығару.

Дәрістің мақсаты: C++ тілінің стандартты кітапханасындағы енгізу-шығару ағындары кластарын түсіндіру.

Дәрістің мазмұны: контейнерлі кластар, алгоритмдер және итераторлар кіретін стандартты кітапханасының бөлігі, үлгілердің стандартты кітапханасы.

Тілдің операторларынан басқа кез келген программа құрудың интегрирленген ортасына қосылатын түрлі кітапханалардың құрылғылары қолданылады. C++ тілінің стандартты кітапханасын екі бөлікке бөлуге болады:

- C++ тілінің кітапханасынан мұрагерлік константалар, типтер, макростар, функциялар;
- стандартты кластар және C++ тілінің басқа құрылғылары.

C++ тілінің стандартты кластарын келесі топтарға бөлуге болады:

- сыртқы құрылғылар мен жедел жады арасындағы, сонымен қатар жедел жады шектерінде деректер ағынын басқаруға арналған ағынды кластар (дисктер, пернетақта, экран, монитор);
- символдық жолдары бар жұмыстың қателерінен қорғалған және қолайлы жолдық класс;
- контейнерлі кластар, алгоритмдер және итераторлар. Контейнерлі кластар деректерді сақтаудың кең таралған танымал құрылымдарымен: тізімдер, векторларды, көпмүшеліктерді және тағы басқа сақтауларды жүзеге асырады;
- стандартты кітапхана құрамына компоненттерді түрлендіретін және қолданатын алгоритмдер жатады. Итераторлар контейнерлі кластар элементтеріне унифицирленген қатынауды қамтамасыз етеді;
- жылжымалы нүктелі массивтер мен кешенді деректерді тиімді өңдеуге арналған математикалық кластар;
- типтер идентификациясы мен қателердің объекттіге бағытталған өңдеуге арналған диагностикалық кластар;
- жадыны динамикалық тарату, локальды ерекшеліктерге бейімделу және тағы басқалар үшін қалған кластар.

Алгоритмдер және итераторлар кіретін стандартты кітапханасының бөлігін үлгілердің стандартты кітапханасы деп аталатын (Standart Template Library,STL) контейнерлі кластар.

C++ тілінің стандартты кітапханасының ағынды кластарын қолдануға негізделген C++ тілінің енгізу-шығару құралдарын қарастырайық.

Ағын - қоректену көзінен қабылданушыға деректерді тасмалдауға қатысты түсінік. C++ тілінің ағындары деректер типінің стандартты және анықталған қолданушылармен сенімді жұмыспен қамтамасыз етеді, сонымен қатар бірегей және синтаксисімен түсінікті.

Ағыннан деректерді оқу оларды ағыннан алу деп аталады, ағынға шығару - ағынға деректерді қосу немесе орналастыру. Ағын байттар тізбегі ретінде анықталады және алмасу орындалатын нақты қондырғыға тәуелсіз болады.

Алмасу бағытына қарай ағындар кіріс (деректер енгізіледі), шығыс (деректер шығарылады) және екі бағытты деп бөлінеді.

Ағын жұмыс істейтін қондырғы түрлі бойынша ағындарды стандарты, файлдық, жолдық деп бөлуге болады.

Стандартты ағындар деректерді пернетақталардан дисплей экранына беруге арналған файлдық ағындар - магнитті дискідегі ақпаратпен алмасу үшін, ал жолдық ағындар - символдардың масивтерімен жұмыс істеу үшін қолданылады.

Ағындарды қолдану үшін C++ тілінің кітапханасына ios және streambur екі базалық класс негізінде құрылған кластар иерархиясы кіреді. Ios класынан енгізу – шығаруға арналған өріс пен әдістер кіреді. Streambur класы ағындардың буферизациясы мен оның физикалық қондырғылармен олардың әсерлесуін қамтамасыз етеді. Бұл кластардан stream класы кіріс ағындар мен ostream класы – шығыс үшін мұрагерленеді.

Бұл кластар екі бағытты ағындарды жүзеге асыратын iostream класы үшін базалық болып табылады. Кейін кластар иерархиясында файлдық және жолдық ағындар орналасқан.

Си тілінің енгізу-шығару стандартты функциясымен салыстырғанда ағындардың негізгі артықшылығы типтерді бақылау болып табылады, сонымен қатар кеңейтілуі, яғни қолданушымен анықталған типтермен жұмыс істеу мүмкіндігі болып табылады. C++ тілі енгізу-шығаруды жүзеге асыруға араналған құрылғылардың кең жиынын береді. C++ тіліндегі енгізу-шығару бұл - iostream класының “<<” және “>>” операцияларды қайта кең қолдану.

Енгізілген типтердің енгізу-шығаруды (стандартты) сәйкес кластар және объектілермен, соның ішінде шарт алдын ала анықталған объектілерден қолдау табады: cin, cout, cerr және clog.

Мысалы, cout«x; операторы шығару үшін iostream класының cout объектісіне x айнымалысының мәнін жібереді. Қайта жүктелген operator<< әдісі ол үшін шақырылған iostream объектісіне сілтемені қайтарады. Сондықтан шығару операциясының негізінде тағы бір шығару операциясын қолдануға болады, яғни шығарудың бірнеше операциясын бір тізбекке қосуға болады.

include<iostream.h>
int x; cout«″ =″ « x«″ \n″;

″ «″ операциясында солдан оңға қарай есептеу тәртібі бар екендігін ескеріп, мұндай жазба келесі түрде интерпретирленеді:

((cout. operator «(″ x=″)) operator « (x)). Operator « (“\n);

Берілген аргументтің типіне тәуелді operator<< әдісінің белгілі бір данасы шақырылады.

C++ тілінде шығуға ұқсас ену анықталады. Енгізілген стандартты тип үшін ″«″ операциясы қайта жүктелген анықталған iostream класы байланысты. Сонымен бірге operator<< әдісі iostream класс объектісіне сілтемені қайтарады, және енгізу кезінде енгізу операцияларын біріктіруге болатындығын білдіреді, мысалы, үш аргументті:

// cin .operator» (x)) . operator» (y) . operator» (z) ;

Енгізу және шығарудың нақты операциясын таңдау енгізілген немесе шығарылған деректердің әрбір типімен анықталады.

Енгізілген типтермен қатар C++ тілі жаңа типтерді (қолданушымен анықталған типтер) анықтауға мүмкіндік береді.

Ол үшін қайта жүктелетін функцияны қолданумен бірге әрбір қолданушы типі үшін “<<” және “>>” операциясын қайта жүктеу керек. Қайта жүктейтін әдіс “<<” немесе “>>” операциясы үшін міндетті түрде класқа дос болуы керек, бірақ кластың мүшесі болмау керек, себебі мұнда операциялардың операндалары түрлі кластан алынған.

Енгізу-шығару операцияларын қайта жүктеу бөліктерін көрсететін мысалды келтірейік:

```
Class Complex          // класс кешенді сан
{ int real, image;
public:
Complex ( int re, int im) {real=re; image=im;}
friend ostream & operator« (ostream &,const Complex &);
friend istream & operator» (istream &, Complex &);
};
ostream & operator« (ostream & obj_out, const Complex &val_out)
{ return obj_out « val_out.real «″i″ « val_out.image «endl;}
// obj_out- шығарылатын объект , val_out шығарылатын мән
Енгізу үшін ″>>″ операциясын қайта жүктеу.
istream & operator » (istream &obj_in, Complex &val_in)
{ obj_in» val_in.real>>val_in.image;
if(!obj_in) { cout «″ \n енгізудегі қате ″ « endl;exit(1);}
return 0;
}
int main (void)
{Complex obj(12,21);
cout«″ объект мәні: «″obj;
cout«″\n кешенді мәнді енгізуіңіз : ″ ; cin »obj;
cout«″\n объект мәні: obj=“« obj;
return 0;
}
```

Ағынды кластарда форматтауды үш әдіспен: манипуляторлар көмегімен (ең қолайлысы), форматтау жалауларымен және форматтау әдістері көмегімен орындауға болады.

Ағынды кластар құрамына дәлдігі әртүрлі, белгілі бір сандар жүйесіне мәндерді енгізу немесе шығаруға мүмкіндік беретін құралдар кіреді.

Шығарылатын мәндер шығару өрісінің сол немесе оң жақ соңына қысуға болады. Форматталған енгізу-шығару құрылғылары енгізу-шығару форматының басқа да бөлшектерін басқаруға мүмкіндік береді. Берілген мүмкіндіктерді жүзге асыруға, анағұрлым қолайлы әдістерінің бірі манипуляторларды қолдану болып табылады.

Манипуляторлар, форматталған енгізу-шығаруды кодтауды жеңілдететін, қайта жүктелетін әдістерді қолданады. Мапипуляторларды - “<<” және “>>” операциялары көмегімен енгізу-шығару тізбегіне енгізуге болады. Манипуляторлар, параметрленген болады, яғни бір аргументі, бар функцияны шақыру түрінде жазылады, және қарапайым яғни манипулятор атынан кейін аргументі бар дөңгелек жақшалар болмайды. Параметрленген манипуляторларды қолдану үшін бағдарламаға <iomanip.h> файлын қосу керек.

Бақылау сұрақтары:

1) cerr мен clog арасында қандай айырмашылықтар бар?
2) Енгізудің қайта жүктелген операторы дегеніміз не және ол қалай жұмыс істейді?
3) Шығарудың қайта жүктелген операторы дегеніміз не және ол қалай жұмыс істейді?
4) Шығарудың қайта жүктелген операторы қандай мәнді қайтарады?

12 Дәріс №12. Файлдық енгізу-шығару

Дәрістің мақсаты: файлдық енгізу-шығару кластарын қарастыру.

Дәрістің мазмұны: istream, ostream, iostream, ofstream, fstream кластарының объектісі.

Файлдық және консольды енгізу-шығару өте тығыз байланысты. Нақты түрде файлдық енгізу-шығару консольды енгізу-шығару сияқты сол кластар иерахиясынан қолдау табады.

Программа өз жұмысын бастаған кезде, төрт ағын автоматты түрде инициализацияланады (жүктелінеді) және оларға сәйкес стандарттты файлдармен (қондырғылармен) байланысады – бұл cin, cout, cerr және clog. Егер басқа ағындарды инициялизациялау және кейбір файлдармен байланыстыру керек болса, онда мұны айқын түрде істеу керек.

Кез келген инициализацияланатын файлдық ағын енгізу ағыны немесе шығару ағыны немесе енгізу-шығару ағыны болу мүмкін. Енгізу ағыны бұл istream класынан туындаған, istream класының объектісі. Шығару ағыны – бұл ostream класынан туындаған, ofstream класының объектісі. Енгізу-шығару ағыны - бұл iostream класынан туындаған, fstream класының объектісі. Осы кластардың әрбіреуінде, класс объектісін сәйкес файлмен біріктіру жұмысын орындайтын конструкторы бар. Файлдармен жұмыс істеу үшін кластар,

ағынды кластардан қайта жүктелген операцияларды ″«″және ″»″, жалауларды және форматтау әдістерін, манипуляторларды, ағындар күйін тексеру құрылғыларын және тағы басқаны мұрагерленеді.

Программада файлдарды қолдану келесі әрекеттерді тұжырымдайды:

1) Ағынды құру.
2) Ағынды ашу және оны файлмен байланыстыру.
3) Ағынмен алмасу (енгізу-шығару).
4) Файлды жабу.
5) Ағынды жою.

Параметрлері жоқ конструкторлар объектті файлмен байланыстырмай құрады. Параметрлері бар конструкторлар объект құрады, берілген режимде көрсетілген аты бар файлды ашады және файлды объектімен байланыстырады. Егер үнсіздік бойынша файлды ашу режимінің мәні программист үшін қолайсыз болса, онда операция арқылы, ios класында анықталған, биттік маскалардан құралған басқа мәнін орнатуға болады.

Ағынды құрған соң, оны файлмен байланыстыру әдістерінің бірі, ағынды кластардың барлық файлдарының мүшесі болып табылатын, open ( ) функциясы болып табылады. Егер open ( ) функциясының орындалуы қатемен аяқталса, онда бульдік өрнекте ағын false мәніне тең болады. Файлды жабу үшін қайтарылатын мәні мен параметрлері жоқ close ( ) функциясы колданылады.

Программалаудың негізгі, мынадай аспектілерін бейнелейтін бір тәжірибеде маңызды мысалды қарастырайық: ағынды енгізу-шығару; туынды класс конструкторынан, базалық класс конструкторына параметрлерді берген кезде, класс үлгілерін қолдану; динамикалық жадыда орналастырылған екі өлшемді массивпен жұмыс. Келтірілетін программмада аспектілердің тек көрсетілген ерекшеліктері ғана қарастырылады, ал бақылау түрлері ескерілмеді.

```
// Файлдарды ашу/жабу үшін IOFile класын анықтау
Class IOFile : public fstream
{ public : void open_f(char *filename, int mode ){ open (filename, mode );
void close _f(char * filename){ close ( );}
};
// матрицалармен жұмыс істеуге арналған, базалық кластар үлгісі
template<class T>       // Т-элементітің жалпы типі
class Matrix_B        // класс-«Матрица» - базалық класс
{ protected:
T**array;             // жолға көрсеткіштер массивінің көрсеткіші
int rows, columns;
public: Matrix_B (int n, int m);              // конструктор
Matrix_B (const Matrix_B <T>&copy);        // көшіру конструкторы
~Matrix_B (void);   // деструктор
void ReadMatrix (char*fp);      // файлдан матрица элементтерін оқу
```

```
};
template<classT> Matrix_B<T>:: Matrix_B(int n, int m)
{
array=new T*[n];  // матрицаға сай жадының динамикалық бөлінуі
for(int i=0; i<0; i++) array[i]=new T[m];
for(i=0; i<n; i++)
for(int j=0; j<m; j++) array[i][j]=(T)o;
rows=n; columns=m;
        return;
}
// көшіру конструкторы
template<classT>Matrix_B<T>::Matrix_B(constMatrix_B<T>&copy)
{ array=new T*[copy.rows];
for(int i=0; i<copy.rows; i++)
array[i]=new T[copy.columns];
for( i=0; i<copy.rows; i++)
for(int j=0; j<copy.columns; j++)
  array[i][j]=copy.array[i][j];
rows=copy.rows; columns=copy.columns;
}
template<class T>Matrix_B<T>::~Matrix_B(void)         // деструктор
{ for(int i=0; i<rows; i++)   delete [] array[i];
delete [] array;
}
// берілген файлдан матрица элементтерін оқу
template<classT>void Matrix_B<T>::ReadMatrix(char*fp)
{ IOFile fin;   fin.open_f(fp,ios::in);
for(int i=0; i<rows; i++)
for(int j=0; j< columns; j++) fin>>array[i][j];
fin.close_f(fp);
return;
}
// матрицалармен жұмыс істеу үшін туынды класс үлгісін анықтау
template<classT>
class Matrix_D:public Matrix_B<T>
{ public: Matrix_D(int nrow, int ncol);
void PrintMatrix(IOFile &);      // матрицаны басып шығару
//қосу, меншіктеу және индексациялау амалдарының жүктелуі
операцияларын жүктеу
Matrix_D<T>operator+(const Matrix_D<T>&);
Matrix_D<T>2operator=(const Matrix_D<T>&);
T*& operator [ ] (int index);
// туынды класс конструкторы
template<classT>
```

```
Matrix_D<T>: : Matrix _D (int nrow, int ncol): Matrix_B<T>( nrow, ncol){
}
// Қосу операцияларын қайта жүктеу
template <class T>
Matrix_D<T> Matrix_D<T>: : operator + (const Matrix_D<T>&ref)
{ Matrix_D<T> result (rows, columns); // бос объектті құру
for(int i=0; i<copy.rows; i++)
for(int j=0; j<copy.columns; j++)
result .array[i][j]=array[i][j]+ref.array[i][j];
return result;
}
// меншіктеу операциясын қайта жүктеу
template<class T>
 Matrix_D<T>& Matrix_D<T>: : operator = (const Matrix_D<T>&ref)
{
for(int i=0; i<rows; i++)
for(int j=0; j<columns; j++)
array[i][j]=ref.array[i][j];
rows=ref.rows; columns=ref.columns;
return*this;
}
// матрицаны басып шығару
template<classT>
void Matrix_D<T>: : PrintMatrix(IOFile &fout)
{
for(int i=0; i<rows; i++)
{for(int j=0; j<columns; j++)
fout « array[i][j] « ;
fout « endl;
}
return;
}
// идексация операциясын қайта жүктеу
template<classT>
T*& class Matrix_D<T>: : operator [ ] (int index)
{ return array[index];
}
// программаны тестеу
int main(void)     //сәттілік кезінде, main  o  қайтарады
{ IOFile  fout; //  ″ matrix.out ″ файлына жазуға арналған объект ;
fout.open_f(“matrix.out”, ios::out);
// объект массивтерді құрамыз, оларды файлдан толтырамыз және басып
шығарамыз
Matrix_D<int> a1(2,2), a2(2,2), a3(2,2);
```

```
a1.ReadMatrix( ″matrix.in  ″ ); a2 . ReadMatrix( ″matrix.in  ″);
fout « ″\na1:  ″« endl; a1. PrintMatrix( fout) ;
fout « ″\na2:  ″« endl; a2. PrintMatrix( fout) ;
fout « ″\na3:  ″« endl; a3. PrintMatrix( fout) ;
//Қайта жүктелген [] - операцияларды тестеу
fout « ″\n қайта жүктелген [] - операцияларды тестеу″
 ″\n int k=a1[1][1];”  «  endl ;
int k=a1[1][1];  fout « ″ \n k=” « k «endl;
// Қарапайым мәндерді тестеу
fout « ″\n Қарапайым мәндерді тестеу \n  ″
 ″a3=a1+a2;”«  endl;
″a3=a1+a2;
fout « ″\n a3= ”« endl; a3 PrintMatrix(fout);
// Шығару файлын жабу
fout.close_f( ″matrix.out ″);
return 0;
}
```

Бақылау сұрақтары:

1) Егер scanf ( ) - fscanf ( ), printf-fprintf-sprintf ( ) жиындары жақсы жұмыс істесе Енгізу-шығару ағындары не үшін құрылады?
2) ignore ( ) әдісін қашан қолдану керек?
3) Шығарудың қайта жүктелген операторы қандай мәнді қайтарады?
4) Бүтін сандарды шығарғанда, үнсіздік бойынша шығарудың өрісінің қандай ені қабылданады?
5) ios: : ate ашудың қарапайым режимі нені қамтамасыз етеді?

13 Дәріс №13. Ерекше жағдайларды өңдеу

Дәрістің мақсаты: ерекше жағдайларды өңдеудің есептеу үдерісін түсіндіру.

Дәрістің мазмұны: өңдеудің жалпы механизмі және ерекшеліктер синтаксисі.

Ерекше жағдай немесе ерекшелік – бұл түрлі себептердің нәтижесінде туындайтын күтпеген немесе авариялық жағдайдың, оқиғаның пайда болуы. Мысалы, бөлінетін ресурстардың жетіспеушілігінен. Ерекше жағдайларды өңдеу құрылғылары көбінесе программа жұмысы кезінде туатын қателерді өңдеу үшін қолданылады. Сондықтан программа жұмысы кезінде туатын қате ерекше жағдайдың жеке түрі болып табылады. Ерекшеліктерді өңдеу пайда болған ерекшелікті тексеру (жеке жағдайдағы қате), бағдарламаның жұмысқа қабілетін қалпына келтіру және оның орындалуын жалғастыруға мүмкіндік береді.

Ерекше жағдайларды өңдеу есептеу процесін логикалық түрде екі бөлікке бөлуге мүмкіндік береді: авариялық жағдайды табу және оны өңдеу. C++ тілінің ерекшеліктерді өңдеу құралдарының басқа артықшылығы қате туралы хабарламаны беру үшін оның өңделуінің орнына қайтарылатын мәнді, параметрлерді немесе глобальды айнымалыларды қолдануды талап етпейді. Сондықтан функциялар интерфейсі таратылмайды.

Өңдеудің жалпы механизмі және ерекшеліктер синтаксисі.

Ерекше жағдайды өңдеу үш шешу сөз арқылы жүзеге асырылады: try, catch және throw.

Қате туындай алатын, бағдарламалық кодтың фрагменті, байқап көру деп аударылатын try-қызметші сөзі алдында жазылған, құрама оператор болып табылатын, бақыланатын блокқа кіру керек .

Егер қате try блогында бар болса, онда ол қоздырылынып (throw), содан кейін ұсталынып (catch) және өңделінеді.

Блок try-дан шақырылатын функцияларда, ерекше жағдайды туғыза алады. Кез келген ерекше жағдай, catch нұсқасымен ұстап алынуы керек, ал ол, ерекше жағдайды қоздыратын try блогынан кейін орналасу керек. try блогымен catch нұсқасының бірнешеуі байланыста бола алады. Онда қандай catch нұсқасы қолданылатыны ерекше жағдайына байланысты, қалған try блогының нұсқалары ескерілмейді. Программаны құратын кластары типтермен бірге кез келген типтерді қолдануға болады. Негізінен ерекше жағдайлар ретінде, жиі түрде класс типтері қолданылады.

throw нұсқасының негізгі формасы: throw ерекше_жағдай;

Нұсқа throw try блогының ішінде, немесе осы блок шақыратын (тура немесе жанама) кез келген функцияның ішінде орындалуы керек. Мұндағы ерекше жағдай – бұл throw ерекше жағдайымен қоздырылатын нұсқау.

Егер сәйкес catch нұсқасы үшін жоқ ерекше жағдайы қондырылса, онда программаның қалыпты емес аяқталуы болуы мүмкін. Standart C ++ стандартына сәйкес өңделмейтін ереше жағдайдың қондырылуы terminate ( ) стандартты кітапханалық функцияның шақырылуына алып келеді. Үнсіздік бойынша бағдарламаны аяқтау үшін terminate функциясы abort ( ) функциясын шақырады.

Ерекше жағдайды өңдеудің жай мысалы:

```
#include<iostream.h>
int main(void)
{ cout «″ басы \n″;
try{                // try блогының басы
cout «″ try блогының ішінде \n″;
throw 10          // қатені қоздыру
cout «″ Бұл нұсқа орындалмайды! cout″;
} catch (int i)    //қатені ұстап алу
```

```
cout «″ қате номері қамтылған :  «″ i cout « endl;
}
cout «″ соңы ″;
return 0;
}
```

catch нұсқауын орындағанан кейін, бағдарламаны басқару одан кейінгі келесі нұсқау беріледі. Оған қарамастан, әдетте catch блогы exit ( ), abort( ) немесе қандай да бір функцияны шақырумен бағдарламаны мәжбүр етіп аяқтайды.

Егер осы нұсқау try блогынан шақырылатын функцияға кірсе, онда ерекше жағдай try блогына кірмейтін нұсқамен де қоздырылуы мүмкін.

try блогын функцияның ішінде орналастыруға болады. Мұндай жағдайда, функцияға әрбір енген сайын ерекше жағдай өңдеушісі қайта орнатылады.

Кейбір жағдайларда жүйені барлық ерекше жағдайлардың типіне тәуелсіз, ұстап алатындай етіп баптау керек. Мұны өте оңай істеуге болады. Ол үшін catch нұсқасының келесі формасын қолданыңыз:

```
catch ( …) { //барлық ерекше жағдайларды өңдеу }
```

Мұндағы көпнүкте мәндердің кез келген типіне сәйкес келеді.

try блогынан шақырылатын функция үшін ерекше жағдай типтерінің санын шектеуге болады, оны осындай функция қоздыра алады. Нақтысында функцияға қандай қандай да бір ерекше жағдайды қоздыруға тыйым салуға болады. Ол үшін функцияның анықтамасына throw кілттік сөзін қосу керек:

```
Нәтиже_типі функция _аты (аргументтер_тізімі) throw (типтер_ тізімі)
{…}
```

Мұнда типтер тізімі өрісінде, функция қоздыра алатын, ерекше жағдайлардың мәндер типтері үтірмен бөлініп келтірілген. Басқа типті ерекше жағдайды қоздыруды байқап көргенде, бағдарламаның авариялық түрде аяқталуына алып келеді. Қоздыратын ерекше жағдайлар типтерін шекте, функцияның try блогынан шақырғаннан кейін мүмкін болатындығын түсіну керек. Яғни, функция ішінде try блогы ерекше жағдайдың кез келген типін қоздыра алады, себебі ол осы функция ішінде ұсталынады.

Ерекше жағдайды қайта қоздыруға болады. Мұның мағынасы, әрқайсысы ерекше жағдайды өзінің аспектісінен өңдейтін бірнеше процедураға, ерекше жағдайды өңдеуге мүмкіндік беру болып табылады. Ерекше жағдай, осы блоктан шақырылатын кез келген функцияда немесе catch блогының ішінде ғана қайта қоздырылады.

Бір типті ерекше жағдайды қайта өңдеу мысалы:

```
#include<iostream.h>
void Exception ( )
{ try { //  char * типті ерекше жағдайды қоздыру
 throw  ″ сәлем  ″;
}
// char* типті ерекше жағдайды ұстап алу
catch (char*)
{ cout « ″Exception ( ) функциясының ішіндегі char*ұстап алу  \n″;
throw;   // char* типті ерекше жағдайды қайта өңдеу, бірақ енді  Exception
( ) функциясының ішінде емес.
}
}
int main(void)
{ cout «″ Басы \n″;
try{   Exception ( );
catch (char*){ cout « ″соңы main ( ) функциясының ішіндегі char*ұстап
алу\n″;}
cout «″ Соңы ″;
return 0;
}
```

Бағдарлама экранға мынаны шығарады:

```
Басы
Exception ( ) функциясының ішіндегі char*ды ұстап алу
main ( ) функциясының ішіндегі char*ды ұстап алу
Соңы.
```

New операторының қазіргі уақыттағы спецификациясына сәйкес, ол жадыдағы орын бөлу сәтсіз болған кезде ерекше жағдайды қоздырады. Жадыдан осындай сұраныс үшін, new операциясынан басқа стандартты құрылғылар, операция-функциялар operator new ( ) және operator new[ ] көмегімен енгізілетін қайта жүктелген операциялар болып табылады. Үнсіздік бойынша, егер new операциясы жадыдан қажет бөлігін бөле алмаса, онда ол нөлдік мәнді (NULL) қайтарады және қабаттас xalloc типті ерекшелікті құрады. xalloc типі - бұл except.h тақырыптық файлында анықталған класс.

Бақылау сұрақтары:

1) Ерекшелік дегеніміз не?
2) try блогы және catch операторы не үшін керек?
3) Ерекшелік қандай ақпаратты сақтайды?
4) Функциялар, глобальды айнымалылар және ерекшелік механизмінің қолдау кластары.
5) Ерекшеліктегі конструкторлар мен деструкторлар.

14 Дәріс №14. Динамикалық идентификация және типтерді келтіру

Дәрістің мақсаты: динамикалық идентификация және типтерді келтіру түсініктерін беру.

Дәрістің мазмұны: const_cast, static _cast, dynamic_cast, reinterpret_cast операциялары.

Полиморфизмді қолдайтын тілдерде компиляция кезінде объекттің типі белгісіз болу жағдайлары мүмкін, себебі бағдарламаның орындалуына дейін объекттің нақты табиғаты анықталмаған. C++ -те полиморфизм кластар иерархиясы, виртуалды функциялар мен базалық класс көрсеткіштері арқылы жүзеге асырылады. Осындай байланыс кезінде базалық класс көрсеткіші, базалық класс объектісіне сілтеуге немесе осы базалық кластан туылған кез келген туынды объектісіне сілтеу үшін қолданылады. Әрбір уақыт мезетінде, базалық класс көрсеткіші сілтеме жасай алатын объект типін алдын ала анықтау мүмкіндігі бола бермейді. Мұндай жағдайларда объект типін анықтау бағдарлама орындалуы кезінде орындалуы керек, ал ол үшін типтің динамикалық идентификациясының механизмі қызмет етеді.

Объекттің типі туралы ақпаратты typeid (объект) операторы арқылы алады, ол объекттің объектті типін сипаттайтын type-info тип объектіге сілтеме қайтарады. type-info класында келесі ашық мүшелер анықталған:

```
Bool operator == (const type_info& объект);     // типтерді салыстыру
Bool operator! = (const type_info& объект);    // типтерді салыстыру
Bool before (const type_info& объект);
Const char*name ( );                    // объект атына нұсқауыш
```

typeid операторы түрлі объект типтерін алуға мүмкіндік беруіне қарамастан, егер оның аргументі ретінде полиморфты базалық кластың көрсеткіші берілсе, онда ол анағұрлым пайдалы болады. Мұндай жағдайда, оператор автоматты түрде, көрсеткіш сілтеме жасайтын нақты объект типін қайтарады. Сөйтіп, typeid операторы көмегімен, программаманың орындалуы кезінде, базалық класс нұсқауыш сілтеме жасайтын объект типін анықтауға болады. Сонымен қатар бұл сілтемелерге де қатысты болып табылады. Егер typeid операторының аргументі ретінде, полиморфты класс объектісіне сілтеме берілсе, онда оператор, сілтемесі бар объекттің нақты типін қайтарады. typeid операторын полиморфты емес класқа қолданғанда, базалық типтің нұсқауышын немесе сілтемесін алады.

Типтің динамикалық идентификациясы әрқашан қолдана берілмейді, бірақ мәндердің полиморфты типтерімен жұмыс істегенде, ол әртүрлі жағдайларда өңделетін объектілер типтерін анықтауға мүмкіндік береді.

// typeid операторын қолдану мысалы:

```
# include<iostream.h>
```

```
# include<typeinfo.h>
class Base
{ vertual void f ( ) { };   // класты полиморфты етеміз
};
class Der1 :  public Base{…};
class Der2 :  public Base {…};
int main(void)
{ int i;
Base *p,baseob; Der1 d1; Der2 d2;
 cout «″ \n i айнымалысының типі -бұл ″  « tepeid (i). name( )  « endl;
// полиморфты типтерді өңдеу
 p=& baseob;
cout «″ p көрсеткіші тип объектісіне сілтеме жасайды ″;
cout «typeid (*p ) name ( )  «endl;
p=&d1;
cout «″p көрсеткіші тип объектісіне сілтеме жасайды ″«typeid (*p ) name
( );
p=&d2;
cout «″\n p көрсеткіші тип объектісіне сілтеме жасайды ″ «typeid (*p )
name ();
return 0 ;
}
```

Бағдарлама экранға келесі ақпаратты шығарады:

```
i айнымалысының типі  - int
p нұсқауышы Base типті объектіге сілтеме жасайды
p нұсқауышы Der1 типті объектіге  сілтеме жасайды ″;
p нұсқауышы Der2  типті объектіге  сілтеме жасайды ″;
```

Объекттер функцияларға сілтеме бойынша берілетін жағдайда typeid операторының нақты қасиеті объекттің нақты типін қайтарады, мысалы:

```
oid func ( Base & ref)
{ cout «″ ref –сілтеме объектіге мына түрдегі «typeid (ret  ). name ( ) ;}
```

d1, d2 объектілері үшін шақыру кезінде олардың дұрыс идентификациясы орындалады:

func (d1) және func (d2) сәйкесінше.

typeid операторының басты артықшылығы компиляция кезінде белгісіз объекттер типтерін идентификациялау мүмкіндігі болып табылады.

Экранда түрлі геометриялық фигураларды салу үшін арналған кластар иерархиясы анықтайтын бағдарлама мысалын қарастырайық. Иерархия басында төрт класты мұрагерленетін Shape абстрактты класы орналасқан: Line, Square, Rectangle және NullShape. Функция generator( ) объектіні туғызады және оған көрсеткішті қайтарады. Онда қандай объект құрылатынын, және rand ( ) - кездейсоқ сандық генераторы анықтайды. main ( ) функциясында, NullShape типті объекттерден басқа, экранда түрлі типті туындалған объектілер қортындысы жүзеге асырылған. Объекттер кездейсоқ туатындықтан, алдын ала, келесі қандай объект құрылатыны белгілі болмайды. Нәтижесінде, құрылатын объектінің типін анықтау үшін типтің динамикалық идентификациясы талап етіледі.

```
# include<iostream.h>
# include<cstdlib.h>
# include<typeinfo.h>
class  Shape
{ public : virtual void example( )=0;};
class  Rectangle:public Shape
{public:void example( ){cout«″ Тікбұрыш\n″;}  };
class Triangle :public  Shape
{ public:void example ( ) { cout«″ \n Үшбұрыш \n ″;} };
class line : public Shape
{ public: void example( ) {cout «″\n  Сызық\n″; } };
class NullShape
{ public :void example( ){ } };
// Shape класынан туындылар  фабрикасы
Shape *generator ( )
{ switch (rand ( ) %4)
{ case o: return new Line ;  case 1: return new Rectangle;
case 2:return new Triangle; case3:return new NullShape;
}
return Null;
} int main (void)
{ int i; Shape *p;
for (i=0;i <10;i++)
{ p=generator( );    // келесі объектіні құру
cout «typeid (*p ) name «endl
// объектіні салады, егер ол NullShape типті емес болса
if (typeid (*p )! =typeid (NullShape)) p--example( );
} return 0;
}
```

Typeid операторы үлгі-кластармен жұмыс істей алады. C++ тілінде жазылған бағдарламаны орындағанда типтер түрлері орындалады: нақты

және нақты емес. Типтердің нақты түрленулері әрқашан программист қалауымен орындалады және ол үшін C++ тілінде мыналар бар:

- Си тілінен мұрагерленген типті келтіру операциясы;
- C++ тіліне енгізілген функционалды жазба формасы;
- const_ cast, dynamic_cast, reinterpret_cast және static_cast операциялары.

Соңғы операциялар типтердің түрленулерінің анағұрлым сенімді операциялары болып табылады, және оларды барлық қажет жағдайларда қолданған жөн.

const_cast операциясы. Берілген операция const модификаторының әрекеттерін нөлге айналдыру үшін қолданылады:

```
const int i;
int *p=const_cast<int*>(&i);    // жай көрсеткішті аламыз
```

Жалпы жағдайда бұл операция келесі форматқа ие болады:

```
const_cast <T> (V),
```

Мұндағы T типі V типі (бұл әдетегі көрсеткіш) сияқты болу керек. Операция T типті нәтижені қайтарады. Бұл операцияның қажеттілігі функцияны жобалағанда const сияқты өзгермейтін параметрлерді сипаттау міндетті емес болумен шатастырылған, бірақ мұны жасау ұсынылады. C++ ережелері константты көрсеткішті жай көрсеткіш орнына беруге рұқсат етпейді. Сондықтан const_cast операциясы осы шектеуді айналып өту үшін енгізілген. Әрине, мұндай жағдайда функция берілетін көрсеткіш сілтеме жасайтын мәнді өзгертпеу керек.

Компиляция кезінде (тақырып, ортаға жаз) типтердің түрленуі (static _cast операциясы) static _cast операциясы компиляторымен орындалады және онсыз жұмыс істеуге болмайтын кезде ғана қолданылады. static_cast операциясын қолданудың мысалын көрсеткіштің қажет типіне void* көрсеткішін түрлендіру деп атауға болады, бүтін типін есептеп шығу типіне келтіру және туыс типтердің басқа да түрленуі деп те атауға болады.

Программаны орындау кезіндегі типтердің түрленуі (dynamic_cast) dynamic_ cast операциясы әдетте полиморфты объектілер үшін және виртуалды базалық кластардың түрленуін төмендету үшін қолданады. Бұл кезде, операция программаны орындау кезінде типтерді идентификациялау (Run-Time Type Information, RTTI) механизмін қолданады.

Бақылау сұрақтары:

1) typeid операторы көмегімен орындауға болатын опрацияларды сипаттаңыз.
2) Қандай объекттің типін typeid операторымен анықтайды?
3) const_cast типті түрлендіру операциясы не үшін керек?

4) dynamic_ cast типіті түрлендіру операциясы не үшін керек?
5) static _cast және reinterpret_cast операциялары не үшін керек?

15 Дәріс №15. C++ тілінің стандартты кітапханасы. Жалпыланған программалау

Дәрістің мақсаты: C++ тілінің стандартты кітапханасы кластарының жұмысын ұғындыру.

Дәрістің мазмұны: контейнерлі кластар. Итераторлар мен функциональды объектілер.

Алгоритмдер.

C++ тілінің стандартизация үдерісі 1989 жылы басталып және 1998 жылы тіл стандартын басып шығарумен аяқтады.(Informating Technology-Programming Language-C++, нөмірі ISO/ IEC 14882-1998).

C++ тілінің ережелері мен синтаксисінің қатаң , құрылған анықтамалары тілге үйретуді, бағдарламаны жазуды, сонымен бірге олардың басқа платформаларға бейімделуін жеңілдетеді.

C++ тілінің стандартты кластарын былай бөлуге болады: ағынды кластар, жолдық кластар, контейнерлі кластар, алгоритмдер және итераторлар, математикалық, диагностикалық және қалған кластар.

Контейнерлі кластардан, алгоритмдерден және итераторлардан құралған стандартты кітапхана бөлігін үлгілердің стандартты кітапханасы деп атайды. (Standard Temlate Library, STL). Кітапханаға қазіргі және тиімді алгоритмдерді қолданып объекттер жиынымен (коллекциялармен) жұмыс істеуге арналған белгіленген құралдар кіреді. C++ тілінің стандартты кітапханасымен оған негізделген жалпыланған технологиясын қолдану, программистерге жұмыс принциптерін қарастырмай, мәндер мен алгоритмдер құрылымдары аймағында жаңа өндірулерді қолдануға мүмкіндік беріледі.

Контейнерлі кластар.

Контейнер – бұл белгілі бір түрде ұйымдастырылған басқа объектілер жиыны құрайтын объект. Контейнерлер анық типтің объект коллекцияларын басқаруға арналған. Контейнерлер мысалдары массивтер (векторлар және ассоциативті массивтер) және тізімдер (жекеленген тізімдер, кезектер, стектер) болып табылады. Контейнерге объектілерді қосуға болады және оларды одан жоюға болады. Контейнермен жұмыс, стандартты кітапханадағы контейнерлі кластар көмегімен қолдау табады. Бұл мүмкіндік класс үлгілері арқылы жүргізіледі. Контейнерлерді қолдану программалар сенімділігін жоғарлатуға, олардың тасымалдануымен бір мезгілдегі уақыттың кемуі мен өндіру құны бар әмбебаптылыққа мүмкіндік береді.

STL контейнерлі класы кезекті және ассоциативті деп бөлінеді. Кезекті контейнерлер үздіксіз кезектілігі түріндегі бір типті объектілердің соңғы санын анықтауда қамтамасыз етеді және келесі түрлері бар:

- векторлар (vector);
- екі жақты кезектер немесе басқаша айтқанда деректер (deque);
- тізімдер (list);
- стегтер (stack);
- кезектер (quere);
- п риоритеттері бар кезектер (priority-quere).

Ассоциотивті кезектер сұрыптаудың белгілі критериі бойынша оның мәніне тәуелді объект(элемент) позициясы бар сұрыпталған коллекцияларды береді. Ассоциативті контейнерлер кілт бойынша деректермен тез қатынауды қамтамасыз етеді және балансталған ағаштар негізінде құрылған. Ассоциативті контейнерлердің келесі түрлері бар:

- сөздіктер (map);
- көшірмелері бар сөздіктер (multimap);
- жиындар (set);
- көшірмелері бар жиындар (multiset);
- биттік жиындар (bitset).

Түрлі контейнерлі кластардың ортақ қасиеттері көп және бұл өте қолайлы. Контейнерлі кластардың стандартталған интерфейсі бар. Бұл түрлі контейнерлі кластардағы бір атты деректер мүшелер мен операциялардың мағынасы бірдей екендігін білдіреді және контейнерлердің барлық типтеріне қолданылады.

Итераторлар мен функциональды объектілер

Итераторлар ұйымдастыру әдістері мен әрекеттер типіне тәуелсіз деректердің әрбір типіне қатынау және кезекті қарастыру құрылғылары болып табылады. Сөйтіп, итератор жалпыланған итератор болып табылады, және итератор мен көрсеткіш семантикасы бірдей STL –да итераторлар контейнерлі кластар, ағындар мен буферлі ағындармен жұмыс істеу үшін қолданылады.

Деректер түрлі түрде ұйымдастырылуы мүмкін (массив, тізім, ағаш және тағы басқа) әрбір кезектілік түрі үшін белгілі операциялар жиынын қолдайтын итератордың өз типі талап етіледі (кіріс, шығыс,тура, екі бағытты, және кезекті қатынау). Итераторлы адаптерлер бар, мыналарды қолдайтын алгоритмдерді орындауды қамтамасыз ететін арнайы итераторлар:

- элементтері кері тәртіпте мөлшерден тыс жинау (кері итераторлар);
- енгізу режимі (енгізу итераторлары);
- деректер ағынымен жұмыс (ағынды итераторлар).

Функционалды объект деп функцияны шақыру операциясы орындалған класс типі бар объектті атайды. Жиі түрде функционалды объекттер өңдеу әдістері мен оларды салыстырудың қолданушы контейнерлерін беруге арналған стандартты алгогритмдер параметрлері ретінде қолданылады. Параметр ретінде функционалды объектілер қолданылатын алгоритмдерде функцияға нұсқауышты қолдануға болады. Bool мәнін қайтаратын

функциональды объектілер бар және предикаттар деп аталады. Предикат деп bool мәнін қайтаратын жай функцияны айтады.

Контейнерлердің өзі үлкен мәндерге иеленбейді. Пайдалы болу үлкен контейнер негізгі операциялармен жабдықталуы керек. Стандартты кітапхана контейнер қолданушысынан талап етілетін көптеген кең таралған операциялардың орындалуына арналған алгоритмдерді ұсынады. Осындай алгоритмдер саны алпыс шамасында.

Әрбір алгоритм функция үлгісі түрінде немесе функциялардың үлгілерінің жиыны түрінде жүзеге асырылған. Соның арқасында алгоритм кезекшіліктің әр түрімен жұмыс істей алады және түрлі типтердің деректерімен (жалпыланған пролграммалаудың технологиясының концепциясы). Қолданушының нақты талаптарының алгоритмін баптау үшін функционалды объекттер қолданылады. Стандартты кітапхананың басқа құрылғылары сияқты стандартты алгоритмдерін қолдану программисті кезекшілікті өңдеу циклдерін құжаттау, түзету, жазудан айырады. Бұл бағдарламадағы қателерді кемітеді, өндіру уақытын қысқартады және анағұрлым оқылатын және жинақты етеді.

STL барлық алгоритмдерді бес категорияға бөледі:

- кезекшілігі бар модиицирленбейтін операциялар;
- кезекшілігі бар модиицирленетін операциялар;
- кезекшілікті сұрыптау алгоритмдері;
- көпшілікпен пирамидалармен жұмыс істеу алгоритмдері;
- жалпыланған сандық алгоритмдер.

Параметрлер ретінде алгоритмдерге өңделетін кезектіліктің басы мен соңын анықтайтын итераторлар беріледі. Итераторлар түрлері берілген алгоритм қолданылуы мүмкін контейнерлер типін анықтайды. Мысалы: сұрыптау алгоритмі sort ( ) еркін қатынау итераторларын қолдануды талап етеді. Сондықтан ол list контейнерлерімен жұмыс істей алмайды. Алгоритмдер кезекшілік шегінен шығуды тексермейді.

Бақылау сұрақтары:

1) Жалпыланған бағдарламалаудың технологиясының концепциясы?
2) Контейнерлер дегеніміз не, олар не үшін керек?
3) Итераторлар не үшін арналған?
4) Қандай операциялар итератордың кез келген типі үшін мүмкін болып табылады?
5) STL алгоритмдерінің негізгі категорияларын атап өтіңіз.

Әдебиеттер тізімі

1 Крячков А.В., Сухинин И.В., Томшин В.К. Программирование на С и С++. Практикум: Учеб.пособие для вузов.- М.: Горячая линия-Телеком, 2006.

2 Кнут Д. Искусство программирования для ЭВМ: в 3т. Т.3 Сортировка и поиск. -М.: Мир, 2012.

3 Дейтел Х.М. Как программировать на С++ . -М.: Бином, 2011.

4 Культин Н. Основы программирования Microsoft в Visual C++2010.- СПб.: БХВ- Петербург, 2010.

5 Лафоре Р. ОПП в С++ . -СПб.: Питер, 2011.

6 Павловская Т.А.С/С++. Структурное и ООП. Практикум.- СПб.: Питер, 2011.

7 Федоренко Ю. Алгоритмы и программы на С++ BUILDER .-М.: ДМК Пресс, 2010.

8 Пахомов Б.С/С++ и MS Visual C++ 2010 для начинающихся + CD. -СПб.: БХВ- Петербург, 2011.

8 Культин Н. С++ Builder в примерах и задачах, Санкт-Петербург, 2007

9 Литвиненко Н.А. Технология программирования на С++ . Начальный курс. –СПб.:БХВ- Петербург, 2005.

АЛМАТЫ ЭНЕРГЕТИКА ЖӘНЕ БАЙЛАНЫС УНИВЕРСИТЕТІНІҢ
КОММЕРЦИЯЛЫҚ ЕМЕС АКЦИОНЕРЛІК ҚОҒАМЫ
Ақпараттық жүйелер кафедрасы

БЕКІТЕМІН
ОӘЖ проректоры
_____ С.В. Коньшин
«_____» _____ 2016 ж

ОБЪЕКТІГЕ БАҒЫТТАЛҒАН ПРОГРАММАЛАУ

5В070300 - Ақпараттық жүйелер, 5В060200 – Информатика мамандықтарының студенттеріне арналған

КЕЛІСІЛДІ
ОӘБ бастығының м.а.
_____Р.Р.Мухамеджанова
«____» _____2016 ж.

АЖ кафедрасының _____ жылғы отырысында қарастырылып, мақұлданды, хаттама №___.
Келісілді
АЖ кафедрасының меңгерушісі
_____ Ш.И. Имангалиев

ОӘКБ төрағасы
_____ Б.К. Курпенов
«____» _____ 2016 ж

Құрастырушы:
_____ А.С.Сәрсенбай

Стандарттау маманы
_____
«____» _____ 2016 ж.

Редактор
_____
«____» _____ 2016 ж.

2016 ж. баспа жоспары, реті 257

Айсұлтан Сәдуақасұлы Сәрсенбай

ОБЪЕКТІГЕ БАҒЫТТАЛҒАН ПРОГРАММАЛАУ

5В070300 - Ақпараттық жүйелер, 5В060200 – Информатика мамандықтарының студенттеріне арналған

Редактор Қ.С.Телғожаева
Стандарттау бойынша маман Н.Қ. Молдабекова

Басуға_____ қол қойылды
Таралымы 30 дана.
Көлемі 3,5 есептік-баспа табақ

Пішімі 60х84 1/16
Баспаханалық қағаз №1
Тапсырыс___ Бағасы1750 тг

«Алматы энергетика және байланыс университеті»
коммерциялық емес акционерлік қоғамының
көшірмелі-көбейткіш бюросы
050013, Алматы, Байтұрсынұлы көшесі, 126