КОНСПЕКТ

по Обектно ориентирано програмиране   
за специалност Информатика, II ри курс, I ви поток

1. Структурно и модулно програмиране.
2. Същност на обектно ориентираното програмиране.
3. Класове и обекти. Примерна програма.
4. Област на действие клас и достъп до елементите на класа.
5. Интерфейс и реализация.
6. Конструктори и деструктори.
7. Използване на данни-елементи и функции-елементи.
8. Константни обекти и константни елементи.
9. Композиция - класове като елементи на други класове (влагане на класове).
10. Приятелски функции и приятелски класове.
11. Използване на указателя **this**.
12. Статични елементи на клас.
13. Предефиниране на операции.
14. Пример за клас масив.
15. Пример за клас низ.
16. Пример за клас дата.
17. Наследяване. Базови и производни класове.
18. Предефиниране на функции-елементи от базовия клас в производния клас.
19. Използване на конструктори и деструктори в производни класове.
20. Множествено наследяване.
21. Виртуални функции и полиморфизъм.
22. Абстрактни базови класове и конкретни класове.
23. Пример за използване на виртуални функции и полиморфизъм.
24. Пример за наследяване на интерфейса и неговата реализация.
25. Вход и изход на потоци.
26. Манипулатори на потоци. Флагове за състоянията на формáтa.

**№ 1 Структурно и модулно програмиране**

Един от първите езици от високо ниво е **FORTRAN**, създаден през 1958г. в **IBM** – тогава основен производител на компютри. Той е предназначен за програмиране на алгоритми за решаване на компютърни задачи по формули. Една година по-късно се създава езика **COBOL** **(COMMON** **BUSINESS-ORIENTED LANGUAGE)** за икономически задачи. Година по-късно се създава развитие на **FORTRAN** - езика **ALGOL–60**, но той не получава практическо разпространение. **IBM** не е продавала компютри без трaнслаторите **FORTRAN** и **COBOL**. След това в **IBM**, следвайки тенденцията към универсализация, обединяват три езика и създават езика **PL/1(PROGRAMNG LANGUAGE-1)**. Практиката на програмирането е показала, че когато се съставят програми, най-трудното е откриването на грешки. Една от основните причини за това е, че програмите имат объркана логика, заради прекомерното използване на оператора **goto**. Пръв Дейкстра защитава тезата срещу оператора **goto** и се счита, че той полага принципите на структурното програмиране.

**Структурно програмиране** - подход за създаване на ясни, добре документирани и структурирани програми, които могат да се модифицират лесно. Изхвърлянето на **goto** от програмите води до линейна структура на програмата.

При структурното програмиране има три структури за управление:

* Последователно изпълнение (например в **C++** операторите се изпълняват последователно)
* Селектиране ( в **C++** операторите **if** и **switch** )
* Повторение, изключва се **goto** (в **С + +** основно средство е оператор **while**)

Като математическа теорема е доказано, че с помощта на тези три структури може да се реализира всекиалгоритъм. Когато се появява структурното програмиране има много дискусии. Те не се базират само на спора за **goto**, но и за ясни и модифицируеми програми, т.е. говори се за принципи и методи на структурното програмиране.

Принципи на структурното програмиране:

* принцип на визуалност и мнемоника (краснопис в програмирането);
* принцип за програмиране без **goto**;
* принцип за стандартизация на управлението при създаването на една програма;
* програмата да се проектира предварително, преди да се кодира;
* програмата да съдържа коментари;
* разбиване на една програма на подпрограми – стъпка по стъпка да се детайлизира алгоритъма;

**мнемоничност** – имената на идентификаторите да са свързани с тяхното използване.

**визуалност** – как по нагледно да се записват операторите(отместването им).

В процеса на програмиране трябва да има възможност за проверка на програмата, като е добре тя да се провери преди, а не след стартирането на програмата.

Принципът на премълчаване, който е развит в **PL/1**, е в противоречие с линейното програмиране. **PL / 1** не успява да замести **FORTRAN** и **COBOL**.

Метод за програмиране отгоре надолу. Решаването на задачата се свежда до решаването на няколко подзадачи.

Извършва се последователна детайлизация на алгоритъма. Когато всички подзадачи са описани на **С++**, тогава сме стигнали последното ниво на задачата и сме завършили програмата. Словесното описание в програмата остава същото. Първо се пишат коментарите, а след това текста на програмата. Можем да следим грешките на първите етапи, още при преходите от едно ниво към друго. Едно такова разбиване на задачата предполага въвеждането на функции(подпрограми). Този метод дава възможност задачата да се решава от колектив.

Подхода на структурното програмиране все още е нефективен при по-големи програми. Усъвършенства се използването на подпрограми – в структурното програмиране не са дефинирани ясно връзките между отделните подпрограми. В резултат на развитието на подпрограмите се заражда идеята за модул.

**ALGOL-60** на практика не се използва, но е развит с идеите на структурното програмиране и като негов наследник се появава езика **PASCAL**. Авторът на **PASCAL**, Никлаус Уирт, се заема с понятието модул и въвеждане на модулното програмиране. Така, като продължение на **PASCAL**, той създава **MODULA-2** – език за модулно програмиране. Програма на **MODULA-2** се състои от модули. При него явно се описва какви ресурси той може да обменя с други модули. Модула има обвивка, която пропуска само онези програмни ресурси които служат за обмен между модули – константи, променливи, типове и процедури. Един модул се състои от две части: едната част се нарича модул **definition**, който съдържа списък от данни, а другата част модул **implementation**, който съдържа подпрограми. Обмяната на ресурси става, чрез специален механизъм за import-export. Чрез този механизъм модулите експортират свои ресурси към програмната среда, а модулите клиенти ги импортират. Явно се показва какви ресурси се експортират и импортират. Модули има и в **ADA**, но там те се наричат пакети. Модул се появява и в **PASCAL**. Единиците „unit” в **PASCAL** са модули. В **С** и **С++** модулът е програмна единица, която се съхранява в отделен първичен файл. Въвеждането на понятието модул улеснява писането на още по-големи програми.

**№ 2 Същност на ООП**

При програмирането отгоре надолу (**top-down**) вниманието е съсредоточено върху създаване текста на програмата, вместо върху структурата й и нейното взаимодействие. Програмата е много силно обвързана с поставената задача. Това я прави по-трудно модифирицируема.

При **ООП** се акцентира върху разглеждането на изпълнението на програмата като моделиране на поведението на някакво явление. Ако по структура една програма е близка до явлението за което се отнася, се счита , че тя е обектно ориентирана. **ООП** може да се разглежда като развитие на модулното програмиране. Всеки модел се състои от обекти и действия. Обектите се характеризират с определени качества, наречени атрибути. Изучавайки реалното явление, програмиста трябва да се абстрахира от несъществените му характеристики и да отдели най-важното свеждайки го до абстрактна структура от данни. Познавайки взаимодействията на обектите на модела, програмиста трябва да въведе изпълним програмен код, който реализира необходимите операции върху абстрактните структури върху които е изграден модула. Новото е, че структурата данни около която се създава модула е извлечена от естествените характеристики на моделирания обект. **ООП** капсулира данни (атрибути) и функции (варианти на поведение) в целокупност наречена **обект**. Данните и функциите са тясно свързани помежду си. Обектите притежават свойство да скриват информация. Това означава, че макар и обектите да знаят как да се свързват един с друг чрез добре определен интерфейс, те не знаят как са реализирани другите обекти. Детайлите на релизиране са скрити в самите обекти. При структурното програмиране на **C** и **С++** , основната програмна единица е функцията. При обектно-ориентираното програмиране основната програмна единица е класът, въз основа на който се създават обекти. **ООП** езиците по-добре представят семаниката на един сложен обект и капсуловат информацията в него.

Според Бярне Строуструп, абстрактни типове данни са типове дефинирани от потребителя. Не трябва да се интересуваме от вътрешното им представяне. В **С++** основна програмна единица е **класа** – тип дефиниран от потребителя. Чрез класовете се дефинират абстракни типове данни. Обектите са екземпляри от определен клас.

Друго понятие е **полиморфизъм** - възможността по еднообразен начин да се обработват различни типове обекти. За **С++** това е възможността обекти от различни класове, свързани с помощта на наследяване, да реагират по различен начин при обръщение към една и съща функция елемент.

В **ООП** се говори за интерфейсите на клас, това обикновенно са функции за **С++**. Отделните функции се наричат методи. Обръщението към функция е изпращане на съобщение до някакъв обект със заявка да се изпълни определен метод.

Езика **С++** е създаден в началото на 80г. от Бярне Строуструп (Bjarne Stroustrup*)*.

**№ 3 Класове и обекти. Примерна програма**

**Класовете** са типове, дефинирани от потребителя. Всеки клас съдържа данни и набор от функции, обработващи тези данни. Данните, които са компоненти на класа, се наричат **данни-елементи** (**член-променливи, член-данни, members**). Функциите, които са компоненти на класа се наричат **функции-елементи** или **член-функции**. Подобно на това, както същност от вграден тип **int** се нарича променлива, то същност от потребителски дефиниран тип(клас) се нарича **обект**. Понятието клас e естествено развитие на понятието структура. Класовете дават възможност да се моделират явления, които имат атрибути, представени като данни-елементи, и варианти на поведение, представени като функции-елементи. Понякога член-функциите се наричат **методи**. Те се извикват в отговор на съобщения изпратени към обекта. Съобщението съответства на извикването на функция-елемент.

Синтаксис на декларацията на клас:

class име\_на\_клас

{

тяло\_на\_клас;

};

След като класът е дефиниран, името му може да се използва за дефиниране на обекти от този клас.

**Пример:** class Time{

public:

Time( );

void setTime(int, int, int);

void printMilitary( );

void printStandard( );

private:

int hour; //0÷23

int minute; //0÷59

int second; //0÷59

};

Етикетите **public** и **private** се наричат **спецификатори за достъп** до елементите. Данните-елементи и функциите-елементи декларирани след **public** до следващия спецификатор за достъп се наричат **открити** и те са достъпни при всяко обръщение от програмата към обект от класа. Данните-елементи и функциите-елементи декларирани след **private** до следващия спецификатор за достъп се наричат **закрити** и те са достъпни само за функциите-елементи от този клас. Спецификаторите за достъп завършват с **':'**. Спецификаторите за достъп могат да се използват в декларацията на класа многократно в произволен ред. Препоръка – да се използват еднократно, като се използва първо **public**. В примера функциите-елементи са открити. Те представляват открит интерфейс за услугите които предоставя класа. Тези функции ще се използват от клиентите на класа за обработка на данни от този клас.

Има функция-елемент със същото име, като името на класа. Тази функция се нарича **конструктор** за този клас. Конструкторът е специална функция-елемент, която инициализира данните-елементи при конструирането на обект от този клас. Конструкторът на класа се извиква автоматично при създаване на обект от този клас. Възможно е класът да има няколко конструктора. Това се постига чрез предефиниране на функции - **overload**.

След **private** има 3 целочисленни данни-елементи (hour, minute, second). Те са достъпни само от функциите-елементи на класа. Обикновенно данните-елементи са в частта **private**, а фунциите-елементи в **public**. Възможно е да има функции-елементи в **private** и данни-елементи в **public**, но не се препоръчва(смята се за лош стил на програмиране). След като класът е дефиниран, той може да се използва като тип при дефинирането на обекти от този клас.

Time s,a[5];//s - обект от тип Тime, a - масив от 5 елемента oт клас Time

Time \*p,&d = s;//d - псевдоним на обектa s от клас Time

Дефинирането на класове(потребителски типове) е едно от средствата за разширяемост на **С ++**.

// Първичен файл PRIM\_PROG.cpp – примерна програма на С++

#include <iostream> //файл за вход/изход

using namespace std;

class Time{

public:

Time();/\* ще се присвоят нулеви начални стойности на всеки елемент от   
 данните,съгласно началното състояние на всеки обект от тип Timе \*/

void setTime(int,int,int);

void printMilitary(); //отпечатване на времето във военен формат

void printStandard(); //отпечатване на времето в стандaртен формат

private:

int hour;

int minute;

int second;

};

Time**::**Time()

{ hour = minute = second = 0;}

**::** - знак за бинарна операция за разрешаване на област на действие. Тъй като различни класове могат да имат елементи с еднакви имена, операцията за разрешаване на област на действие **::** привързва името на елемента към името на съответния клас, за да се идентифицира еднозначно функцията-елемент на даден клас. Функция-елемент, чийто прототип се намира в декларация на клас, може да бъде описана извън декларацията на класа, като в същото време тя има областта на действие на класа и следователно името ú е известно само на другите елементи на класа. Ако функция- елемент е описана в декларация на клас, тя автоматично се разглежда като вградена, т.е.**inline**. Когато функция-елемент е описана извън декларацията на класа, тя може да се направи вградена чрез използване на ключовата дума **inline**. Ще отбележим, че компилаторът може да не вгражда никакви функции, т.е. да не изпълни операцията inline и да не вгради функцията. Описанието на неголеми функции-елементи в декларацията на класа подобрява производителността, ако компилаторът ги вгражда, но не подобрява качеството на проектиране на програмата.

**cout** е стандартен обект, който е свързан със стандартния поток за извеждане, който най-често операционната система свързва с екрана. Когато **cout** извежда стойност на израз трябва да се поставят скоби, защото ако не се поставят ще се разглежда двуместната операция отместване вляво.

void Time::printMilitary()

{

cout<<(hour<10 ? "0":"")<<hour<<":"

<<(minute<10 ? "0":"")<<minute<<":"

<<(second<10 ? "0":"")<< second;

}

void main()

{

Time t; /\* при това дефиниране се създава обект t, при което автоматично се извиква   
 конструктора на класа Time. Този конструктор явно присвоява нулеви начални   
 стойности на всички данни-елементи от закритата част private: \*/

cout<<"Началната стойност във военен формат е ";

t.printMilitary(); // 00:00:00

/\* endl - манипулатор на cout, който извежда символа за минаване на нов ред и след това изчиства буфера за извеждане т.е. предизвиква незабавно извеждане на екрана на всички натрупани в буфера данни. В случая ще мине на нов ред и след това ще изведе "Началната стойност в стандартен формат е " (без кавичките). Mожем вместо endl да изведем ‘\n’ \*/

cout<<endl<<"Началната стойност в стандартен формат е ";

t.printStandard();// 12:00:00 AM

t.setTime(13, 27, 6);

cout<<endl<<endl<<"Военното време след setTime e ";

t.printMilitary();// 13:27:06

cout<<endl<<"Стандартното време след setTime e ";

t.printStandard();// 1:27:06 PM

t.setTime(99, 99, 99);

cout<<endl<<endl<<"След задаване на неправилни стйности: "

<<endl<<"Военното време е ";

t.printMilitary();// 00:00:00

cout<<endl<< "Стандартното време e ";

t.printStandard();// 12:00:00 AM

cout<<endl;

}

Функциите-елементи обикновено са по-къси в сравнение с обикновените функции при процедурното програмиране. Основната причина е, че достоверността на данните в **ООП**, които се съхраняват в данните-елементи е проверена от конструктора и функциите-елементи които задават нови данни. Данните-елементи на класа не могат да получат начални стойности в декларацията на класа, където са декларирани. Тези данни-елементи трябва да бъдат инициализирани, чрез конструктора или чрез други функции-елементи на класа. Обикновено функциите-елементи имат по-малко параметри, тъй като могат директно да използват данните-елементе на съответния клас. Това намалява вероятността за грешки при обръщенията към тях.

Функция със същото име, като името на класа започваща с **~** се нарича **деструктор** на този клас. Деструкторът изпълнява завършващи служебни действия над обект на класа преди паметта разпределена за този обект да бъде повторно използвана от системата.

Откритите функции-елементи реализират възможностите на класа, необходими на неговите клиенти. Те се наричат **интерфейс** на класа. Декларирането на функциите-елементи в тялото на класа и тяхното описание извън него, отделят интерфейса на класа от неговата реализация. Това подпомага разработването на качествено програмно осигуряване. Клиентите на класа го използват без да знаят вътрешните детайли на неговата реализация. Вътрешното представяне на данните на класа не засяга клиентите и в този случай се говори за скриване на информация от клиента. Ако реализацията на класа се измени, например с цел повишаване на производителността, интерфейса на класа остава непроменен и текстът на програмата на клиента не се променя. Това значително опростява модификацията на програмите. Тази тенденция се задълбочава с използването на скрити данни-елементи.

**№ 4 Област на действие клас и достъп до елементите на класа**

Данните-елементи на класа, т.е. променливите декларирани в декларацията на класа и функциите-елементи, т.е. функциите декларирани в декларацията на класа имат **област на действие** клас. Функциите, които не са елементи на клас имат област на действие файл. В областта на действие клас, елементите на класа са непосредствено достъпни за всички функции-елементи и данни-елементи на този клас и могат да се използват просто по име. Извън областта на действие клас, елементите на класа, които са открити, могат да се използват или с помощта на името на обекта, или чрез псевдоним на обекта, или чрез указател към обекта. Функциите-елементи на класа могат да се предефинират, но само с помощта на други функции-елементи на този клас. Типична грешка е опитът да се предефинира функция-елемент на класа с помощта на функция, която не е от областта на действие на този клас. Компонентите на функциите-елементи на класа имат област на действие функция – например променливите, които са дефинирани в тялото на функция-елемент са известни само на тази функция. Ако във функция-елемент се дефинира променлива с име, което съвпада с името на променлива в областта на действие клас, то последната променлива е невидима в тази функция-елемент. Такива скрити променливи могат да станат достъпни като се използва бинарната операция за разрешаване областта на действие ‘**::**’с ляв операнд името на класа и десен операнд името на променливата. Скритите външни променливи могат да станат достъпни с помощта на унарната операция за разрешаване на област на действие **::** .

float value = 1.2345;

void main()

{

int value = 7;

cout<<value<<endl // value – това е локалната променлива

<<**::**value<<endl;// ::value - това е глобалната променлива

};

Изход: 7

1.2345

Операциите, които се използват за достъп до елементите на клас са аналогични на операциите които се използват за достъп до елементите на структура.

Операцията **.** (точка) за достъп до елемент на обект има вида :

име\_на\_обект или псевдоним\_на\_обект**.**име\_на\_елемент\_на\_обект

Операцията → за достъп до елемент на обект има вида : указател\_към\_обект→име\_на\_елемент\_на\_обект

**Пример:**#include <iostream>

using namespace std;

class Count{

public:

int x;

void print(){cout<<x<<endl;}

};

void main()

{

Count counter,

\*counterPtr = &counter,// counterPtr - указател към обект

&counterRef = counter; // counterRef - псевдоним на обект

counter.x = 7;

counter.print();

counterRef.x = 8;

counterRef.print();

counterPtr->x = 10;

counterPtr->print();

};

Спецификаторите за достъп до елемент **public**, **private** и **protected** се използват за управление на достъпа до данните-елементи и функциите-елементи на класа. По премълчаване спецификатора за достъп е **ptivate** т.е. всички елементи след заглавието на класа до първия спецификатор за достъп са закрити. Закритите елементи на класа са достъпни само за функциите-елементи и приятелските функции на този клас. Откритите елементи на класа са достъпни за всички функции в програмата. Закритите елементи на класа и описанията на откритите функции-елементи са недостъпни за клиентите на класа и представляват реална защита на класа. Типична грешка е да се направи опит за достъп до закрит елемент на даден клас във функция, която не е елемент на класа и не е приятелска функция на класа.

void main()

{

Time t;

t**.**hour = 7; // Компилаторът извежда съобщение за грешка !

cout<<t**.**minute; // Компилаторът извежда съобщение за грешка !

};

При компилирането на програмата ще се получи съобщение за грешка, защото елементите hour и minute са закрити.

В декларацията на класа се препоръчва най-напред да се използва спецификатор за достъп **public**, защото това концентрира вниманието на клиента върху интерфейса, а не върху реализацията на класа. Добър стил на програмиране е използването на закрити данни-елементи и открити функции-елементи за задаване и получаване на стойностите на закритите данни-елементи. Клиент на класа може да бъде функция-елемент на друг клас или пък външна функция.

Структурите в **C++** са разширени с възможности за задаване на режим на достъп до елементите и с възможности за включване на функции-елементи. Достъпа по премълчаване в структурите е **public**, а в класовете е **private**. В **C++** структурите и класовете са взаимнозаменяеми. Създателите на класове използват режимите на достъп за скриване на информация и поради принципа на най-малките привилегии. Достъпа до закритите елементи на класа се извършва от функции-елементи на класа или функции приятели на този клас, които се наричат **функции за достъп**. Най често се използва функция за достъп с име **get,** която е за прочитане на стойностите на закритите данни или функция за достъп с име **set** за задаване и изменение на закрити данни. Ако е необходимо тези две функции биха могли да извършват проверка за коректността на данните. Те, също така, могат да преобразуват данните от вида, използван в интефейса, във вида, използван при реализацията. Друго типично използване на функциите за достъп е проверката на определено условие – такива функции се наричат **предикатни функции.** Пример за предикатна функция е функцията **isEmpty** за всеки клас-контейнер, т.е. клас, който може да съдържа много елементи, например свързан списък, стек, опашка. Програмата ще използва функцията **isEmpty** преди да се опита да прочете поредния елемент от обекта на контейнера. Предикатната функция **isFull** би могла да проверява обект от клас-контейнер дали има достатъчно място в него. Подобни обслужващи функции (**utility**) не са част от интерфейса и те не са предназначени за използване от клиентите на класа, затова те са закрити функции-елементи и се използват за обслужване на другите функции-елементи на класа.

**Пример:**  (с обслужваща функция)

#include <iostream>

#include <iomanip>

using namespace std;

class SalesPerson{

public:

SalesPerson();

void setSales(int, double);

void printAnnualSales();

private:

double sales[12]; // сведения за месечните продажби

double totalAnnualSales();

};

SalesPerson::SalesPerson()

{

for(int i=0; i<12; i++)

sales[i]=0.0;

}

/\* Въвежда стойността на продажбите за посочения месец \*/

void SalesPerson::setSales(int month, double amount)

{

if(month>=1 && month<=12 && amount>0)

sales[month-1]=amount;

else

cout<<"Грешка в месеца или сведенията за продажбите"<<endl;

}

/\* Сумира сведенията за продажбите през годината за

нуждите на printAnnualSales() \*/

double SalesPerson::totalAnnualSales()

{

double total =0.0;

for(int i=0; i<12; i++)

total+=sales[i];

return total;

}

/\* Извежда сумата на продажбите за годината,

като преди това я редактира и форматира \*/

void SalesPerson::printAnnualSales()

{

cout<<setprecision(2)<<setiosflags(ios::fixed | ios::showpoint)

<<endl<<"Cумa на продажбите за година: $"

<<totalAnnualSales()<<endl;

}

void main()

{

SalesPerson s;

double salesFigure;

for (int i=1; i<=12; i++)

{

cout<<"Въведете обема на продажбите за месец "

<<i<<": ";

cin>>salesFigure;

s.setSales(i, salesFigure);

}

s.printAnnualSales();

}

**setprecision(int n)** – манипулатор дефиниран в заглавния файл <iomanip.h>. Този манипулатор установява точност от **n** знака след десетичната точка при извеждане на числа с плаваща точка. По премълчаване точността е 6 знака след десетичната точка, катопри това се извършва закръгляне към най-близкото цяло число.

**setiosflags(long f)** – манипулатор дефиниран в заглавния файл <iomanip.h>. Задава форматни спецификации, определени чрез флага **f**. Константата **ios::fixed** води до извеждане на числата с плаваща точка като дробно-десетични. Константата **ios::showpoint** води до извеждане на десетичната точка и нулевите младши разряди. Двете константи са свързани с ‘|’ (побитово или), тъй като и двете трябва да се включат във флага **f**.

**№ 5 Интерфейс и реализация**

Отделянето на интерфейса от реализацията е един от най-фундаменталните принципи за създаване на добро програмно осигуряване. Това улеснява модификацията на програмата. Измененията в реализацията на класа не влияят на клиента, докато интерфейса на класа не се промени. Препоръчва се декларацията на класа да се оформи като отделен заглавен файл. Така тя е достъпна за всеки клиент, който иска да използва класа. Това формира открития **интерфейс** на класа. Препоръчва се описанието на функциите-елементи на класа да се оформи в отделен първичен файл. Това формира **реализацията** на класа. Заглавният файл, в който е описана декларацията на класа, обаче, може да съдържа някои части от реализацията и кратки сведения за други части от реализацията. Вградените функции-елементи, например, трябва да се намират в заглавния файл – по този начин клиентът може да включва определени функции **inline**. Закритите елементи са изброени в заглавния файл, така че тези елементи са видими за клиента, макар че той няма достъп до тях. Софтуерните фирми използват заглавни файлове + обектни модули за да защитят продуктите си.

При създаване на програма на **С++** всяка декларация на клас обикновено се оформя като заглавен файл, а описанията на фунциите-елементи на този клас се оформят като първичен файл със същото базово име. Заглавните файлове се включват чрез директивата **#include** във всеки първичен файл, в който се използва класа. Тези първични файлове и първичния файл с главната функция се компилират разделно. Разделната компилация се извършва по различен начин в различните среди за програмиране. В някои от средите ( **С**, **С ++**) за разделна компилация се използва **project file**.

// заглавен файл time1.h

// предпазва от многократно включване на заглавния файл

#ifndef TIME1\_H

#define TIME1\_H

// декларация на класа Time

class Time{

public:

Time(); // constructor

void setTime( int, int, int ); // set hour, minute, second

void printMilitary(); // print military time format

void printStandard(); // print standard time format

private:

int hour; // 0 - 23

int minute; // 0 - 59

int second; // 0 - 59

};

#endif

При създаване на големи програми в заглавния файл се включват и други дефиниции и декларации. Използваните директиви на предпроцесора в заглавния файл **time1.h** не позволяват включването на кода между **#ifndef** и **#endif**, ако е дефиниран макросът **TIME1\_H.** Ако заглавния файл **time1.h** още не е включен в първичния файл, то се дефинира макроса **TIME1\_H** чрез **#define**, и заглавния файл се включва в първичния файл. Мнемонично е името на макроса да се получава от името на заглавния файл, като се замени точката с долна черта. Ако заглавния файл вече е включван в първичния файл, то макросът **TIME1\_H** е вече дефиниран и заглавния файл не се включва отново. Опити за многократно включване на заглавен файл обикновено се правят в големи програми с много заглавни файлове, които от своя страна включват други заглавни файлове. Например, ако в **.cpp** файла има **#include** “**a.h**” и **#include** “**b.h**”, но в **b.h** има **#include** “**a.h**” и ако в **a.h** има дефиниции, то те ще се окажат дублирани в **.cpp** файла. В такъв случай се прилага защита срещу повторнo включване.

Функциите-елементи на класа Time са дефинирани(описани) в **time1.cpp**

// първичен файл time1.cpp

#include <iostream>

#include "time1.h" // "" - за текущата директория

using namespace std;

Time::Time()// конструктор

{ hour=minute=second=0;}

void Time::setTime(int h,int m,int s){

hour = (h>=0 && h<24) ? h:0;

minute = (m>=0 && m<60) ? m:0;

second = (s>=0 && s<60) ? s:0;

}

void Time::printMilitary(){

cout<<(hour<10 ? "0":"")<<hour<<":"

<<(minute<10 ? "0":"")<<minute<<":"

<<(second<10 ? "0":"")<< second;

}

void Time::printStandard(){

cout<<((hour==0 || hour==12) ? 12:hour%12)<<":"

<< (minute<10 ? "0":"")<<minute<<":"

<< (second<10 ? "0":"")<<second

<< (hour<12 ? " AM":" PM");

}

================ End of program file **time1.cpp** ===================

// първичен файл prim\_prog.cpp

#include <iostream>

#include "time1.h"

using namespace std;

//описание на функцията клиент

void main(){

Time t; // извиква се конструктора на класа Time

cout<<"Началната стойност във военен формат е ";

t.printMilitary();// 00:00:00

cout<<endl<<"Началната стойност в стандартен формат е ";

t.printStandard();// 12:00:00 AM

t.setTime(13, 27, 6);

cout<<endl<<endl<<"Военното време след setTime e ";

t.printMilitary();// 13:27:06

cout<<endl<<"Стандартното време след setTime e ";

t.printStandard();// 1:27:06 PM

t.setTime(99, 99, 99);

cout<<endl<<endl<<"След задаване на неправилни стйности: "

<<endl<<"Военното време е ";

t.printMilitary();// 00:00:00

cout<<endl<< "Стандартното време e ";

t.printStandard();

cout<<endl; // 12:00:00 AM

}

=============== End of program file **prim\_prog.cpp** ================

**time1.cpp** и **prim\_prog.cpp** се компилират разделно

**№ 6 Конструктори и деструктори**

След създаване на обект, неговите елементи могат да бъдат инициализирани с помощта на функция **конструктор**. Конструкторът е функция-елемент на класа, със същото име като името на класа и се извиква автоматично при създаване на обект. За конструкторите не може да се задава тип на връщания резултат и те не връщат резултат. Конструкторите могат да се предефинират и така да се определя множество от начални стойности за обектите на класа. Когато се дефинира обект от даден клас, може след името на обекта, в скоби, да се зададе списък за инициализация на елементи. Тези начални стойности се предават като параметри в конструктора на класа. Конструктор, при който всички аргументи са аргументи по премълчаване или който няма аргументи се нарича **конструктор по премълчаване**. За всеки клас може да съществува само един конструктор по премълчаване. Конструкторът в разглеждания клас **Time**, който дава нулеви начални стойности на данните-елементи е без аргументи.

Ще модифицираме класа Time така, че всички аргументи на неговия конструктор да са аргументи по премълчаване:

#include <iostream>

using namespace std;

class Time{

public:

Time(int=0, int=0, int=0);

void setTime(int, int, int);

void printMilitary();

void printStandard();

private:

int hour;

int minute;

int second;

};

Time::Time(int hr, int min, int sec)

{ setTime(hr, min, sec);}

void Time::setTime(int h, int m, int s)

{

hour = (h>=0 && h<24) ? h : 0;

minute = (m >=0 && m<60) ? m : 0;

second = (s>=0 && s<60) ? s : 0;

}

void Time::printMilitary()

{**…**}

void Time::printStandard()

{**…**}

Конструкторът **Time** извиква функцията-елемент **setTime** със стойностите, които са подадени към него или със стойности по премълчаване. **setTime** проверявя коректността на данните, анулира некоректните и задава стойност на данните-елементи на класа.

Конструкторът **Time** може да бъде описан със същите оператори, които се съдържат във функцията-елемент **setTime**. Това е по-ефективно, тъй като няма излишно извикване на **setTime**, но еднаквите кодове в конструктора **Time** и функцията-елемент **setTime** затрудняват съпровождането на тази програма. Ако реализацията на функцията **setTime** се промени, съответното изменение трябва да бъде направено и в реализацията на конструктора.

Извикването на **setTime** от конструктора улеснява съпровождането на програмата, тъй като при промяна на реализацията на **setTime**, това ще бъде отразено и в конструктора. Въпреки това се губи ефективност – това може да се поправи, ако функцията **setTime** явно се обяви като **inline** (в описанието извън класа). Ако функцията **setTime** е описана вътре в декларацията на класа, тя по подразбиране се обявява като **inline**.

Добър стил на програмиране е декларирането на аргументи по премълчаване само в прототипа на функцията-елемент вътре в декларацията на класа. Грешка е задаването на начални стойности по премълчаване за една и съща функция-елемент както в прототипа, така и в нейното описание.

void main()

{

Time t1, t2(2), t3(21, 34), t4(12, 25, 42), t5(27, 74, 99);

t1.printMilitary(); //00:00:00  
t2.printMilitary(); //02:00:00  
t3.printMilitary(); //21:34:00  
t4.printMilitary(); //12:25:42  
t5.printMilitary();/\* 00:00:00 заради некоректните данни подадени   
 като аргументи към setTime \*/

}

Ако за класа не е дефиниран конструктор, то компилаторът създава конструктор по премълчаване. Този конструктор не задава начални стойности, така че след създаването на обекта, данните в него са с произволни стойности и най-вероятно са некоректни.

**Деструкторът** е специална функция-елемент на класа. Името на деструктора съвпада с името на класа, но пред него се поставя символа **~** . В това има мнемоника. **~** е операция за поразрядно логическо отрицание и така деструкторът може да се разглежда като отрицание на конструктора. Деструкторът на класа се извиква при унищожаване на обект. Например, деструкторът за даден обект се извиква, когато управлението на изпълняваната програма напуска областта на действие, в която е бил създаден обекта от този клас. В действителност самият деструктор не унищожава обекта. Той извършва завършителни действия, преди системата да освободи паметта, в която е бил съхраняван обекта. Деструкторът няма параметри и не връща резултат. Не се разрешава предефиниране на деструктора, т.е. един клас може да има само един деструктор. На практика деструкторите рядко се използват в обикновени класове. Деструкторите имат смисъл за класове, които използват динамично разпределение на паметта за обектите.

Конструкторите и деструкторите се извикват автоматично. Редът, в който се изпълняват тези извиквания зависи от реда, в който управлението при изпълнението на програмата влиза в и излиза от областите на действие, в които се създават обектите. В общият случай, извикването на деструкторите се изпълнява в ред, обратен на извикването на конструкторите на съответните обекти. Класът памет на обектите може да измени този ред:

* конструкторите на обекти, дефинирани като външни или глобални, се извикват преди изпълнението на функциите от даден файл. Съответните деструктори се извикват когато завърши изпълнението на главната функция **main** или когато се извика функция **exit**.
* конструкторите на автоматичните вътрешни обекти се извикват, когато изпълнението на програмата достигне мястото, където те са дефинирани. Съответните деструктори се изпълняват когато се напусне областта на действие на обектите, т.е. когато се напусне блока в който тези обекти са дефинирани. Конструкторите и деструкторите на автоматичните обекти се извикват всеки път при влизане или излизане от областта им на действие.
* конструкторите на статичните вътрешни обекти се извикват еднократно, когато изпълнението на програмата достигне за първи път мястото, където те са дефинирани. Съответните деструктори се извикват когато завършва главната функция **main** или когато се извиква функция **exit**.

Примерна програма, илюстрираща реда на извикване на конструкторите и деструкторите**:**

#include <iostream.h>

class CreateAndDestroy

{

public:

CreateAndDestroy(int);

~CreateAndDestroy();

private:

int data;

};

CreateAndDestroy::CreateAndDestroy(int value)

{

data=value;

cout<<"Обект "<<data<<" конструктор";

}

CreateAndDestroy::~CreateAndDestroy()

{ cout<<"Обект "<<data<<" деструктор"<<endl;}

void create();// прототип на create

CreateAndDestroy first(1);/\* външен или глобален обект . Неговия конструктор се   
 извиква веднага след стартиране на програмата.   
 Деструкторът на first се извиква при завършване на  
 програмата след като всички други обекти са   
 унищожени. \*/

void main()

{

cout<<" (външен създаден до main)"<<endl;

CreateAndDestroy second(2);

cout<<" (вътрешен автоматичен в main)"<<endl;

static CreateAndDestroy third(3);/\* third е статичен обект и съществува до  
 завършване на програмата.Деструкторът на third   
 се извиква преди деструктора на first, но след   
 унищожаване на всички други обекти. \*/

cout<<" (вътрешен статичен в main) "<<endl;

create();

CreateAndDestroy fourth(4);

cout<<" (вътрешен автоматичен в main)"<<endl;

}

void create()

{

CreateAndDestroy fifth(5);

cout<<" (вътрешен автоматичен в creatе)"<<endl;

static CreateAndDestroy sixth(6);/\* sixth е статичен обект и съществува до   
 завършване на програмата. Деструкторът на sixth се   
 извиква преди деструктора на fourth. \*/

cout<<" (вътрешен статичен в create) "<<endl;

CreateAndDestroy seventh(7);

cout<<" (вътрешен автоматичен в create)"<<endl;

}

Резултат от изпълнението на програмата **:**

Обект 1 конструктор (външен създаден до **main**)

Обект 2 конструктор (вътрешен автоматичен в **main**)

Обект 3 конструктор (вътрешен статичен в **main**)

Обект 5 конструктор (вътрешен автоматичен в **creatе**)

Обект 6 конструктор (вътрешен статичен в **create**)

Обект 7 конструктор (вътрешен автоматичен в **create**)

Обект 7 деструктор

Обект 5 деструктор

Обект 4 конструктор (вътрешен автоматичен в **main**)

Обект 4 деструктор

Обект 2 деструктор

Обект 6 деструктор

Обект 3 деструктор

Обект 1 деструктор

**№ 7 Използване на данни-елементи и функции-елементи**

Закритите данни-елементи на един клас могат да се променят само с помощта на функции-елементи или на приятелски функции на този клас. Тези функции трябва да проверяват коректността на данните (не е задължително, но е желателно). Класовете обикновено съдържат открити функции-елементи, чрез които клиентите на класа задават(записват) или получават(четат) стойностите на закритите данни-елементи. Функциите, които четат се наричат функции за заявки. Често функциите които записват, връщат стойност, която може да се използва от клиентите на класа, за да се установи дали е направен опит да се запишат некоректни данни. Ако данните-елементи на класа са открити, те могат директно да бъдат записвани и четени от всяка функция в програмата, но използването на открити данни-елементи не се препоръчва в ООП.

// заглавен файл time3.h

#ifndef TIME3\_H

#define TIME3\_H

class Time {

public:

Time(int=0, int=0, int=0);

void setTime(int, int, int);

void setHour(int);

void setMinute(int);

void setSecond(int);

int getHour();

int getMinute();

int getSecond();

void printMilitary();

void printStandard();

private:

int hour;

int minute;

int second;

};

#endif

// първичен файл time3.cpp

#include <iostream>

#include "time3.h"

using namespace std;

Time::Time(int hr,int min,int sec)

{ setTime(hr, min, sec);}

void Time::setTime(int h, int m, int s)

{

hour = (h>=0 && h<24) ? h : 0;

minute = (m>=0 && m<60) ? m : 0;

second = (s>=0 && s<60) ? s : 0;

}

void Time::setHour(int h)

{ hour = (h>=0 && h<24) ? h : 0;}

void Time::setMinute(int m)

{ minute = (m>=0 && m<60) ? m : 0;}

void Time::setSecond(int s)

{ second = (s>=0 && s<60) ? s : 0;}

int Time::getHour()

{ return hour;}

int Time::getMinute()

{ return minute;}

int Time::getSecond()

{ return second;}

void Time::printMilitary()

{

cout<< (hour<10 ? "0" :"")<<hour<<":"

<< (minute<10 ? "0" : "")<<minute<<":"

<< (second<10 ? "0" : "")<<second;

}

void Time::printStandard()

{

cout<< ((hour==0 || hour==12) ? 12 : hour%12)<<":"

<< (minute<10 ? "0" : "")<<minute<<":"

<< (second<10 ? "0" : "")<<second

<< (hour<12 ? " AM" : " PM");

}

// първичен файл setget.cpp

#include <iostream>

#include "time3.h"

using namespace std;

void incrementMinutes(Time &, int); //увеличава минутите

void main()

{

Time t;

t.setHour(17);

t.setMinute(34);

t.setSecond(25);

cout<<endl<<"Чac: "<<t.getHour()

<<" Минути: "<<t.getMinute()

<<" Секунди: "<<t.getSecond()

<<endl<<endl;

t.setHour(234);

t.setMinute(43);

t.setSecond(6373);

cout<<endl<<"Чac: "<<t.getHour()

<<" Минути: "<<t.getMinute()

<<" Секунди: "<<t.getSecond()

<<endl<<endl;

t.setTime(11, 58, 0);

incrementMinutes(t, 3);

cout<<endl<<"Чac: "<<t.getHour()

<<" Минути: "<<t.getMinute()

<<" Секунди: "<<t.getSecond()

<<endl<<endl;

}

void incrementMinutes(Time &tt, int count)

{

for(int i=1; i<=count; i++)

{

tt.setMinute((tt.getMinute() + 1)%60);

if(tt.getMinute()==0)

tt.setHour((tt.getHour() + 1)%24);

}

}

В случая формалният параметър е псевдоним на фактическия параметър. Достъпът до закритите данни елементи се осъществява с функциите **get** и **set**, което води до намаляване на ефективността. Този недостатък може да се отстрани чрез използване на приятелски функции.

Възможно е чрез открита функция-елемент на класа да се върне в програмата неконстантен псевдоним или указател на закрита данна-елемент от този клас. Това нарушава енкапсулацията на класа. Пример:

#include <iostream>

using namespace std;

class Time{

public:

Time(int=0, int=0, int=0); // аргументи по премълчаване

void setTime(int, int, int);

int getHour();

int &badSetHour(int);

private:

int hour;

int minute;

int second;

};

Time::Time(int hr, int min, int sec)

{ setTime(hr, min, sec);}

void Time::setTime(int h,int m,int s)

{

hour = (h>=0 && h<24) ? h : 0;

minute =(m>=0 && m<60) ? m : 0;

second =(s>=0 && s<60) ? s : 0;

}

int Time::getHour()

{ return hour;}

int &Time::badSetHour(int hh)

{

hour = (hh>=0 && hh<24) ? hh : 0;

return hour; //опасно връщане на псевдоним на закрита данна-елемент

}

void main()

{

Time t;

int &hourRef = t.badSetHour(20);/\* По този начин hourRef става псевдоним   
 на закритата данна-елемент hour. Чрез дясната част,   
 закритата данна-елемент hour е получила стойност 20. \*/

cout<<endl<<t.getHour()<<endl;

hourRef = 30; // Тази некоректна стойност (hour=30) не се проверява

cout<<endl<<t.getHour()<<endl;

t.badSetHour(12) = 74; /\* hour=74 не се проверява за коректност \*/ cout<<endl<<t.getHour()<<endl;

}

Препоръчва се открита функция-елемент да не връща неконстантен псевдоним или указател на закрита данна-елемент, тъй като това нарушава енкапсулацията на класа.

Един обект може да се присвои на друг обект от същия клас. Такова присвояване обикновено се извършва чрез побитово копиране, т.е. всеки елемент на единия обект се копира индивидуално в съответния елемент на другия обект. Побитовото копиране може да предизвика сериозни проблеми, когато се прилага за класове, чиито данни-елементи използват динамично разпределение на паметта.

#include <iostream>

using namespace std;

class Date{

public:

Date( int=1, int=1, int=1990);

void print();

private:

int month;

int day;

int year;

};

Date::Date(int m, int d, int y)

{

month=m; //за простота не се прави проверка на данните

day=d;

year=y;

}

void Date::print()

{ cout<<month<<"-"<<day<<"-"<<year;}

void main()

{ Date date1(7,4,1993), date2;

cout<<"date1= ";

date1.print();

cout<<endl<<"date2= ";

date2.print();

date2=date1;

cout<<endl<<"date2= ";

date2.print();

cout<<endl;

}

Обектите могат да се предават като параметри на функции и могат да се връщат като резултат. По премълчаване се предава или връща копие на обекта, т.е. се реализира предаване по стойност. Обекта също може да бъде предаден или върнат чрез псевдоним или указател и по този начин да се реализира предаване по име. В този случай формалният параметър е псевдоним на фактическия параметър, а името на функцията е псевдоним или указател на връщания обект. За да не се променя предавания или връщания обект може да се използва ключовата дума **const**.

Съществуват много библиотеки от класове, други са в процес на разработка. Целта е тези библиотеки да станат широко достъпни за да може при създаването на програмно осигуряване да се използват добре описани, внимателно проверени и разбираемо документирани компоненти. Проблем е да се установи дали съществува клас, който съответства на конкретни изисквания на разработчиците на програмно осигуряване.

**№ 8 Константни обекти и константни елементи**

Константните обекти и константните елементи са едни от средствата за прилагане на един от най-фундаменталните принципи за създаване на добро програмно осигуряване, а именно принципа за най-малките привилегии. Ключовата дума **const** се използва за да се укаже, че обектът е неизменяем, т.е. константен, и че всеки опит да се промени обекта е грешка.

const Time noon(12,0,0); //дефиниране на константен обект от клас Time

Опитите да се промени константен обект се откриват като грешка от компилатора, но много компилатори дават само предупреждение. Константни обекти могат да извикват само константни функции-елементи, които не могат да изменят обекта. Една функция-елемент се обявява като константна, като и в декларацията на класа и в описанието на функцията се поставя ключовата дума **const** след заглавието на функцията. За конструкторите и деструкторите на константни обекти не се указва **const**, тъй като конструкторът трябва да има възможност да даде начални стойности на данните-елементи на обекта, а деструкторът трябва да има възможност да изпълни завършителни действия преди унищожаването на обекта.

Препоръчва се да се описват като константни всички функции-елементи, за които се предполага, че ще се използват с константни обекти. Типична грешка е да се описва като константна, функция-елемент, която извиква неконстантна функция-елемент. Константна функция-елемент може да бъде предефинирана с неконстантен вариант. Изборът коя от предефинираните функции-елементи да се използва, се извършва от компилатора автоматично, в зависимост от това дали обекта, който извиква функцията е константен или не е.

Пример (разширение на класа **Time**):

#include <iostream>

using namespace std;

class Time{

public:

Time(int=0, int=0, int=0);

void setTime(int, int, int);

void setHour(int);

void setMinute(int) ;

void setSecond(int);

int getHour()const;

int getMinute()const;

int getSecond()const;

void printMilitary()const;

void printStandard()const;

private:

int hour;

int minute;

int second;

};

Time::Time(int hr,int min,int sec)

{ setTime(hr, min, sec);}

void Time::setTime(int h,int m,int s)

{

hour = (h>=0 && h<24) ? h:0;

minute = (m>=0 && m<60) ? m:0;

second = (s>=0 && s<60) ? s:0;

}

void Time::setHour(int h)

{ hour = (h>=0 && h<24)? h:0;}

void Time::setMinute(int m)

{ minute = (m>=0 && m<60) ? m:0;}

void Time::setSecond(int s)

{ second = (s>=0 && s<60) ? s:0;}

int Time::getHour()const

{ return hour;}

int Time::getMinute()const

{ return minute;}

int Time::getSecond()const

{ return second;}

void Time::printMilitary()const

{

cout<< (hour<10 ? "0":"")<<hour<<":"

<< (minute<10 ? "0":"")<<minute<<":"

<< (second<10 ? "0":"")<<second<<endl;

}

void Time::printStandard()const

{

cout<< ((hour==0 || hour==12) ? 12:hour%12)<<":"

<< (minute<10 ? "0":"")<<minute<<":"

<< (second<10 ? "0":"")<<second

<< (hour<12 ? " AM":" РM")<<endl;

}

void main()

{

const Time t(19,33,52); // създава се константен обект

t.setHour(12);/\* дава грешка при компилиране, защото константен обект  
 извиква неконстантна функция-елемент \*/

t.setMinute(20); // дава грешка

t.setSecond(39); // дава грешка

}

В горния пример, **Borland C++** дава само предупреждение, че неконстантна функция елемент се използва с константен обект. Това означава, че самата функция фактически ще се изпълни, ако се пренебрегне предупреждението (константният обект ще се промени без съобщение за грешка).

Ако данни-елементи на класа са декларирани като константни (**const**), то тогава трябва да се използват инициализатори на елементи, за да може конструкторът на този клас да зададе начални стойности на тези данни-елементи. Пример:

#include <iostream>

using namespace std;

class Increment{

public:

Increment(int c=0,int i=1);

void addIncrement(){count+=increment;}

void print()const;

private:

int count;

const int increment;

};

Increment::Increment(int c, int i):increment(i) /\* инициализатор на константния

{ елемент increment \*/

count = c;

}

/\* инициализатора на константния елемент предизвиква задаване на начална стойност i на данната-елемент increment. Ако е необходимо да се зададат начални стойности на няколко константни данни-елементи, то техните инициализатори се изреждат в списък, разделени със запетаи. Чрез инициализатори могат да се присвояват стойности и на неконстантни данни-елементи \*/

void Increment::print()const

{ cout<<"count= "<<count<<", increment= "<<increment<<endl;}

void main()

{

Increment value(10,5);

cout<<"Преди нарастването: ";

value.print();

for(int j=1;j<=3;j++)

{

value.addIncrement();

cout<<"След нарастване "<<j<<": ";

value.print();

}

}

Типична грешка е опитът за инициализиране на константен елемент чрез директно присвояване, а не чрез инициализатор към конструктора. Ако се запише така:

Increment::Increment(int c, int i)

{

count=c;

increment=i; // грешка

}

компилаторът ще даде предупреждение (**warning**) – неинициализиран константен елемент, защото липсва инициализатор и съобщение за грешка (**error**) – константен елемент не може да се променя с присвояване.

**№ 9 Композиция – класове като елементи на други класове   
(влагане на класове)**

Един клас може да включва в себе си обекти от други класове като данни-елементи. Както е известно, когато обект от даден клас се дефинира (влиза в областта си на действие), автоматично се извиква неговия конструктор. Ако класът съдържа обекти-елементи, в неговия конструктор трябва да се укаже как аргументите се предават към конструкторите на обектите-елементи. Обектите-елементи се създават в този ред, в който са декларирани, а не в този, в който са изброени в списъка от инициализатори на елементи към конструктора и преди да бъдат създадени обектите на включващия ги клас. Начални стойности за обектите-елементи могат да се задават чрез списък от инициализатори към конструктора на включващия ги клас. Ако не е зададен списък от инициализатори, за обектите-елементи автоматично се извиква конструктор с аргументи по премълчаване. След това обектите-елементи могат да бъдат променяни чрез съответните функции-елементи за запис.

Типична грешка е да не е предвиден конструктор с аргументи по премълчаване за обекти-елементи, за които не е зададен инициализатор. В резултат съответният обект-елемент може да остане неинициализиран (въпреки, че за него е извикан служебен конструктор по премълчаване). От съображения за ефективност се препоръчва обектите-елементи да се инициализират явно, с помощта на инициализатори.

#include <iostream>

#include <string.h>

using namespace std;

class Date{

public:

Date(int=1, int=1, int=1900);//конструктор с аргументи по премълчаване

void print()const;

private:

int month;

int day;

int year;

int checkDay(int);

};

Date::Date (int mn,int dy,int yr)

{

if(mn>0&&mn<=12)

month = mn;

else

{ month=1;

cout<<"Месец "<<mn

<<" е некоректен. Записан е месец 1."

<<endl;

}

year = yr;

day=checkDay(dy);

cout<<"Конструктор на обекта Date ";

print();

cout<<endl;

}

int Date::checkDay(int testDay) // проверка дали деня е валиден

{

static int daysPerMonth[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};

if(testDay>0&&testDay<=daysPerMonth[month])

return testDay;

if(month==2&&testDay==29&&(year%400==0||(year%4==0&&year%100!=0)))

return testDay;   
cout<<"Ден "<<testDay

<<" е некоректен. Записан е ден 1."<<endl;

return 1;

}

void Date::print()const

{ cout<<month<<'/'<<day<<'/'<<year;}

class Employee{

public:

Employee(char\*, char\*, int, int, int, int, int, int);

void print()const;

private:

char lastName[25];

char firstName[25];

Date birthDate; //дата на раждане

Date hireDate; //дата на постъпване на работа

};

Employee::Employee(char \*fname, char \*lname,

int bmonth, int bday, int byear,

int hmonth, int hday, int hyear):

birthDate(bmonth, bday, byear),

hireDate(hmonth, hday, hyear)/\* използвани са ициализатори,

{ за да се зададат аргументи за

int length=strlen(fname); конструкторите на обектите-

length=length<25?length:24; елементи \*/

strncpy(firstName,fname,length);

firstName[length]='\0';

length=strlen(lname);

length=length<25?length:24;

strncpy(lastName,lname,length);

lastName[length]='\0' ;

cout<<"Конструктор на обекта Employee: "

<<firstName<<' '<<lastName<<endl;

}

void Employee::print()const

{

cout<<lastName<<", "<<firstName<<endl

<<"Постъпил на работа: ";

hireDate.print();

cout<<endl<<"Роден на: ";

birthDate.print();

cout<<endl;

}

void main()

{

Employee е("Иван", "Петров", 7, 24, 49, 3, 12, 88);

cout<<endl;

е.print();

cout<<endl;

Date d(14, 35, 94);// некоректни данни

}

Резултат от изпълнението на програмата **:**

Конструктор на обекта Date 7/24/49

Конструктор на обекта Date 3/12/88

Конструктор на обекта Employee: Иван Петров

Петров, Иван

Постъпил на работа: 3/12/88

Роден на: 7/24/49

Месец 14 е некоректен. Записан е месец 1.

Ден 35 е некоректен. Записан е ден 1.

Конструктор на обекта Date 1/1/94

**№ 10 Приятелски функции и приятелски класове**

Приятелските функции на един клас се декларират извън областта на действие на този клас, но имат право на достъп до закритите елементи (**private**) и защитените елементи (**protected**) на дадения клас. Функция или цял клас могат да бъдат обявени за приятели на друг клас. Приятелските функции се използват за повишаване на производителността, например при предефиниране на операции или при създаване на класове итератори.

Обектите от класа на итератора се използват за да избират последователно елементи или да изпълняват операции над елементи на обект от класа контейнер. Масиви, стекове, опашки и свързани списъци за примери за класове контейнери. Типични операции върху елементи на обект от клас контейнер са вмъкване, изключване, търсене, сортиране, проверка за наличие на елемент и други.

За да се обяви, че функция е приятел на клас, преди прототипа на функцията в описанието на класа се записва ключовата дума **friend**. За да се обяви например, че **ClassTwo** е приятел на **ClassOne** трябва в описанието на **ClassOne** да се запише **friend** **ClassTwo**; . В този случай всички функции-елементи на **ClassTwo** стават приятелски функции за **ClassOne**. Приятелството не е симетрично.

Обикновено класът на итераторите се обявява като приятелски на класът на контейнерите. Обявяването на приятелство може да стане на произволно място в описанието на класа. Препоръчва се обаче, това да става в началото, преди спецификаторите за достъп до елемент.

Някои програмисти смятат, че приятелството вреди на скриването на информация и отслабва обектно-ориентирания подход.

Примерна програма:

#include <iostream>

using namespace std;

class Count{

friend void setx(Count &, int); //setx е приятелска на класа Count

public:

Count(){x=0;}

void print()const{ cout<<x<<endl;}

private:

int x;

};

void setx(Count &c, int val)

{ c.x = val;}

void main()

{

Count object;

object.print(); //извежда 0

setx(object,8); //задаваме нова стойност на object

object.print(); //извежда 8

}

Ако функцията-елемент **setx** не беше обявена за приятел на класа **Count**, щеше да се получи съобщение за грешка, че данната-елемент **х** не е достъпна за **setx**.

**№ 11 Използване на указателя *this***

Всеки обект може да определи своя собствен адрес с помощта на указателя ***this***. Указателят ***this*** неявно се използва за достъп , както до данните-елементи , така и до функциите-елементи на обекта. Също така, той може да се използва и явно. Указателят ***this*** се инициализира автоматично да сочи към обекта, за който се извиква функция-елемент. Типа на указателя ***this***  зависи от типа на обекта и от това, обявена ли е функцията-елемент, в която се използва ***this,*** като **const**.

в неконстантна функция-елемент, указателят ***this*** има тип**:**

тип\_на\_обекта(име на класа) \***const** - т.е. указателят е константен.

в константни функции-елементи указателя ***this*** има тип**:**

**const** тип\_на\_обекта(име на класа) \***const** - т.е. това е константен указател, който сочи към константа.

Всяка функция-елемент има достъп до указателя ***this*** на обекта, за който е извикана тази функция-елемент. С цел икономия на памет, за всяка функция-елемент съществува само едно копие на клас и тази функция-елемент се извиква от всеки обект на даден клас.От друга страна, всеки обект има свое собствено копие на данните-елементи на своя клас.

Пример за използване на ***this***:

#include <iostream>

using namespace std;

class Test{

public:

Test(int=0);

void print()const;

private:

int x;

};

Test::Test(int a)

{ x=a;}

void Test::print()const

{

cout<<x<<endl

<<this->x<<endl

<<(\*this).x<<endl;

}

Функцията-елемент **print()** отначало извежда **х**, непосредствено след това достъпът до **х** е чрез ***this*** и операцията -> и накрая, чрез обекта, сочен от ***this*** и операцията точка. В израза (\***this**)**.х** кръглите скоби са задължителни, защото операцията ‘.’е с по-висок приоритет от операцията \*. Ако липсват кръглите скоби, ще излезе съобщение за синтактична грешка, тъй като ‘.’не се прилага към указател. Във функцията-елемент **print()** се обръщаме по три начина към данната-елемент **x** на обекта **Time,** които на машинно ниво не се различават.

void main()

{

Test a(12);

a.print();//3 пъти извежда 12

}

Едно интересно приложение на указателя ***this*** е предотвратяване на присвояване на един обект сам на себе си. При предефиниране на операции, самоприсвояването може да стане причина за сериозни грешки в случаите, когато обекта съдържа указател към динамично разпределена памет.

Друго приложение на указателя ***this*** е възможността за слепване на извиквания към функции-елементи.

Примерна програма (с класа **Time**):

#include <iostream>

using namespace std;

class Time{

public:

Time(int=0, int=0, int=0);

Time &setTime(int, int, int);//връща псевдоним на обект от клас Time

Time &setHour(int);

Time &setMinute(int);

Time &setSecond(int);

int getHour()const;

int getMinute()const;

int getSecond()const;

void printMilitary()const;

void printStandard()const;

private:

int hour;

int minute;

int second;

};

Time::Time(int hr,int min,int sec)

{

setTime(hr, min, sec);

}

Time &Time::setTime(int h,int m,int s)

{

hour = (h>=0 && h<24) ? h:0;

minute = (m>=0 && m<60) ? m:0;

second = (s>=0 && s<60) ? s:0;

return \*this;//връща се обекта към който е приложена функцията

}

Time &Time::setHour(int h)

{

hour = (h>=0 && h<24) ? h:0;

return \*this;

}

Time &Time::setMinute(int m)

{

minute = (m>=0 && m<60) ? m:0;

return \*this;

}

Time &Time::setSecond(int s)

{

second = (s>=0 && s<60) ? s:0;

return \*this;}

int Time::getHour()const

{ return hour;}

int Time::getMinute()const

{ return minute;}

int Time::getSecond()const

{ return second;}

void Time::printMilitary()const

{

cout<< (hour<10?"0":"")<<hour<<":"

<< (minute<10?"0":"")<<minute<<":"

<< (second<10?"0":"")<<second;

}

void Time::printStandard()const

{

cout<< ((hour==0 || hour==12) ? 12:hour%12)<<":"

<< (minute<10?"0":"" )<<minute<<":"

<< (second<10?"0":"")<<second

<< (hour<12 ? " AM":" PM");

}

void main( )

{

Time t;

cout<<endl;

t.setHour(18).setMinute(30).setSecond(22);

/\* Слепване по отношение на операцията . (точка)

Асоциативността на . е отляво надясно, т.e. първо се извиква setHour(18),

но тя връща псевдоним на t, за който се извиква setMinute(30), която също

връща псевдоним на t, за който се извиква setSecond(22) \*/

t.printMilitary();//18:30:22

cout<<endl;

t.setTime(20, 20, 20).printStandard();//8:20:20 PM

/\* setTime връща псевдоним на t, за който се изпълнява printStandard().

Ако им разменим местата и запишем t.printStandard().setTime(20,20,20)

това ще доведе до синтактична грешка, тъй като printStandard() не връща

псевдоним на t. \*/

cout<<endl;

}

**№ 12 Статични елементи на клас**

Обикновено всеки обект на клас има свое собствено копие на всички данни-елементи на класа. В определени случаи обаче, е нужно да има само едно копие на някои данни-елементи за всички обекти на класа. За тези и за други цели се използват статични данни-елементи, които съдържат информация за целия клас.

Декларирането на статичните елементи започва с ключовата дума **static**. Препоръчва се, с цел икономия на памет, ако е достатъчно единствено копие на данните, да се използват статични данни-елементи. Статичните данни-елементи имат област на действие клас. Те могат да бъдат **public**, **private** и **protected**. На статичните данни-елементи може да се задават начални стойности само веднъж в областта на действие файл. Достъпът до откритите (**public**) статични елементи е възможен посредством всеки обект на класа или посредством името на класа, с помощта на бинарната операция за разрешаване на област на действие **::** . Вижда се, че откритите статичните данни-елементи на клас са достъпни дори когато не съществуват обекти от този клас. В този случай достъпът до закритите и защитените статични елементи на класа се осъществява с предвидена открита статична функция-елемент, която се извиква, като се използва бинарната операция за разрешаване на област на действие.

Една функция-елемент може да бъде декларирана като **static**, ако тя не трябва да има достъп до нестатични елементи на класа, т.е. всяка статична функция-елемент има достъп само до статичните данни-елементи на класа. За разлика от нестатичните функции-елементи, в статичните не може да се използва указател ***this***, тъй като статичните данни-елементи и статичните функции-елементи съществуват независимо от обектите на класа. Ключовата дума **static** се задава само в декларацията на класа, но не и в описанието на статичната функция-елемент.

Типична грешка е използването на указателя ***this*** в статична функция-елемент. Друга типична грешка е декларирането на статична функция-елемент като **const**, понеже контролът за **const** се осъществява точно от указателя ***this***.

#define NDEBUG/\* предпроцесора игнорира всички оператори за контрол (assert) \*/

#include <iostream>

#include <string.h>

#include <assert.h>

using namespace std;

class Employee

{

public:

Employee(const char \*, const char \*);

~Employee();// деструктор

const char \*getFirstName() const;

const char \*getLastName() const;

static int getCount();//връща броя на създадените обекти

private:

char \*firstName;

char \*lastName;

static int count;//брояч на създадените обекти

};

int Employee::count=0;/\* инициализиране на статичната данна-елемент count в  
 областта на действие файл, когато все още не съществува   
 обект от клас Employee \*/

Ако съществува обект от клас **Employee** то до елемента **count** може да има достъп от всяка функция-елемент на този обект. В примера достъп до **count** има и в конструктора и в деструктора. Ако не съществува обект от клас **Employee** достъпът до **count** се осъществява посредством **Employee::getCount(); .** В примера функцията **getCount()** се използва за определяне на текущия брой на създадените обекти от клас **Employee**.

Employee e1, \*e1Ptr=&e1;

//getCount() може да бъде извикана чрез е1.getCount() или чрез e1Ptr->getCount().

int Employee::getCount()/\* static не се задава при описание

{ return count;} на статична функция-елемент \*/

Employee::Employee(const char \*first, const char \*last)

{

firstName = new char[strlen(first)+1];

assert(firstName!=0); //проверява дали е разпределена успешно памет

strcpy(firstName, first);

lastName = new char[strlen(last)+1];

assert(lastName!=0);

strcpy(lastName, last);

++count; //следи броя на създадените обекти

cout<<"Koнстpyктop Employee зa "<<firstName<<' '

<<lastName<<" e uзвuкaн."<<endl;

}

Функцията **assert** е оператор за контрол. Нейният прототип е описан в заглавния файл **assert.h** . Тя проверява стойността на своя аргумент, който е израз. Ако стойността на израза е 0, т.е **false**, **assert** извежда съобщение за грешка и извиква функцията **abort()** от библиотеката на стандартните функции, която завършва изпълнението на програмата. Съобщението за грешка съдържа проверяваният израз, името на файла съдържащ оператора за контрол и номера на реда, на който се намира този оператор. Прототипът на функцията **abort()** е описан в **stdlib.h**, но за използването на **assert**, този файл не трябва да се включва Операторите за контрол не трябва да се отстраняват от програмата след завършване на нейната проверка. Когато операторите за контрол не са нужни повече, в началото на първичния файл с програмата (преди включване на **assert.h**) се включва директивата **#define NDEBUG** . В резултат на това, предпроцесорът игнорира всички оператори за контрол.

Операция **new** автоматично разпределя памет за обект със съответния размер, извиква конструктора на обекта и връща указател към обекта. Ако **new** не може да намери необходимата памет, връща указател NULL(0).

Employee::~Employee()

{

cout<<"~Employee() e извикан зa "<<firstName<<' '

<<lastName<<endl;

delete []firstName;

delete []lastName;

--count; //защото се унищожава обект

}

Операцията **delete** автоматично извиква деструктора на обекта и освобождава паметта разпределена с **new** за същия този обект.

const char \*Employee::getFirstName() const

{ return firstName;}/\* функцията връща указател към константа, за да се  
 предотврати промяната на закритите данни-елементи от клиента \*/

const char \*Employee::getLastName() const

{ return lastName;} /\* функцията връща указател към константа, за да се  
 предотврати промяната на закритите данни-елементи от клиента \*/

**const** пред типа на връщания резултат предотвратява промяна на закритите данни-елементи от клиента. Ако клиента желае да запази данните, той трябва да копира върнатия низ преди деструктора да освободи динамичната памет, в която се намира низа. Това неудобство може да се отстрани, ако **getFirstName()** и **getLastName()** имат за фактически параметри масив от символи и неговия размер. Тогава функциите биха могли да прекопират името и фамилията в масива от символи.

void main()

{

cout<<"Брой на служителите преди създаване на обекти: "

<<Employee::getCount()<<endl;/\* определя текущия брой на служителите(обектите от клас Employee) \*/

Employee \*e1Ptr=new Employee("Петър", "Pадев ");/\* създава обект от клас   
 Employee и извиква конструктора на обекта \*/

Employee \*e2Ptr=new Employee("Иван", "Михайлов");

cout<<"Брой на служителите след създаване на обекти: "

<<e1Ptr->getCount()<<endl;//2

cout<<endl<<"Служител 1: "<<e1Ptr->getFirstName()

<<" "<<e1Ptr->getLastName()

<<endl<<"Служител 2: "<<e2Ptr->getFirstName()

<<" "<<e2Ptr->getLastName()<<endl<<endl;

delete e1Ptr;/\* извиква деструктора на класа Employee и

освобождава заетата от обекта памет \*/

delete e2Ptr;

cout<<"Брой на служителите след унищожаването: "

<<Employee::getCount()<<endl;

}

Резултат от изпълнението на програмата **:**

Брой на служителите преди създаване на обекти: 0

Koнстpyктop Employee зa Петър Pадев e uзвuкaн.

Koнстpyктop Employee зa Иван Михайлов e uзвuкaн.

Брой на служителите след създаване на обекти: 2

Служител 1: Петър Pадев

Служител 2: Иван Михайлов

~Employee() e извикан зa Петър Pадев

~Employee() e извикан зa Иван Михайлов

Брой на служителите след унищожаването: 0

**№ 13 Предефиниране на операции**

Манипулирането с обектите на класовете се реализира чрез изпращане на съобщение до обектите, т.е. чрез извикване на функции-елементи за обектите. Понякога тези обръщения към функциите-елементи са дълги и обемисти, особено за математически класове. В тези случаи е удобно да се използва богатия набор от наличните в **С++** вградени операции за работа с обекти, което се постига с помощта на механизма за предефиниране на операции. Например, операцията с означение ‘**<<**’ се използва в **C++** за различни цели – или като операция за побитово отместване вляво или като операция за извеждане в поток. По същия начин операцията ‘**>>**’ се използва или за въвеждане от поток или за отместване надясно. Всяка от тези операции е **предефинирана** в библиотеката от класове на **C++**.

Друг пример за предефиниране на операции в **C++** са операциите ‘**+**’ и ‘**-**‘ – те се изпълняват по различен начин при цели операнди, при операнди с плаваща точка или при адресна аритметика. **С++** не позволява да се създават означения за нови операции.

Препоръчва се да се използва предефиниране на операции, само ако това прави програмата по-ясна в сравнение с използването на явни обръщения към функциите, които изпълняват тези операции. Операциите се предефинират чрез функция с име, което се състои от ключовата дума **operator** и означението на операцията. Например **operator+** е име на предефинираща функция за операцията за събиране.

За да се използва операция с обекти на класове, тази операция трябва да бъде предефинирана с две изключения **:**

**1.** операцията за присвояване ‘**=**’ може да се използва с всеки клас без явно да се предефинира. По премълчаване тази операция се свежда до побитово копиране на данните-елементи на класа. Такова побитово копиране, обаче, е опасно за класове с елементи указатели, които сочат към динамично разпределена памет. В такъв случай след присвояване на един обект на друг, различни указатели ще сочат към една и съща динамично разпределена област от паметта. Изпълнението на деструктора на кой да е от тези обекти ще освободи тази област от паметта и ако след това чрез другия обект има обръщение към вече освободената памет, резултата ще бъде неопределен. За такива класове трябва явно да се предефинира операцията за присвояване ‘**=**’.

**2.** операцията за адресиране ‘**&**’ също може да се използва с обекти от всеки клас без да се предефинира. Тя връща адреса на обекта в паметта. Въпреки това, операцията за адресиране също може да се предефинира.

Предефинирането на операции е характерно за математическите класове, например предефинирането на операции за работа с комплексни числа. Препоръчва се операциите да се предефинират така, че те да извършват над обектите на класа същата функция или близка до нея, каквато те изпълняват над обектите от вградените типове.

Отделните компилатори имат особености при работа с предефинирани операции. Повечето операции в **C++** могат да бъдат предефинирани. Не могат да бъдат предефинирани операциите за условен израз (**?:**), за разрешаване на област на действие (**::**), операцията за избор на елемент на обект ‘**.**’ и операцията **sizeof**. Приоритета и асоциативността на операциите не могат да бъдат променяни при тяхното предефиниране. При предефинирането на операции не могат да се използват аргументи по премълчаване. Също така не е възможно да се промени броя на операндите на операциите – предефинираните унарни операции остават унарни, а предефинираните бинарни – бинарни. Операциите ‘**+**’, ‘**\***’, ‘**&**’, ‘**-**‘ имат бинарен и унарен вариант и тези варианти се предефинират отделно. Не могат да се създават означения за нови операции. Това забранява на програмистите да използват някои популярни означения, например ‘\*\*’ за повдигане на степен, която се използва в **Basic** и **Fortran**. Предефинирането не може да промени начина на изпълнение на операциите с вградени типове. Предефинирането има значение само за работа с обекти дефинирани от потребителя или със смесени обекти от типове, дефинирани от потребителя и вградени типове.

Предефинирането на операциите за присвояване и събиране ни позволяват да напишем **:**

object2 = object2 + object1;

но това не означава, че операцията ‘**+=**' също е предефинирана, за да може да напишем

object2 += object1;

Това може да се постигне чрез явно предефиниране на операцията ‘**+=**' за дадения клас. За да се осигури съгласуваност на такива свързани операции се препоръчва при предефинирането на едните да се използват другите. Например при предефинирането на ‘**+=**’ да се използва операцията ‘**+**’, която вече е предефинирана.

В различни случаи предефиниращите функции е най-добре да бъдат приятелски функции или функции елементи или обикновени функции.

Ако левият операнд на предефинираната операция трябва задължително да бъде обект на клас или псевдоним на обект на клас, тогава тя се предефинира с функция-елемент. Например операциите за извикване на функция ‘**()**’, за достъп до елемент на масив ‘**[]**’, указателната операция ‘**->**’ и операцията за присвояване ‘**=**’ се предефинират с функция-елемент на клас.

Ако левият операнд трябва да бъде обект от друг клас или от вграден тип, тогава операцията не може да бъде предефинирана с функция-елемент. Ако тази предефинираща функция трябва да има достъп до закритите или защитените данни елементи на класа, препоръчително е тя да бъде приятел на този клас. Друга причина за използване на предефинираща функция, която не е елемент на класа е когато искаме предефинираната операция да е комутативна. Ако операция се предефинира с обикновена функция, тогава достъпът до закритите и защитените данни елементи на класа ще бъде неефективен.

Унарна операция може да се предефинира с помощта на нестатична функция-елемент без аргументи или с функция, която не е елемент на клас с един аргумент. Този аргумент трябва да бъде или обект на клас или псевдоним на обект на клас; предефиниращата функция-елемент трябва да е нестатична, за да има достъп до нестатичните данни-елементи на класа.

Бинарна операция може да се предефинира с помощта на нестатична функция-елемент с един аргумент или с функция, която не е елемент на клас с два аргумента. Единият от тези аргументи трябва да бъде или обект от клас или псевдоним на обект от клас.

Пример за предефиниране на операции за въвеждане и извеждане в поток. Те са предефинирани в библиотеката от класове на компилатора за работа с вградени типове. Ще ги предефинираме за обекти на PhoneNumber. Ще се предполага, че телефонните номера са коректни.

#include <iostream>

using namespace std;

class PhoneNumber{

friend ostream& operator<<(ostream&, const PhoneNumber&);

friend istream& operator>>(istream&, PhoneNumber&);

private:

char areaCode[4]; //код

char exchange[4]; //трицифрен комутатор

char line[5]; //4цифрена линия

};

Предефинираните операции трябва да имат ляв операнд от тип **ostream** (**istream**), например **cout** (**cin**), така че предефиниращите функции не могат да бъдат функции-елементи на класа. Тези функции се нуждаят от пряк достъп до закритите данни-елементи на класа и затова се обявени като приятелски функции на този клас.

//пример за телефонен номер: (123) 456 7890

ostream& operator<<(ostream& output, const PhoneNumber& num){

output<<"("<<num.areaCode<<") "<<num.exchange<<"-"<<num.line;

return output; //осигурява слепване на извиквания

}

istream& operator>>(istream& input, PhoneNumber& num){

input.ignore(); //пропуска при въвеждането '('

/\* функцията ignore е функция-елемент на класа istream, който е описан в iostream.h. Tя игнорира указания брой символи от входния поток.   
По подразбиране игнорира един символ. \*/

input.getline(num.areaCode,4); /\* в num.areaCode се записва общия брой на   
 символите - 4 (това включва и '\0') \*/

/\* функцията getline е функция-елемент на класа istream. Tя прочита   
 указаният брой символи и ги записва в низ, като автоматично добавя   
 нулев байт \*/

input.ignore(2);

input.getline(num.exchange,4);

input.ignore();

input.getline(num.line,5);

return input; //осигурява слепване на извиквания

}

void main()

{

PhoneNumber phone;

cout<<"Въведете телефонен номер във вида (123) 456 7890: "<<endl;

cin>>phone; //<=> operator>>(cin,phone);

/\*След извикване на предефинираната функция operator>> параметъра input   
 става псевдоним на cin, а параметъра num става & на phone. Операцията >>   
 въвежда телефонен номер от входния поток в обекта phone. Накрая връща & на   
 cin. Това позволява операцията >> за тип PhoneNumber да е слепена за други   
 такива обекти или обекти от други типове. Например :

cin>>phone1>>phone2;

Отначало ще се изпълни cin>>phone1 чрез operator>>(cin,phone1). Това   
 обръщение към функция ще върне псевдоним на cin. Тогава останалата част ще   
 се интерпретира като cin>>phone2, което е тъждествено на   
operator>>(cin,phone2) \*/

cout<<"Въведения телефонен номер е:\n"<<phone<<endl;

}

Когато компилатора срещне cout<<phone ще генерира обръщението operator<<(cout,phone).

**№ 14 Пример за клас масив**

Ще създадем клас масив, който извършва проверка на диапазона, за да се гарантира, че индексите остават в границите на масива. Обектите на този клас ще съдържат самият масив, както и броя на елементите в него, така че при предаването на масиви като параметри на функции няма нужда от отделен параметър за техния размер. Позволява се **присвояване** (на един обект масив да се присвои друг обект масив), **въвеждане** на масив, **извеждане** на масив, както и **сравняване** на масиви (с **==** и **!=**). Класът масив използва статичен елемент **arrayCount**, който съдържа броя на създадените обекти масиви в програмата.

#include <iostream>

#include <assert.h>

using namespace std;

class Array{

friend ostream& operator<<(ostream&, const Array&);

friend istream& operator>>(istream&, Array&);

public:

Array(int=10);

Array(const Array&); //copy constructor

~Array();

int getsize()const;

const Array& operator=(const Array&);

int operator==(const Array&)const;

int operator!=(const Array&)const;

int& operator[](int);

static int getArrayCount(); //за броя на направените масиви

private:

int \*ptr; //указател към първия елемент на масива

int size; //размер на масива

static int arrayCount;//брой на създадените елементи масиви

};

int Array::arrayCount=0;

Array::Array(int arraySize)

{

++arrayCount;

size=arraySize; //по премълчаване е 10

ptr=new int[size];

assert(ptr!=0);

for(int i=0;i<size;i++)

ptr[i]=0;

}

Първият конструктор, **Array (int = 10)**, ще се извиква когато компилаторът срещне израза **Array integers (7);** Еквивалентен запис е **Array integers = 7;** Изразът **Array integers;** в този случай е еквивалентен на **Array integers (10);,**който пък е еквивалентен на **Array integers = 10;** Конструктора увеличава с единица **arrayCount**, копира параметъра **arraySize** в елемента от данни **size**, с помощта на **new**  заделя динамична памет за масива и връща указател, който се присвоява на данната-елемент **ptr**. Използва **assert** за проверка дали нормално е завършил **new** и след това инициализира масива с нулеви стойности. Може **Array** да не задава начални стойности на масива, но това е лош стил на програмиране. Масивът трябва винаги да бъде инициализиран с коректно въведени данни.

Array::Array(const Array &init)// конструктор за копие

{

++arrayCount;

size=init.size;

ptr=new int[size];

assert(ptr!=0);

for(int i=0;i<size;i++)

ptr[i]=init.ptr[i];

}

**Конструкторът за копие** ( **copy constructor** ) задава начални стойности на обект от класа **Array** чрез копиране на стойностите на съществуващ обект от същия клас. Такова копиране трябва да се извършва много внимателно, за да се избегнат уловки състоящи се в това, и двата обекта да съдържат данни-елементи указатели, които сочат към една и съща динамично разпределена памет.

Конструкторът **Array (const Array &)** е **конструктор за копие.** Той се извиква всеки път когато възникне необходимост да се копира обект – например при връщане по стойност на обект от функция, при предаване по стойност на обект към функция. Конструкторът за копие също така се извиква при инициализиране на обект, който трябва да е копие на друг обект от същия клас. Например **: Array integers2(integers1);** , което е еквивалентно на **Array integers2 = integers1;**

Ще отбележим, че конструкторът за копие трябва да използва извикване по име, т.е. да бъде **Array&**, а не извикване по стойност, т.е. **Array** . Това е така, защото при извикване по стойност активирането на **copy constructor** ще доведе до безкрайна рекурсия, тъй като при всяко извикване по стойност трябва да се създаде копие на обекта, което се предава на конструктора за копие, което води до ново активиране на конструктора за копие и т.н.

Ако **copy constructor**-a само копира указателя **ptr** от стария обект в новия, те ще сочат към една и съща динамично разпределена памет. Изпълнението на първия деструктор по нататък ще унищожи динамично разпределената памет и указателя **ptr** на останалите обекти ще се окаже неопределен. Това ще доведе до ситуации, в които ще възникнат сериозни грешки в програмата.

**Добър стил на програмиране е конструкторът, конструкторът за копие, деструкторът и предефинираната операция за присвояване ‘=’ да се използват съвместно за класове, които използват динамично разпределена памет.**

Array::~Array()

{

--arrayCount;

delete []ptr;

}

int Array::getsize()const

{ return size;}

const Array &Array::operator=(const Array &right)

{

if(&right!=this) // проверка за самоприсвояване

{

delete[]ptr;

size=right.size;

ptr=new int[size];

assert(ptr!=0);

for(int i =0;i<size;i++)

ptr[i]=right.ptr[i];

}

return \*this;

}

В предефиниращата функция за операцията ‘=’ се прави проверка за самоприсвояване. Ако има опит за самоприсвояване, присвояването се пропуска тъй като фактически е реализирано. Ако нямаше такава проверка, масивът десен операнд щеше да се унищожи още преди самото присвояване ( защото и в левия и в десния операнд, указателите **ptr** щяха да сочат към една и съща област в динамично разпределената памет ).

Типична грешка е отсъствието на проверка за самоприсвояване при предефиниране на операцията ‘=’ за клас, който съдържа указател към динамично разпределена област от паметта.

По-нататък в програмата, се унищожава масива отляво на ‘=’ и се създава нов масив с размера на масива отдясно на ‘=’. След това масива десен операнд се копира в масива ляв операнд. Независимо от това има или няма самоприсвояване предефиниращата функция връща като резултат левият операнд, което дава възможност за слепване.

Когато компилатора срещне израз **integers1 = integers2;** ще генерира извикването **integers1.operator=(integers2);**

Типична грешка е да не се предвиди предефиниране на оператора ‘=’ и **copy constructor**-a за клас, когато обектите на този клас съдържат указатели към динамично разпределена памет.

Възможно е да се забрани присвояването на един обект на класа на друг. Това се постига като предефиниращата функция за операцията ‘=’ се опише като закрит елемент на класа.

Могат да се предпазят обектите на класа от копиране. За тази цел се правят закрити и предефиниращата функция за ‘=’ и **copy constructor**-a.

int Array::operator==(const Array &right)const

{

if(size!=right.size)

return 0;

for(int i=0;i<size;i++)

{

if(ptr[i]!=right.ptr[i])

return 0;

}

return 1;

}

/\* Ако компилатора срещне израз от вида **integers1==integers2**

ще генерира извикването **integers1.operator==(integers2)** \*/

int Array::operator!=(const Array& right)const

{

if(size!=right.size) return 1;

for(int i=0;i<size;i++)

{

if(ptr[i]!=right.ptr[i])

return 1;

}

return 0;

}

/\* Може да се опише и така с помощта на operator==

int Array::operator!=(const Array &right)const

{

return !(\*this==right);

} \*/

int Array::getArrayCount()

{ return arrayCount;}

/\* Тази функция може да се използва дори и да не съществува нито един обект от класа, защото е статична \*/

int &Array::operator[](int subscript)

{

assert(0<=subscript && subscript<size);

return ptr[subscript];

}

/\* Трябва да се използва **assert**, защото не трябва да се прави проверка за коректност при вече работеща програма. При предефинирането на операцията за индексиране ‘**[]**' , резултатът е псевдоним на елемент от масива, което позволява този резултат да се използва като ляв операнд в присвояване. Когато срещне израза **integers1[5] = 1000;** компилаторът го заменя с **integers1.operator[](5) = 1000;**

Операция за индексиране се използва не само за масиви. Може да се използва и за избор на елементи и от други видове класове контейнери - свързани списъци, низове. Освен това индексите не винаги са цели числа. Могат да бъдат символи или някой друг тип. \*/

istream &operator>>(istream &input, Array &a)

{

for(int i=0;i<a.size;i++)

input>>a.ptr[i];

return input;

}

ostream &operator<<(ostream &output,const Array &a)

{

for(int i=0;i<a.size;i++)

{

output<<a.ptr[i]<<' ';

if((i+1)%10==0)

output<<endl ;

}

/\*if (i%10!=0)\*/ //!!!!!!!!!!!!!!!!!!!!!!

if(a.size%10!=0)

output<<endl;

return output;

}

void main()

{

cout<<"Брой на създадените масиви = "

<<Array::getArrayCount()<<endl; // 0

Array integers1(7), integers2; /\* integers1 има 7 елемента,   
 а integers2 има 10 \*/

cout<<"Бpoй на създадените масиви = "

<<Array::getArrayCount()<<endl; // 2

cout<<"Размерът на масива integers1= "

<<integers1.getSize()<<endl // 7

<<"Масивът след задаване на начални стойности: "<<endl

<<integers1<<endl; //0 0 0 0 0 0 0

/\* Аналогично се извеждат елементите и размера на 2-рия масив \*/

cout<<"Размерът на масива integers2= "

<<integers2.getSize()<<endl

<<"Масивът след задаване на начални стойности: "<<endl

<<integers2<<endl;

cout<<"Въведете 17 цели числа: "<<endl;

cin>>integers1>>integers2;

cout<<"След въвеждането масива съдържа: "<<endl

<<"integers1: "<<integers1

<<"integers2: "<<integers2;

cout<<"Оценка: integers1!=integers2 "<<endl;

if(integers1!=integers2)

cout<<"Te не са равни"<<endl;

Array integers3=integers1; //integers3 е копие на integers1

cout<<"Размерът на масива integers3= "

<<integers3.getSize()<<endl

<<"Масивът след задаване на начални стойности: "<<endl

<<integers3<<endl;

cout<<"Присвояване масива integers2 на масива integers1:"<<endl;

integers1=integers2;

cout<<"integersl: "<<integers1

<<"integers2: "<<integers2;

cout<<"Оценка: integers1==integers2 "<<endl;

if (integers1==integers2)

cout<<"Te са равни"<<endl<<endl;

cout<<"integers1[5]= "<<integers1[5]<<endl;

integers1[5]=1000;

cout<<"integers1: "<<integers1<<endl;

integers1[15]=1000;

/\* Изпълнението на програмата завършва с грешка от **assert** \*/

}

**№ 15 Пример за клас низ**

В **С++** няма вграден тип низ. Със средствата на **С++** може да се построи клас, който да управлява създаването и обработката на низове.

#include <iostream>

#include <iomanip>

#include <string.h>

#include <assert.h>

using namespace std;

class String{

friend ostream &operator<<(ostream &,const String &);

friend istream &operator>>(istream &, String &);

public:

String(const char\* = "");/\* конструктор за преобразуване   
 ("" е аргумент по премълчаване) \*/

String (const String &);//copy constructor

~String();

const String &operator=(const String &);

String &operator+=(const String &);

int operator!()const;//използва се за проверка дали един низ е празен

int operator==(const String &)const;

int operator!=(const String &)const;

int operator<(const String &)const;

int operator>(const String &)const;

int operator<=(const String &)const;

int operator>=(const String &)const;

char &operator[](int);

String operator()(int, int);//извличане на подниз

int getLength()const;

private:

char\* sPtr;/\* указател към динамично разпределена памет   
 за съхранение на низ от символи \*/

int length;//дължина на низа без '\0'

};

String::String(const char\* s)

{

cout<<"Конструктор за преобразуване: "<<s<<endl;

length = strlen(s); //взема дължината на **s** без '\0'

sPtr = new char[length+1];

assert(sPtr!=0);

strcpy(sPtr,s);

}

Компилаторът не знае как да преобразува вграден тип в тип, дефиниран от потребителя. Програмистът трябва явно да укаже как се извършват тези преобразувания - това може да стане чрез **конструктор за преобразуване.** Това е конструктор с един аргумент, който се използва за преобразуване на аргумента в обект от класа на конструктора. Компилаторът може да извиква такъв конструктор неявно. Обикновено предефинираната операция ‘**=**’ се използва за присвояване на един обект, друг обект от същия клас. С помощта на конструктора за преобразуване, предефинираната операция ‘**=**’ също може да се използва за присвояване на обекти от различни класове или за присвояване на един обект стойност или променлива от вграден тип. Всеки конструктор с единствен аргумент може да се разглежда като конструктор за преобразуване. Важно е да се отбележи, че при неявно преобразуване на типовете, дефинирано от потребителя (става въпрос за преобразуването), в **C++** може да се извика само един конструктор, т.е. съгласуването на типовете в един израз не може да се получава след повече от едно преобразуване, дефинирано от потребителя. Възможно е преди да се изпълни потребителско преобразуване, компилаторът да извърши преобразуване между вградени типове и класове.

В примера конструкторът преобразува тип **char\*** в обект от клас **String** и това (заедно с предефинирата операция ‘**=**’) позволява, на обект от клас **String** да се присвои масив от символи.

Конструкторът за преобразуване може да бъде извикан по следните начини :

String s1("Поздрав!”);

String s1 = "Поздрав!”;

String s1; s1 = "Поздрав!”;

String::String(const String & copy)

{

cout<<"Конструктор за копие: "<<copy.sPtr<<endl;

length = copy.length;

sPtr = new char[length +1];

assert(sPtr!=0);

strcpy(sPtr,copy.sPtr);

}

/\* **Copy constructor**-a инициализира нов обект от клас **String**, като прави това от вече съществуващ обект на класа. Такова копиране трябва да се извършва акуратно за да се избегнат уловките, когато два обекта от клас **String** сочат към една и съща област от динамично разпределената памет. По тази причина не се прави просто копиране на **sPtr** от стария в новия обект. \*/

String::~String()

{

cout<<"Деструктор: "<<sPtr<<endl;

delete []sPtr;

}

const String &String::operator=(const String &right)

{

cout <<"Извикване на operator="<<endl;

if (&right!=this) // проверка за самоприсвояване

{

delete[]sPtr;

length = right.length;

sPtr = new char[length+1];

assert(sPtr!=0);

strcpy(sPtr, right.sPtr);

}

else

cout<<"Onuт за самоприсвояване"<<endl;

return \*this;

}

/\* **string1=string2;** е еквивалентно на **string1.operator=(string2);**

В предефиниращата функция за операцията ‘**=**’ се прави проверка за самоприсвояване. Ако има опит за самоприсвояване, присвояването се пропуска тъй като фактически е реализирано. Ако нямаше такава проверка , низът десен операнд щеше да се унищожи още преди самото присвояване ( защото и в левия и в десния операнд, указателите **sPtr** щяха да сочат към една и съща област в динамично разпределената памет ). Ако няма самоприсвояване, левия операнд се унищожава и се създава нов ляв операнд с дължината на десния, в който се копира низа от десния операнд. Независимо дали има или няма самоприсвояване, се връща левия операнд, което дава възможност за слепване. \*/

String &String::operator+=(const String &right)

{

char\* tempPtr = sPtr;

length += right.length;

sPtr = new char[length+1];

assert(sPtr!=0);

strcpy(sPtr, tempPtr);

strcat(sPtr, right.sPtr);

delete []tempPtr;

return \*this;

}

/\* **s1+=s2;** е еквивалентно на **s1.operator+=(s2);**

Предефиниращата функция за операцията конкатенация ‘**+=**’ създава временен указател **tempPtr**, за да съхрани низа от символи на текущия обект. Намира общата дължина на конкатенирания низ, заделя динамична памет за него и го формира, чрез копиране в него на временно създадения низ и конкатенация с низа сочен от указателя **right.sPtr** на десния операнд. Връща текущия обект като псевдоним, за да осигури възможност за слепване.

Не е необходимо да се предефинира за втори път операцията за конкатенация ‘**+=**’ , за аргумент с тип **const char\***. Конструктора за преобразуване **String(const char\* s)** ще преобразува низа във временен обект от клас **String** , с който се изпълнява предефинираната операция ‘**+=**’. **С++** може да изпълни не повече от едно такова преобразуване. Може също да извърши преобразуване на вградени типове преди да изпълни преобразуване между вградени типове и класове.

При създаване на временен обект от клас **String** при (**s1 += " to you";**) , се извиква конструктора за преобразуване, а при неговото унищожаване се извиква деструктора. Това са допълнителни разходи на памет при неявни преобразувания. Аналогични разходи създава конструкторът за копие при предаване на обекти като параметри по стойност към функции или при връщане по стойност на обект като резултат от функция. Това води до неефективност в горната програма. Например, по-ефективно е предефиниращата функция за операцията ‘**+=**’ да приема направо аргумент от тип **(const** **char\*)** , отколкото да се изпълни неявно преобразуване и конкатенация. От друга страна неявните преобразувания водят до по-малък обем на програмата и по-малко грешки в нея. \*/

int String::operator!()const

{ return length==0;}

int String::operator==(const String& right)const

{ return strcmp(sPtr,right.sPtr)==0;}

int String::operator!=(const String& right)const

{ return strcmp(sPtr,right.sPtr)!=0;}

int String::operator<(const String& right)const

{ return strcmp(sPtr,right.sPtr)<0;}

int String::operator<=(const String& right)const

{ return strcmp(sPtr,right.sPtr)<=0;}

int String::operator>(const String& right)const

{ return strcmp(sPtr,right.sPtr)>0;}

int String::operator>=(const String& right)const

{ return strcmp(sPtr,right.sPtr)>=0;}

/\* Много програмисти предпочитат да използват вече предефинирани операции при предефиниране на нови операции. Например **:**

int String::operator>=(const String& right)const

{ return (\*this>right || \*this==right) ? 1 : 0;} \*/

char& String::operator[](int subscript)

{

assert(subscript>=0 && subscript<length);

return sPtr[subscript];

}

/\* **string[i]** е еквивалентно на **string.operator[](i)**

Ако индекса е в диапазона, функцията връща като резултат псевдоним на елемент от съответния низ, което позволява този резултат да се използва като ляв операнд в присвояване ‘**=**’ и да се модифицира указания символ от низа. \*/

/\* Прието е да се предефинира ‘()**’** за отделяне на подниз от низ обект на **String**

String String::operator()(int index, int subLength)

{ /\* двата цели параметъра указват индекса на началото и дължината на подниза \*/

assert(index>=0&&index<length&&subLength>=0);

String sub;/\* извиква конструктор за преобразуване **String(const char\* ””)**,

където "" е аргумент по подразбиране \*/

if((subLength==0||(index+subLength>length)))

sub.length = length-index; /\* приема се, че при дължина на подниза 0   
 се взема низа до края \*/

else

sub.length = subLength;

delete sub.sPtr;

sub.sPtr = new char[sub.length+1];

assert(sub.sPtr!=0);

strncpy(sub.sPtr, &sPtr[index], sub.length);

sub.sPtr[sub.length] = '\0';

return sub;

}

/\* **string1(2,2)** е еквивалентно на **string1.operator()(2,2)**

Предефинирането на операцията ‘**()**' е мощно средство, тъй като функциите могат да имат списъци от параметри с произволна дължина и сложност. Да отбележим, че предефиниращата функция на тази операция връща копие на локален обект, а не псевдоним, тъй като локалният обект се унищожава след напускане на функцията. Това копие ще бъде унищожено веднага след използването на резултата (подниза) в съответен израз. \*/

int String::getLength()const

{ return length;}

ostream& operator<<(ostream& output,const String& s)

{

output<<s.sPtr;

return output;

}

istream &operator>>(istream &input, String &s)

{

char temp[100];

input>>setw(100)>>temp; //няма да бъдат прочетени повече от 99 символа

s = temp;/\* извършва се неявно преобразуване и се използва   
 предефинираната операция ’**=**’ \*/

return input;

}

/\* Манипулаторът setw е описан в **iomanip.h** . Той гарантира, че няма да бъдат

прочетени повече от 99 символа (добавя се нулев байт). \*/

void main()

{

String s1("happy"), s2(" birthday"), s3;

/\*Конструктор за преобразуване: happy  
 Конструктор за преобразуване: birthday  
 Конструктор за преобразуване: \*/

cout<<"s1 = \""<<s1<<"\";s2 = \"" // s1 = "happy";

<<s2<<"\";s3 = \";"<<s3<<'\"'<<endl // s2 = " birthday";s3 = "";

<<"s2==s1 е "<<(s2==s1)<<endl

<<"s2!=sl е "<<(s2!=s1)<<endl

<<"s2>s1 е "<<(s2>s1)<<endl

<<"s2<sl е "<<(s2<s1)<<endl

<<"s2>=s1 е "<<(s2>=s1)<<endl

<<"s2<=s1 е "<<(s2<=s1)<<endl;

if (!s3)

{

cout<< "s3 е празен низ"<<endl;

s3 = s1; //"Извикване на operator="

cout<<"s3 = \""<<s3<<"\""<<endl; //s3 = "happy"

}

s1 += s2;

cout<<"sl += s2 дава резултат s1 = "<<s1<<endl;

cout<<"sl += \" to you\" дава резултат "<<endl;

s1 += " to you"; /\* Конструктор за преобразуване: to you

Деструктор: to you \*/

cout<<"sl = " <<s1<<endl; // sl = happy birthday to you

cout<<"sl(0, 14) = "<<s1(0,14)<<endl; /\* Конструктор за преобразуване:

sl(0, 14) = happy birthday

Деструктор: happy birthday \*/

cout<<"sl(15, 0) = "<<s1(15, 0)<<endl; /\* Конструктор за преобразуване:

sl(15, 0) = to you

Деструктор: to you \*/

String \*s4Ptr=new String(s1);//Конструктор за копие: happy birthday to you

cout<<"\*s4Ptr = "<<\*s4Ptr<<endl; // \*s4Ptr = happy birthday to you

cout<<"Самоприсвояване на \*s4Ptr"<<endl; //Самоприсвояване на \*s4Ptr

\*s4Ptr = \*s4Ptr;/\* Извикване на operator=

Onuт за самоприсвояване \*/

cout<<"\*s4Ptr = "<<\*s4Ptr<<endl;// \*s4Ptr = happy birthday to you

delete s4Ptr;// Деструктор: happy birthday to you

s1[0] = 'H';

s1[6] = 'B';

cout<<"sl = "<<s1<<endl; // sl = Happy Birthday to you

s1[25] = 'Y'; // съобщение за грешка и излизане от програмата

}

Резултат от изпълнението на програмата **:**

Конструктор за преобразуване: happy

Конструктор за преобразуване: birthday

Конструктор за преобразуване:

s1 = "happy";s2 = " birthday";s3 = ";"

s2==s1 е 0

s2!=sl е 1

s2>s1 е 0

s2<sl е 1

s2>=s1 е 0

s2<=s1 е 1

s3 е празен низ

Извикване на operator=

s3 = "happy"

sl += s2 дава резултат s1 = happy birthday

sl += " to you" дава резултат

Конструктор за преобразуване: to you

Деструктор: to you

sl = happy birthday to you

Конструктор за преобразуване:

sl(0, 14) = happy birthday

Деструктор: happy birthday

Конструктор за преобразуване:

sl(15, 0) = to you

Деструктор: to you

Конструктор за копие: happy birthday to you

\*s4Ptr = happy birthday to you

Самоприсвояване на \*s4Ptr

Извикване на operator=

Onuт за самоприсвояване

\*s4Ptr = happy birthday to you

Деструктор: happy birthday to you

sl = Happy Birthday to you

Assertion failed:subscript>=0&&subscript<length, file .\15\_shb\_173.cpp,line 136

**Операция за преобразуване** може да се използва за преобразуване на обект от един клас в обект от друг клас или в обект от вграден тип. Такава операция за преобразуване трябва да бъде нестатична функция-елемент, тя не може да бъде приятелска функция. Например следният прототип

**А::operator char\*() const;**

декларира предефинираща функция на операция за преобразуване, която създава временен обект от тип **char\*** чрез преобразуване на обект от клас **А**. В предефиниращата функция на операция за преобразуване не се задава типа на връщания резултат, тъй като това е типът към който се преобразува обекта. Например ако **s** е обект от клас, то когато компилаторът срещне израза

**(char \*) s** , той генерира извикването на функцията **s.operator char \*()** .

Прототипите:

**A::operator int()const;**

**A::operator otherClass()const;**

декларират предефиниращи функции на операции за преобразуване на обект от клас **А** в цял тип и в обект от друг потребителски тип **otherClass**.

Ако е необходимо, компилаторът автоматично извиква операциите и конструкторите за преобразуване при създаване на временни обекти, които да се използват в изрази, при предаване на параметри на функция или при връщане на резултат от функция. Например, ако операцията за преобразуване на обект от класа **String** към обект от тип **char\*** беше определена, това позволява да не се предефинира операцията за извеждане в поток (**cout**) за този клас.

**№ 16 Пример за клас дата**

Ще дефинираме клас **Date**, който използва предефинираната операция ‘**++**’ в префиксна и постфиксна форма, за добавяне на 1 ден към датата, което може да доведе до увеличаване с 1-ца на месеца и годината.

#include <iostream>

using namespace std;

class Date{

friend ostream &operator<<(ostream &, const Date &);

public:

Date(int m=1, int d=1, int y=1900);

void setDate (int, int, int);

Date operator++();//префиксна форма

Date operator++(int); //постфиксна форма

/\* За да се предефинира операцията ‘**++**’ с възможност да се използва в префиксна или   
 постфиксна форма, трябва двете предефинирани операции да имат различни сигнатури за   
 да бъдат различавани от компилатора. При постфиксния вариант уникалната сигнатура се   
 постига с помощта на фиктивен аргумент от тип **int** . \*/

const Date &operator+=(int); //за добавяне на дни

int leapYear(int); //проверка за високосна година

int endОfMonth(int); //проверка за край на месец

private:

int month;

int day;

int year;

static int days[];//брой дни в отделните месеци

void helpIncrement();//обслужваща функция

};

int Date::days[]={0,31,28,31,30,31,30,31,31,30,31,30,31};

Date::Date(int m, int d, int y)

{ setDate(m, d, y);}

void Date::setDate(int mm, int dd, int yy)

{

month = (mm>1&&mm<=12) ? mm:1;

year = (yy>=1900&&yy<=2100) ? yy:1900;

if (month==2&&leapYear(year))

day = (dd>=1&&dd<=29) ? dd:1;

else

day = (dd>=1&&dd<=days[month]) ? dd:1;

}

Date Date::operator++()

{

helpIncrement();

return \*this;

}

/\* Променя се датата на текущия обект, активира се **copy constructor**-a на класа **Date** и се връща копие на текущия обект.

Префиксният вариант се дефинира както всяка унарна операция. Например в горната програма, ако **d1** е обект от клас **Date** и компилаторът срещне израза **++d1**, той ще генерира на негово място израза **d1.operator++()** . Префиксната форма на операцията '**++**' може да се реализира и с приятелска функция на класа, която се описва в декларацията на класа така **:**

**friend Date operator++(Date &);**

Тогава компилаторът заменя израза **++d1** с **operator++(d1)** \*/

Date Date::operator++(int)

{

Date temp = \*this;

helpIncrement();

return temp;

}

/\* Връща се непроменен, съхранен в **temp**, временен обект като стойност, а не като псевдоним, т.е. връща се непроменено копие на обекта от **temp**. Тази функция не може да върне псевдоним на обект **Date**, защото стойността, която трябва да се върне, се съхранява в локален обект в описанието на функцията, а локалните обекти се унищожават, когато функцията, в която са дефинирани, завърши изпълнението си. Така че, ако типа на връщания резултат е **Date&** , това би довело до връщане на псевдоним на обект, който вече не съществува.

**Връщането на & на локален обект е типична грешка, която се открива трудно.**

При постфиксния вариант, уникалната сигнатура на предефиниращата функция се постига с помощта на фиктивен аргумент от тип **int** . Например в горната програма, ако **d1** е обект от клас **Date** и компилаторът срещне израза **d1++**, той ще генерира на негово място израза **d1.operator++(0)** . Постфиксната форма на операцията '**++**' може да се реализира и с приятелска функция на класа, която се описва в декларацията на класа така **:**

**friend Date operator++(Date &, int);**

тогава компилаторът заменя израза **d1++** с **operator++(d1, 0)**

Разгледаното за ‘++’ важи и за ‘--‘ \*/

const Date &Date::operator+=(int additionalDays)

{

for(int i=1; i<=additionalDays; i++)

helpIncrement();

return \*this;//позволява слепване

}

int Date::leapYear(int y)

{

if(y%400==0||(y%100!=0&&y%4==0))

return 1;

else

return 0;

}

int Date::endОfMonth(int d)

{

if(month==2&&leapYear(year))

return d==29; // това е сравнение

else

return d==days[month]; // това е сравнение

}

void Date::helpIncrement()

{

if(endОfMonth(day)&&month==12)

{

day=1;

month=1;

++year;

}

else if(endОfMonth(day))

{

day=1;

++month;

}

else

++day;

}

ostream &operator<<(ostream &output, const Date &d)

{

static char \*monthName[13]={"","Януари","Февруари","Март",

"Април","Maй","Юни","Юли","Август","Септември","Октомври",

"Ноември","Декември"};

output<<monthName[d.month]<<' '<<d.day<<", "<<d.year;//конкатенация

return output;

}

void main()

{

Date d1, d2(12,27,1992), d3(0,99,8045);

cout<<"d1 = "<<d1<<endl

<<"d2 = "<<d2<<endl

<<"d3 = "<<d3<<endl;

cout<<"d2 += 7 е "<<(d2+=7)<<endl;

d3.setDate(2,28,1992);

cout<<"d3 = "<<d3<<endl;

cout<<"++d3 = "<<++d3<<endl;

Date d4(3,18,1969);

cout<<"d4 = "<<d4<<endl ;

cout<<"++d4 = "<<++d4<<endl;

cout<<"d4 = "<<d4<<endl;

cout<<"d4++ = "<<d4++<<endl;

cout<<"d4 = "<<d4<<endl;

}

Резултат от изпълнението на програмата **:**

d1 = Януари 1, 1900

d2 = Декември 27, 1992

d3 = Януари 1, 1900

d2 += 7 е Януари 3, 1993

d3 = февруари 28, 1992

++d3 = февруари 29, 1992

d4 = Март 18, 1969

++d4 = Март 19, 1969

d4 = Март 19, 1969

d4++ = Март 19, 1969

d4 = Март 20, 1969

**№ 17 Наследяване. Базови и производни класове**

Наследяването е начин за повторно използване на програмно осигуряване. Програмистът може да укаже, че създаден нов клас наследява данните-елементи и функциите-елементи на по-рано дефиниран клас. Наследяващият клас се нарича **производен клас**, а наследеният – **базов клас**. Всеки производен клас може да се използва като базов при създаването на нови производни класове.

При **просто наследяване**, производният клас наследява само един базов клас. При **множествено наследяване**, производният клас наследява два или повече базови класа.

В общия случай, производният клас добавя свои собствени данни-елементи и функции-елементи, така че обикновено той е по-голям от своя базов клас. В този смисъл, производният клас е по-специфичен по своето предназначение, в сравнение със своя базов клас и представя по-малка група от обекти. Основното предимство на наследяването е във възможността в производния клас да се определят добавки, замени или усъвършенствания на чертите, наследени от базовия клас.

При наследяване могат да се използват съществуващи библиотеки от класове. В този смисъл, значителна част от програмното осигуряване може да се конструира с използването на стандартизирани компоненти.

Производният клас няма достъп до закритите елементи на своя базов клас. Разрешаването на такъв достъп би нарушило енкапсулацията на базовия клас. Производният клас, обаче, има достъп до откритите и защитените елементи на своя базов клас.

Производният клас би могъл да има достъп до закритите елементи на своя базов клас, например чрез функции за достъп, предвидени в открития интерфейс на базовия клас.

Откритите елементи на базовия клас са достъпни за всички функции в програмата, закритите елементи на базовия клас са достъпни само за функциите-елементи и за приятелските функции на този клас. **Защитеното ниво на достъп** (**protected**) на базовия клас е междинно ниво между открития и закрития достъп. Защитените елементи на базовия клас са достъпни за елементите и приятелите на базовия клас и за елементите и приятелите на производния клас. В производния клас, този достъп се осъществява само с имената на тези елементи.

Базовият клас може да се наследява от производния като **public** (открито), **protected** (защитено) или **private** (закрито)**.** Защитеното и закритото наследяване се срещат рядко и те трябва да се използват много внимателно.

* при наследяване от тип **public**, откритите елементи на базовия клас стават открити елементи на производния клас и защитените елементи на базовия клас стават защитени елементи на производния клас;
* при наследяване от тип **protected**, откритите и защитените елементи на базовия клас стават защитени елементи на производния клас;
* при наследяване от тип **private**, откритите и защитените елементи на базовия клас стават закрити елементи на производния клас;

И при трите вида наследявания, производният клас няма достъп до закритите елементи на базовия клас.

Базовият клас може да бъде **пряк** или **косвен** базов клас на производния клас. Прекият базов клас явно се указва в дефиницията на производния клас, за разлика от косвения базов клас, който се наследява през две или повече нива в йерархията на класовете.

При открито наследяване (от тип **public**), обектите на производния клас могат да се разглеждат и като обекти на базовия клас. Това не е вярно при закрито или защитено наследяване. Например, нека при открито наследяване базовия клас да е четириъгълник, а производните класове да са квадрат, правоъгълник, успоредник и трапец. Вярно е твърдението, че квадрата, правоъгълника, успоредника и трапеца са четириъгълници. Обратното не е вярно. Например, четириъгълника не е квадрат. Обектите на базовия клас не могат автоматично да се разглеждат като обекти на производния клас. Програмистът може явно да преобразува типа на указател към базовия клас в тип на указател към производния клас. Типична грешка е този указател да се използва за достъп до несъществуващи елементи на производния клас.

Примерна програма **:**

#include <iostream>

#include <iomanip>

using namespace std;

class Point{

friend ostream &operator<<(ostream &, const Point &);

public:

Point(float=0, float=0);

void setPoint(float, float);

float getX()const {return x;}

float getY()const {return y;}

protected:/\* protected елементите могат да бъдат достъпвани директно

в производните класове \*/

float x, y;

};

Point::Point(float a, float b)

{ setPoint(a, b);}

void Point::setPoint(float a, float b)

{

x = a;

y = b;

}

ostream &operator<<(ostream &output, const Point &p)

{

output<<'['<<p.x<<", "<<p.y<<']';

return output;

}

/\* класът Circle открито наследява класа Point; тук ‘:’ е указание за

наследяване; ключовата дума public указва типа на наследяването \*/

class Circle:public Point{

friend ostream &operator<<(ostream &, const Circle &);

public:

Circle(float r=0.0, float x=0, float y=0);

void setRadius(float);

float getRadius()const;

float area()const;

protected:

float radius;

};

Circle::Circle(float r, float a, float b):Point(a,b)

{ radius = r;}

void Circle::setRadius(float r)

{ radius = r;}

float Circle::getRadius()const

{ return radius;}

float Circle::area()const

{ return 3.14\*radius\*radius;}

ostream &operator<<(ostream &output, const Circle &c)

{

output<<"Центърът = ["<<c.x<<", "<<c.y

<<"]; Радиусът = "<<setiosflags(ios::showpoint)

<<setprecision(2)<<c.radius;

return output;

}

void main()

{

Point \*pointPtr, p(3.5, 5.3);

Circle \*circlePtr, c(2.7, 1.2, 8.9);

cout<<"Point p: "<<p<<endl

<<"Circle c: "<<c<<endl;

pointPtr = &c;

cout<<\*pointPtr<<endl;

Point &pRef = c;

cout<<pRef<<endl;

circlePtr = (Circle \*)pointPtr;

cout<<\*circlePtr<<endl

<<"Лицето = "<<circlePtr->area()<<endl;

pointPtr = &p;

circlePtr = (Circle \*)pointPtr;//грешно

cout<<\*circlePtr<<endl

<<"Лицето = "<<circlePtr->area()<<endl;//ще изведе случайна стойност

}

Конструкторът на **Circle** извиква конструктора на **Point** за да зададе начални стойности на частта от обектите на класа **Circle**, която се отнася до базовия клас **Point**. Това става чрез списък от инициализатори на елементи. Ако конструкторът на **Circle** не активира конструктора на **Point**, то конструкторът на **Point** се активира със стойности по премълчаване. Ако класът **Point** няма конструктор с елементи по премълчаване, данните елементите на класа **Point** ще имат неопределени стойности.

Предефиниращата функция на операцията за извеждане в поток на класа **Circle** има непосредствен достъп до защитените данни елементи **x** и **y** на класа **Point**, тъй като тази функция е приятелска на производния клас **Circle**.

Указател към производния клас се присвоява винаги правилно на указател към базовия клас, тъй като обектите на производния клас могат да се разглеждат като обекти на базовия клас. Компилаторът извършва такова преобразуване неявно. Указателят към базовия клас вижда само тази част от производния клас, която се отнася към базовия клас. Такова присвояване може да стане и чрез псевдоним, например: **Point &pRef=c** – дефиниране на псевдоним на обект от базовия клас **Point**, който става псевдоним на обект от производния клас **Circle**.

Присвояването на указател към базовия клас на указател към производния клас е опасно и в този случай компилаторът не извършва неявно преобразуване – затова трябва да се използва явно преобразуване. Разглеждането на обект от клас **Point** като обект от клас **Circle** води до получаване на неопределена стойност за данната елемент **radius**. В общия случай, достъпът до несъществуващи данни-елементи не е опасен, но извикването на несъществуваща функция-елемент може да доведе до необратими грешки в програмата.

Възможно е с помощта на предефинирана операция за присвояване или конструктор за преобразуване на обект от производен клас да се присвои обект от съответен базов клас. Такова присвояване в общия случай би оставило неопределени собствените елементи на производния клас.

Резултат от изпълнението на програмата **:**

Point p: [3.5, 5.3]

Circle c: Центърът = [1.2, 8.9]; Радиусът = 2.7

[1.2, 8.9]

[1.2, 8.9]

Центърът = [1.2, 8.9]; Радиусът = 2.7

Лицето = 23.

Центърът = [3.5, 5.3]; Радиусът = 6.0e-039

Лицето = 0.00

**№ 18 Предефиниране на фунции-елементи от базовия клас в производния клас**

Производният клас може да предефинира функция-елемент на базовия клас. Ако в производния клас се опише функция-елемент със същото име както функция-елемент на базовия клас, то версията на тази функция в производния клас предефинира версията от базовия клас. За да се направи достъпна версията на функцията от базовия клас за производния клас, трябва да се използва бинарната операция за разрешаване на област на действие ‘**::**’.

Да разгледаме програмата, в която се използва клас **Employee**, който съдържа имената на служителите. Тази информация е обща за всички служители, включително и за тези от производния клас **HourlyWorker** (работещи на почасово заплащане).

Примерна програма **:**

#include <iostream>

#include <iomanip>

#include <string.h>

#include <assert.h>

using namespace std;

class Employee{

public:

Employee(const char \*, const char \*);

void print()const;

~Employee();

private:

char \*firstName;

char \*lastName;

};

Employee::Employee(const char \*first, const char \*last)

{

firstName = new char[strlen(first)+1];

assert(firstName!=0);

strcpy(firstName, first);

lastName = new char[strlen(last)+1];

assert(lastName!=0);

strcpy(lastName, last);

}

void Employee::print()const

{ cout<<firstName<<' '<<lastName;}

Employee::~Employee()

{

delete []firstName;

delete []lastName;

}

class HourlyWorker:public Employee{

public:

HourlyWorker(const char \*, const char \*, float, float);

float getPay()const;

void print()const;

private:

float wage; // заплащане на час

float hours; // брой заработени часове

};

HourlyWorker::HourlyWorker(const char \*first, const char \*last,

float initHours, float initWage)

:Employee(first, last)

{

hours = initHours;

wage = initWage;

}

/\* Връща изработената от служителя заплата \*/

float HourlyWorker::getPay()const

{ return wage\*hours;}

void HourlyWorker::print()const

{

Employee::print();

cout<<" – "<<'$'<<setiosflags(ios::fixed|ios::showpoint)

<<setprecision(2)<<getPay()<<endl;

}

/\* Функцията елемент **print()** на класа **HourlyWorker** е пример за предефинирана функция-елемент на базов клас в производен клас.

Функции-елементи от базовия клас често се предефинират в производния клас за изпълняване на някои по-специфични действия. Тези предефинирани функции понякога извикват версията на функцията от базовия клас, за да изпълнят част от новата задача. Функцията-елемент **print()** на производния клас **HourlyWorker** извиква функцията-елемент **print()** на базовия клас **Employee**. Тъй като имат еднакви имена и сигнатури, то за да се извика функцията-елемент **print()** на базовия клас, тя трябва да се предшества от **име\_на\_клас** и ‘**::**’ ( **Employee::print();** ) \*/

void main()

{

HourlyWorker h("Боб", "Смит", 40.0, 7.50);

h.print();

}

Базовият клас описва общите черти на обектите. Всички класове породени от базовия клас наследяват възможностите на базовия клас. В процеса на обектно-ориентираното проектиране, разработчика намира общите черти на обектите и ги представя под формата на базов клас. След това към производните класове се добавят допълнителни възможности в сравнение с наследеният базов клас. Полезно е да се наследяват свойства и методи от съществуващи класове, организирани във вид на библиотеки. Ако наследеният съществуващ клас има излишни функционални възможности, то те могат да се скрият в производните класове.

Резултат от изпълнението на програмата **:**

Боб Смит – $300.00

**№ 19 Използване на конструктори и деструктори в производни класове**

При създаване на обект от производен клас, конструкторът на производния клас винаги отначало извиква конструкторите на своите базовите класове, след това конструкторите на обектите-елементи на производния клас и най-накрая се изпълнява самият конструктор на производния клас. При унищожаване на обект от производен клас, деструкторите се извикват в ред, обратен на този в който са извикани съответните конструктори.

Редът, в който се извикват конструкторите на обектите-елементи, е редът, в който те са описани в декларацията на класа. На този ред не влияе последователността, в която са изброени инициализаторите на обектите елементи в заглавието на конструктора.

Редът, в който се извикват конструкторите на базовите класове, е редът, в който тези класове са посочени в заглавието на производния клас. На този ред не влияе последователността, в която са изброени инициализаторите на базовите класове в заглавието на конструктора на производния клас.

В конструктора на производния клас при явно извикване на конструкторите на базовите класове се използва списък от инициализатори на елементи. Ако такъв списък не се опише, конструкторът на производния клас неявно ще извика съответните конструктори с аргументи по премълчаване на базовите класове. Ако производният клас няма конструктор, то вграденият конструктор по премълчаване на производния клас ще извика конструкторите по премълчаване на базовите класове.

Примерна програма:

#include <iostream>

using namespace std;

class Point

{

public:

Point(float=0.0,float=0.0);

~Point();

protected:

float x, y;//защитени елементи

};

Point::Point(float a, float b)

{

x = a;

y = b;

cout<<"Конструктор на Point: "

<<'['<<x<<", "<<y<<']'<<endl;

}

Point::~Point()

{

cout<<"Деструктор на Point: "

<<'['<<x<<", "<<y<<']'<<endl;

}

class Circle:public Point{ // класа Circle явно наследява класа Point

public:

Circle(float r=0.0, float x=0, float y=0);

~Circle();

private:

float radius;

};

Circle::Circle(float r, float a, float b):Point(a,b)/\* конструктора на Circle извиква конструктора на базовия клас \*/

{

radius = r;

cout<<"Конструктор на Circle: "

<<radius

<<"[ "<<a<<", "<<b<<']'<<endl;

}

Circle::~Circle()

{

cout<<"Деструктор на Circle: "

<<radius

<<"[ "<<x<<", "<<y<<']'<<endl;

}

void main ()

{

{ Point p(1.1, 2.2);}/\*фигурните скобки определят област на действие,   
 затова най-напред се извиква конструктора на **Point**,   
 а след това неговия деструктор \*/

cout<<endl;

Circle circle1(4.5, 7.2, 2.9);/\* активира се конструктора на Circle,   
 изпълнява се конструктора на Point и след   
 това продължава изпълненито на конструктора   
 на Circle \*/

cout<<endl;

Circle circle2(10, 5, 5); /\* активира се конструктора на Circle,   
 изпълнява се конструктора на Point и след   
 това продължава изпълненито на конструктора   
 на Circle \*/

cout<<endl;

}

Резултат от изпълнението на програмата **:**

Конструктор на Point: [1.1, 2.2]

Деструктор на Point: [1.1, 2.2]

Конструктор на Point: [7.2, 2.9]

Конструктор на Circle: 4.5[ 7.2, 2.9]

Конструктор на Point: [5, 5]

Конструктор на Circle: 10[ 5, 5]

Деструктор на Circle: 10[ 5, 5]

Деструктор на Point: [5, 5]

Деструктор на Circle: 4.5[ 7.2, 2.9]

Деструктор на Point: [7.2, 2.9]

Конструкторите и операцията за присвояване не се наследяват от производните класове. Въпреки това, конструкторите и операцията за присвояване на производните класове могат да извикват конструкторите и операцията за присвояване на базовите класове.

**№ 20 Множествено наследяване**

При простото наследяване, всеки производен клас се поражда само от един базов клас. Йерархията при простото наследяване може да се изобрази чрез дърво. При множественото наследяване, производният клас се поражда от два или повече базови класа. Йерархията при множественото наследяване може да се изобрази чрез ориентиран ацикличен граф. Йерархията на наследяването може да бъде с произволна дълбочина в границите на физическите ограничения за конкретната система. Множественото наследяване се указва с ‘**:**’ в дефиницията на производния клас и след това изброени базовите класове разделени със запетая.

Примерна програма **:**

#include <iostream>

using namespace std;

class Base1{

public:

Base1(int x) { value = x;}

int getData()const {return value;}

protected:

int value;

};

class Base2{

public:

Base2(char c) {letter = c;}

char getData()const { return letter;}

protected:

char letter;

};

class Derived: public Base1, public Base2{ //множествено наследяване

friend ostream &operator<<(ostream &, const Derived &);

public:

Derived(int, char, float);

float getReal() const;

private:

float real;

};

/\* Конструкторът на **Derived** извиква конструкторите на всеки от своите базови класове  
 ( **Base1** и **Base2** ) в реда, в който те са посочени в заглавието на производния клас **Derived**, а не в реда в който са изброени инициализаторите на базовите класове в заглавието на конструктора на производния клас \*/

Derived::Derived(int i, char c, float f):Base1(i), Base2(c)

{ real = f;}

float Derived::getReal()const

{return real;}

/\* Функцията '**<<**' е приятелска за **Derived**, затова има пряк достъп до **real**, но функцията '**<<**' е приятелска на производен клас и затова има пряк достъп до защитените данни-елементи **value** и **letter** на наследените базови класове **Base1** и **Base2** \*/

ostream &operator<<(ostream &output, const Derived &d)

{

output<<" цяло: "<<d.value<<endl

<<"символ: "<<d.letter<<endl

<<"реално: "<<d.real<<endl;

return output;

}

void main()

{

Base1 b1(10), \*base1Ptr;

Base2 b2('z'), \*base2Ptr;

Derived d(7, 'A', 3.5);

cout<<b1.getData()<<endl // 10

<<b2.getData()<<endl // z

<<d<<endl; /\* цяло: 7

символ: A

реално: 3.5

\*/

cout<<d.Base1::getData()<<endl /\* заради еднаквите имена и сигнатури на   
 методите се използва операция за   
 разрешаване на област на действие **::** \*/

<<d.Base2::getData()<<endl

<<d.getReal()<<endl;

base1Ptr = &d; /\* разглеждаме обект от клас **Derived**   
 като обект от клас **Base1** \*/

cout<<base1Ptr->getData()<<endl;/\* няма да има неопределеност, тъй като се   
 взема **getData()** дефинирана в **Base1** и   
 наследена от **Derived** \*/

base2Ptr = &d; /\* разглеждаме обект от клас **Derived**   
 като обект от клас **Base2** \*/

cout<<base2Ptr->getData()<<endl;/\* няма да има неопределеност, тъй като се   
 взема **getData()** дефинирана в **Base2** и   
 наследена от **Derived** \*/

}

Резултат от изпълнението на програмата **:**

10

z

цяло: 7

символ: A

реално: 3.5

7

A

3.5

7

A

Множественото наследяване дава големи възможности, но може да предизвика различни проблеми при неправилно използване. За да се извлече полза от множественото наследяване системите трябва да се проектират много внимателно. Не се препоръчва да се използва множествено наследяване в случаите, когато можем да се ограничим с просто наследяване.

**№ 21 Виртуални функции и полиморфизъм**

С помощта на виртуални функции и полиморфизъм, могат да се разработват и реализират програми за обобщена обработка на обектите от всички съществуващи в йерархията класове, като обекти от базовия клас. Ако по време на разработката на такава програма трябва да се добави нов клас в йерархията, това може да стане с незначителни изменения на самата програма.

Един от начините за специфична обработка на обекти от различни класове е използването на оператор **switch**. Във връзка с това, обаче, възникват различни проблеми – например липса на проверка за типовете, на проверка за всички възможни случаи. Освен това, добавянето или изключването на клас води до промяна на съответния оператор **switch**.

От друга страна, програмистът може да използва механизма на виртуалните функции за автоматично изпълнение на логиката на оператора **switch**, избягвайки горепосочените проблеми и неудобства. Използването на виртуални функции и полиморфизъм води до създаване на програми, които имат по-малко логически разклонения и по този начин се улеснява проверката и съпровождането на тези програми.

Да предположим, че редицата класове за форми, такива като **Circle**, **Triangle**, **Square** и т.н. са производни от базовия клас **Shape**(форми). Всеки клас може да има своя собствена функция **Draw**. В този случай е удобно при рисуване на всяка форма да можем да извикваме функцията **Draw** за базовия клас **Shape** и тогава програмата динамично, т.е. по време на своето изпълнение, да определи коя от функциите **Draw** на съответния производен клас да се изпълни. За да се създаде такава възможност, трябва функцията **Draw** да се декларира като виртуална в базовия клас и след това да се предефинира във всеки от производните класове, така че да рисува съответната форма.

Една функция се декларира като виртуална с помощта на ключовата дума **virtual**, записана преди прототипа на функцията в базовия клас. Например в базовия клас можем да запишем **:**

**virtual void Draw() const;**

Ако една функция веднъж е декларирана като виртуална, то тя остава виртуална на всяко по-ниско ниво в йерархията. Някои програмисти обаче, предпочитат да декларират явно функцията като виртуална на всяко ниво. Ако производният клас няма собствена реализация на виртуалната функция, то се използва реализацията, описана в базовия клас. Ако функцията **Draw** от базовия клас е декларираната като **virtual** и ако след това тя се извика чрез указател от базовия клас, сочещ към обект от производния клас, например :

**shapePtr -> Draw();**

то програмата динамично, т.е. по време на изпълнение, ще избере съответната функция от производния клас. Това се нарича **динамично свързване**.

Предефинирата виртуална функция в производния клас трябва да има същия тип на резултата и същата сигнатура, както виртуалната функция в дефиницията на базовия клас. В противен случай се дава съобщение за синтактична грешка.

Ако виртуална функция се извиква чрез обръщение по име и при това се използва операцията за достъп до елемент ‘**.**’, например **:**

**squareObject.Draw();**

то извикването се обработва по време на компилация и това се нарича **статично свързване** и се избира функцията **Draw** от класа на дадения обект или който е наследил този клас.

Полиморфизмът е възможност за обектите от различни класове, свързани с помощта на наследяване, да реагират по различен начин при обръщение към една и съща функция-елемент. Полиморфизмът се реализира с помощта на виртуални функции.

Ако при използване на виртуална функция, заявката се осъществи с помощта на указател от базовия клас или псевдоним, то **C++** избира нужната предефинирана функция от съответния производен клас свързан с дадения обект. Понякога функция-елемент, която не е дефинирана като виртуална в базовия клас се предефинира в производния клас. Ако такава функция-елемент е извикана чрез указател от базовия клас, то се използва версията на функцията в базовия клас, ако е извикана чрез указател от производния клас, то се използва версията на функцията от производния клас. Това не е полиморфно поведение. Например **:**

имаме базов клас **Employee** и производен клас **HourlyWorker**

**HourlyWorker h, \*hPtr = &h;** //дефинираме обект и указател от производния клас

**Employee \*ePtr = &h;** //дефинираме указател от базовия клас сочещ към обект от производния клас

Ще напомним, че функцията-елемент **print** е невиртуална, дефинирана в базовия клас и предефинирана в производния клас.

**hPtr -> print();** //извиква се функцията-елемент **print** от производния клас

**ePtr -> print();** //извиква се функцията-елемент **print** от базовия клас

Причината, че това не е полиморфно поведение – функциите не са виртуални и имат еднаква сигнатура. Ако функцията **print** беше дефинирана като **virtual** в базовия клас, то при обръщението

**ePtr -> print();**

щеше да се извика функцията-елемент **print** от производния клас.

По време на компилация не е необходимо да се знае типа на обекта, за да се генерира извикване на виртуална функция. По време на изпълнението на програмата, чрез динамично свързване, извикването на виртуална функция-елемент се преобразува в извикване на вариант на виртуалната функция от съответния клас. За целта се използва **таблица на виртуалните методи**. Всеки клас, който съдържа виртуални функции има своя таблица на виртуалните методи. За всяка виртуална функция-елемент от класа, таблицата има елемент, съдържащ указател към вариант на виртуалната функция, който се използва с обектите от дадения клас. Всеки обект от клас, който съдържа виртуални функции, има указател към таблицата на виртуалните методи на този клас. При динамичното свързване, чрез този указател се отива в таблицата на виртуалните методи и там се намира нужният указател към вариант на виртуалната функция от съответния клас. Реализацията на динамичното свързване изисква минимални разходи на време за изпълнение.

**№ 22 Абстрактни базови класове и конкретни класове**

Има случаи, в които е полезно да се дефинират класове, за които програмистът няма да създава обекти. Такива класове се наричат **абстрактни класове**. Тъй като те се използват като базови класове в процеса на наследяване, обикновено се наричат **абстрактни базови класове**. Обектите на абстрактните базови класове не могат да бъдат реализирани, т.е. не съществуват обекти от абстрактни базови класове. Единственото предназначение на абстрактните базови класове е да могат да предават интерфейс и реализация на производните от тях класове. Класове, чиито обекти могат да бъдат реализирани се наричат **конкретни класове**. Например, можем да имаме абстрактен базов клас **TwoDimensionalObject** и производни конкретни класове **Square**, **Circle**, **Triangle** и т.н. Абстрактните базови класове са твърде общи за дефиниране на реални обекти. За това са предназначени конкретните класове – те притежават необходимата определеност и специфика за дефиниране на реални обекти.

Един клас става абстрактен чрез дефиниране на една или повече виртуални функции като **чисто виртуални**. Една виртуална функция става чисто виртуална, когато в нейната декларацията тялото е дефинирано като 0, т.е. инициализатор = 0. например **:**

**virtual float earnings() const = 0;**

Ако един клас е производен от клас с чисто виртуална функция и ако тази чисто виртуална функция не е реализирана в този производен клас, тогава функцията остава чисто виртуална и производният клас автоматично става абстрактен.

При опит за създаване на обект от абстрактен клас се дава съобщение за синтактична грешка.

Абстрактният базов клас определя интерфейса за различни типове обекти в йерархията на класовете. Всички обработки в йерархията могат да прилагат един и същи интерфейс, използвайки полиморфизъм – дефинират се указатели от абстрактния базов клас и след това те се използват за полиморфно опериране с обектите на производните конкретни класове.

Полиморфизмът е особено ефективен при реализацията на системно програмно осигуряване. Например при операционните системи всеки тип физическо устройство може да работи по съвършено различен начин. Независимо от това, командите за четене ‘**read**’ и командите за писане ‘**write**’ могат да бъдат подобни. Например съобщението ‘**write**’, изпратено на обект драйвер за устройство се интерпретира по различен начин в зависимост от типа на драйвера и от това по какъв начин той управлява съответното устройство. Въпреки това, извикването ‘**write**’ е почти едно и също за различните устройства – при това извикване просто се копират определен брой байтове от паметта в конкретното устройство. Обектно-ориентираните операционни системи могат да използват абстрактни базове класове за да се реализира интерфейс, пригоден за драйверите на всички устройства. След това, с помощта на наследяване, тези абстрактни класове могат да образуват производни класове. Откритият интерфейс към драйверите на устройствата се предава с помощта на чисто виртуални функции. Реализацията на тези виртуални функции в производните класове съответства на конкретните типове драйвери за устройства.

Полиморфизмът се използва и при класове итератори – например при обхождане в свързан списък на обекти от различни нива в йерархията. Всички указатели в такъв списък могат да бъдат указатели от базовия клас.

Конструкторите не могат да бъдат виртуални. Типична грешка е дефинирането на конструктор като виртуална функция.

При използването на полиморфизъм за обработка на динамично разпределени обекти може да възникне един проблем – ако обекта се унищожава с явно използване на операцията **delete**, приложена към указател от базовия клас, сочещ към този обект, то в случая се извиква деструктора на базовия клас за този обект, независимо от типа на обекта. Съществува просто разрешение на проблема – деструкторът от базовия клас се дефинира като виртуален и това автоматично прави виртуални деструкторите на производните класове, въпреки че те имат имена, различни от името на деструктора на базовия клас. По този начин операцията **delete**, приложена към указател от базовия клас, сочещ към обект от производен клас, ще активира деструктора на производния клас. Както знаем, когато обект от производен клас се унищожава, след изпълнение на деструктора на производния клас се извиква деструктора на базовия клас за да се унищожи частта на обекта от базовия клас.

Ако един клас има виртуални функции, добър стил на програмиране е създаването на виртуален деструктор, дори той да не е необходим.

Благодарение на виртуалните функции и полиморфизма, програмистът може да управлява широк спектър от обекти дори без да знае какъв е техният тип. Управлението по време на изпълнението на програмата автоматично ще отчита спецификата на тези обекти.

**№ 23 Пример за използване на виртуални функции и полиморфизъм**

В примера ще се пресмята заплатата на служители, като се отчита типа на служителя. Базовият клас е **Employee** (служители), производните класове са :

* клас **Boss** – служител с фиксирана заплата;
* клас **CommissionWorker** - служител с базова заплата и комисионен процент
* клас **PieceWorker** – служител със заплата, пропорционална на изработеното количество
* клас **HourlyWorker** – служител със заплата, пропорционална на часовете за работа

Функцията **earnings()**  (изчислява заплатата) се използва за всички служители, но начинът за пресмятане на заплатата е различен в зависимост от типа на служителя, затова тя се декларира като виртуална в базовия клас, а след това е реализирана във всеки от производните класове. За да се пресметне заплатата на данен служител, в програмата се използва указател от базовия клас, който сочи към обекта на конкретния служител при извикване на функцията earnings().

Програма **:**

#include <iostream>

#include <iomanip>

#include <string.h>

#include <assert.h>

using namespace std;

class Employee{

public:

Employee(const char \*, const char \*);//име и фамилия

~Employee();//освобождава динамично разпределена памет

const char \*getFirstName()const;//чете името

const char \*getLastName()const;//чете фамилията

virtual float earnings()const=0;//чисто виртуална функция за заплатата

virtual void print()const=0;//чисто виртуална функция

private:

char \*firstName;

char \*lastName;

};

Декларацията на виртуалните функции **earnings()** и **print()** като **чисто виртуални** е с цел те да се обезсмислят за класа **Employee**, защото не може да се пресмята и извежда заплатата на абстрактен служител. Те ще бъдат реализирани в производните класове, т.е. за определен тип служител.

Employee::Employee(const char \*first, const char \*last)

{

firstName = new char[strlen(first)+1];

assert(firstName!=0);

strcpy(firstName, first);

lastName = new char[strlen(last)+1];

assert(lastName!=0);

strcpy(lastName, last);

}

/\* Конструкторът динамично разпределя памет за името и фамилията и използва **strcpy** за тяхното деклариране в обекта. \*/

Employee::~Employee()

{

delete []firstName;

delete []lastName;

}

/\* Деструкторът освобождава динамично разпределената от конструктора памет \*/

const char \*Employee::getFirstName()const

{ return firstName;}

const char \*Employee::getLastName()const

{ return lastName;}

Функциите **getFirstName** и **getLastName** връщат константни указатели към закритите данни-елементи, което не позволява на клиентите да модифицират тези данни. За да се избегне получаването на неопределен указател, клиентът трябва да прекопира получените низове, преди те да бъдат унищожени от деструктора на класа.

class Boss:public Employee

{

public:

Boss(const char \*, const char \*, float=0.0);

void setWeeklySalary(float);

virtual float earnings()const;

virtual void print()const;

private:

float weeklySalary;

};

Boss::Boss(const char \*first, const char \*last, float s):Employee(first, last)

{ setWeeklySalary(s);}

void Boss::setWeeklySalary(float s)

{ weeklySalary = s>0 ? s:0;}

float Boss::earnings()const

{ return weeklySalary;}

void Boss::print()const

{

cout<<endl<<" Постоянна заплата: "

<<getFirstName()<<' '<<getLastName();

}

class CommissionWorker:public Employee

{

public:

CommissionWorker(const char \*, const char \*,

float=0.0, float=0.0, unsigned=0);

void setSalary(float);

void setCommission(float);

void setQuantity(unsigned);

virtual float earnings()const;

virtual void print()const;

private:

float salary;

float commission;

unsigned quantity;

};

CommissionWorker::CommissionWorker(const char \*first, const char \*last,

float s, float c, unsigned q):Employee(first, last)

{

salary = s>0 ? s:0;

commission = c>0 ? c:0;

quantity = q>0 ? q:0;

}

void CommissionWorker::setSalary(float s)

{ salary = s>0 ? s:0;}

void CommissionWorker::setCommission(float c)

{ commission = c>0 ? c:0;}

void CommissionWorker::setQuantity(unsigned q)

{ quantity = q>0 ? q:0;}

float CommissionWorker::earnings( ) const

{ return salary + commission \* quantity;}

void CommissionWorker::print()const

{ cout<<endl<< " Комисионна заплата: "

<<getFirstName()<<' '

<<getLastName();

}

class PieceWorker:public Employee{

public:

PieceWorker(const char \* , const char \*, float=0.0, unsigned=0) ;

void setWage(float);

void setQuantity(unsigned);

virtual float earnings()const;

virtual void print()const;

private:

float wagePerPiece;

unsigned quantity;

};

PieceWorker::PieceWorker(const char \*first , const char \*last,

float w, unsigned q):Employee(first, last)

{

wagePerPiece = w>0 ? w:0;

quantity = q>0 ? q:0;

}

void PieceWorker::setWage(float w)

{ wagePerPiece = w>0 ? w:0;}

void PieceWorker::setQuantity(unsigned q)

{ quantity = q>0 ? q:0;}

float PieceWorker::earnings()const

{ return quantity \* wagePerPiece;}

void PieceWorker::print()const

{

cout<<endl<<"Пpouзвoдcтвена заплатa: "

<<getFirstName()<<' '

<<getLastName();

}

class HourlyWorker:public Employee{

public:

HourlyWorker(const char \*, const char \*, float=0.0, float=0.0);

void setWage(float);

void setHours(float);

virtual float earnings()const;

virtual void print()const;

private:

float wage;

float hours;

};

HourlyWorker::HourlyWorker(const char \*first, const char \*last,

float w, float h):Employee(first, last)

{

wage = w>0 ? w:0;

hours = h>0&&h<168 ? h:0;

}

void HourlyWorker::setWage(float w)

{ wage = w>0 ? w:0;}

void HourlyWorker::setHours (float h)

{ hours = h>0&&h<168 ? h:0;}

float HourlyWorker::earnings()const

{return hours > 40 ? 40\*wage+1.5\*(hours-40)\*wage : hours\*wage;}

void HourlyWorker::print()const

{

cout<<endl<<" Почacoвa заплатa: "

<<getFirstName()<<' '<<getLastName();

}

void main()

{

cout<<setiosflags(ios::showpoint)<<setprecision(2);

Employee \*ptr;

Boss b("Иван", "Христов", 800.00) ;

ptr = &b;

ptr->print();//това е динамично свързване

cout<<" $"<<ptr->earnings();//динамично свързване

b.print();//това е статично свързване

cout<<" $"<<b.earnings();//статично свързване

CommissionWorker c("Еленa", "Кирова", 200.0, 3.0, 150);

ptr = &c;

ptr->print();

cout<<" $"<<ptr->earnings();

c.print();

cout<<" $"<<c.earnings();

PieceWorker p("Иван", "Колев", 2.5, 200);

ptr = &p;

ptr->print();

cout<<" $"<<ptr->earnings();

p.print();

cout<<" $"<<p.earnings();

HourlyWorker h("Анa", "Попова", 13.75, 40);

ptr = &h;

ptr->print();

cout<<" $"<<ptr->earnings();

h.print();

cout<<" $"<<h.earnings();

cout<<endl;

}

Резултат от изпълнението на програмата **:**

Постоянна заплата: Иван Христов $800.00

Постоянна заплата: Иван Христов $800.00

Комисионна заплата: Еленa Кирова $650.00

Комисионна заплата: Еленa Кирова $650.00

Пpouзвoдcтвена заплатa: Иван Колев $500.00

Пpouзвoдcтвена заплатa: Иван Колев $500.00

Почacoвa заплатa: Анa Попова $550.00

Почacoвa заплатa: Анa Попова $550.00

**№ 24 Пример за наследяване на интерфейса и неговата реализация**

Ще разгледаме йерархията на формите **Point**, **Circle** и **Cylinder** (точка, кръг и цилиндър ). Допълнително във върха на йерархията ще създадем абстрактен базов клас **Shape**, който съдържа четири виртуални функции – две **чисто виртуални** за отпечатване на формите (**printShapeName** и **print**)и две виртуални за пресмятане на лицето и на обема на формите (**area** и **volume**), реализирани да връщат стойност 0. Например класът **Point** не предефинира функциите за лице и обем, тъй като те имат стойност 0 за точка. Клас **Circle** наследява функцията **volume** от клас **Point**, но има собствена реализация на функцията **area()**, а класа **Cylinder** има собстевена реализация на функците **area()** и **volume()**. Ще отбележим, че макар класа **Shape** да е абстрактен базов клас, той съдържа реализацията на някои функции и тези реализации се наследяват. Класът **Shape** предоставя за наследяване интерфейс във вид на четири виртуални функции, които ще влизат на всички нива в йерархията.   
**В примера се акцентира върху това, че класовете могат да наследяват и интерфейса и реализацията на базовия клас.**

**В йерархия, проектирана за наследяване на реализация, стремежът е да се осигури функционална възможност на възможно по-високо ниво за да може всеки нов производен клас да наследява функции-елементи, описани в базовия клас и да използва тези описания.**

**В йерархия, проектирана за наследяване на интерфейс, стремежът е да се осигури функционална възможност на възможно по-ниско ниво.** В този случай базовият клас декларира функции, които трябва да се извикват идентично за всеки обект в йерархията, т.е. извикванията да имат еднаква сигнатура. Производните класове сами осигуряват реализации на тези функции.

Пример **:**

#include <iostream>

#include <iomanip>

using namespace std;

class Shape{

public:

virtual float area()const {return 0.0;}

virtual float volume()const {return 0.0;}

virtual void printShapeName()const = 0;

virtual void print()const = 0;

};

Функциите **area** (лице) и **volume** (обем) са описани в абстрактния базов клас **Shape** и те връщат стойност нула. Тези функции се предефинират в онези производни класове, в които лицето и обема имат смисъл и **area()** и **volume()** трябва да се използват за съответни пресмятания на лицето и обема.

class Point:public Shape{

friend ostream &operator<<(ostream &, const Point &);

public:

Point(float=0, float=0);

void setPoint(float, float);

float getX()const {return x;}

float getY()const {return y;}

virtual void printShapeName()const { cout <<"Toчкa: ";}

virtual void print()const;

private:

float x,y;

};

Point::Point(float a, float b)

{ setPoint(a, b);}

void Point::setPoint(float a, float b)

{

x = a;

y = b;

}

void Point::print()const

{ cout<<'['<<x<<", "<<y<<']';}

ostream &operator<<(ostream &output, const Point &p)

{

p.print();

return output;

}

Производният клас **Point** открито наследява класа **Shape**. Точката няма лице и обем и затова реализациите на функциите-елементи **area()** и **volume()** се наследяват от базовия клас **Shape**. Функциите **printShapeName()** и **print()** са реализации на виртуалните функции, които бяха описани в базовия клас **Shape** като чисто виртуални. Други функции-елементи са функциите **:**

* **setPoint** – съхранява координатите на точка в данните-елементи **х** и **у**
* **getPoint** – връща координатите на точка съхранени в данните-елементи **х** и **у**

class Circle:public Point

{

friend ostream &operator<<(ostream &, const Circle &);

public:

Circle(float r=0.0, float x=0.0, float y=0.0);

void setRadius(float);

float getRadius()const;

virtual float area()const;

virtual void printShapeName()const { cout<<"Kpъг: ";}

virtual void print()const;

private:

float radius;

};

Circle::Circle(float r, float a, float b):Point(a,b)

{ radius = r>0?r:0;}

void Circle::setRadius(float r)

{ radius = r>0?r:0;}

float Circle::getRadius()const

{ return radius;}

float Circle::area()const

{ return 3.14159\*radius\*radius;}

void Circle::print()const

{

cout<<'['<<getX()<<", "<<getY()

<<"]; Радиус = "<<setiosflags(ios::fixed | ios::showpoint)

<<setprecision(2)<<radius;

}

ostream &operator<<(ostream &output, const Circle &c)

{

c.print();

return output;

}

Производният клас **Circle** открито наследява класа **Point**. Кръгът няма обем, затова фунцията-елемент **volume()** се наследява от базовия клас **Shape**, посредством класа **Point**. Кръгът има лице, затова функцията-елемент **area()** е предефинирана в класа **Circle**. Функциите-елементи **printShapeName()** и **print()** са реализации на виртуалните функции, които бяха декларирани в базовия клас **Shape**, като чисто виртуални. Ако те не бяха дефинирани в класа **Circle**, то щяха да се наследят техните версии от класа **Point**. Други функции-елементи са функцията **setRadius**, която присвоява нова стойност на радиуса и **getRаdius**, която връща стойността на радиуса.

class Cylinder:public Circle{

friend ostream &operator<<(ostream &, const Cylinder &);

public:

Cylinder(float h=0.0, float r=0.0, float x=0.0, float y=0.0);

void setHeight(float);

float getHeight()const { return height;}

virtual float area()const;

virtual float volume()const;

virtual void printShapeName()const { cout<<"Цилиндър: ";}

virtual void print()const;

private:

float height;

};

Cylinder::Cylinder(float h, float r, float x, float y):Circle(r, x, y)

{ height = h>0?h:0;}

void Cylinder::setHeight(float h)

{ height = h>0?h:0;}

float Cylinder::area()const

{ return 2\*Circle::area()+2\*3.14159\*getRadius()\*height;}

float Cylinder::volume()const

{ return Circle::area()\*height;}

void Cylinder::print()const

{

Circle::print();

cout<<"; Височина = "<<height;

}

ostream &operator<<(ostream &output, const Cylinder &c)

{

c.print();

return output;

}

Производният клас **Cylinder** открито наследява класа **Circle**. Цилиндъра има лице и обем, затова и двете функции-елементи **area()** и **volume()** са предефинирани, a функциите-елементи **printShapeName()** и **print()** са реализации на виртуалните функции, които бяха декларирани в базовия клас **Shape**, като чисто виртуални. Други функции-елементи са функцията **setHeight**, която присвоява стойност на височината на цилиндъра и **getHeight**, която връща стойността на височината.

void main()

{

//--- статично свързване ---

Point point(7, 11);

Circle circle(3.5, 22, 8);

Cylinder cylinder(10, 3.3, 10, 10);

point.printShapeName();

cout<<point<<endl;

circle.printShapeName();

cout<<circle<<endl;

cylinder.printShapeName();

cout<<cylinder<<"\n\n";

//--- динамично свързване ---

Shape \*arrayOfShapes[3];

arrayOfShapes[0] = &point;

arrayOfShapes[1] = &circle;

arrayOfShapes[2] = &cylinder;

for(int i=0; i<3; i++)

{

arrayOfShapes[i]->printShapeName();

cout<<endl;

arrayOfShapes[i]->print();

cout<<"\nЛицето = "<<arrayOfShapes[i]->area()

<<"\nОбемьт = "<<arrayOfShapes[i]->volume()

<<endl<<endl;

}

/\* чрез динамично свързване се активират виртуалните

функции за i–тия елемент на масива \*/

}

Резултат от изпълнението на програмата **:**

Toчкa: [7, 11]

Kpъг: [22, 8]; Радиус = 3.50

Цилиндър: [10.00, 10.00]; Радиус = 3.30; Височина = 10.00

Toчкa:

[7.00, 11.00]

Лицето = 0.00

Обемьт = 0.00

Kpъг:

[22.00, 8.00]; Радиус = 3.50

Лицето = 38.48

Обемьт = 0.00

Цилиндър:

[10.00, 10.00]; Радиус = 3.30; Височина = 10.00

Лицето = 275.77

Обемьт = 342.12

**Задача 1.**  Да се напише клас **CPoint(точка),** който има следните методи **:**

* Конструктор
* Конструктор с параметри
* Set метод
* Get методи за член-променливите

**Задача 2** Да се напише клас **CComplex(комплексни числа),** който има следните методи:

* Конструктор без параметри и конструктор с параметри
* Методи за събиране на комплексни числа с тип на връщания резултат **void** и псевдоним на обекта
* Методи за изваждане на комплексни числа с тип на връщания резултат **void** и псевдоним на обекта
* Методи за умножение на комплексни числа с тип на връщания резултат **void** и псевдоним на обекта
* Метод за намиране на модул на комплесно число
* Get метод за член променливите
* Метод за извеждане на комплексно число

**Задача 3** Да се напише клас **CRat(рационални числа)**, който да предефинира следните операции:

* Въвеждане от поток и извеждане в поток
* Унарен ''+'' (плюс) и ''-'' (минус)
* Бинарен ''+'', ''-'', ''\*'' (умножение) и ''/'' (деление)
* Оператори за сравнение ''= = '' (равно), ''! = '' (различно), ''<'' и ''> '' (по-малко и по-голямо)

Дробите да са представени в нормализиран вид.

**Задача 4** Да се напише клас за тримерни вектори **CVector**, който да има предефинирани следните операции:

* Въвеждане от поток и извеждане в поток
* Унарен ''+'' (плюс) и ''-'' (минус)
* Бинарен ''+'', ''-'', ''\*'' (умножение) и ''/'' (деление)
* Оператори за сравнение ''= = '' (равно), ''! = '' (различно), ''<'' и ''> '' (по-малко и по-голямо)
* Оператори за умножение на вектор с число и число с вектор

**Задача 5** Да се напише клас дата **CDate**, който да има предефинирани следните операции:

* Въвеждане от поток и извеждане в поток
* Оператор за присвояване ''=''
* Бинарен ''+'' (прибавят се дни ) и ''-'' (изваждат се дни)
* Оператори за сравнение ''= = '' (равно), ''! = '' (различно), ''<'' и ''> '' (по-малко и по-голямо)
* Префиксни и постфиксни '' ++ '' (прибавяне на единица), '' - - '' (изваждане на единица)

**Задача 6** Да се напише програма, която извършва операции с вектори, представени чрез техните координати. За целта да се изпълнят слените изисквания **:**

* Да се декларира клас **CPoint**, който описва точка с координати **х** и **у**. Да се напише съответната реализация на конструктора на класа и константен метод **void print()** за извеждане на координатите на точка;
* Да се декларира клас **CVector** за работа с вектори, който наследява клас **CPoint**. Да се напише съответната реализация на конструктора на класа, да се предефинират оператори за събиране, изваждане и умножение на вектори и константен метод **void print()** за извеждане на координатите на вектор;
* Да се дефинират обекти от класовете **CPoint** и **CVector** и указатели към тези обекти. Да се изведат координатите на вектор, който е сбор на два вектора. Да се изведат координатите на точка като се използват указателите.

**Задача 7** Да се напише програма, в която **:**

* Да се декларира клас **CPerson**, който описва човек с име и възраст. Да се дефинира конструктор на класа, деструктор, константен метод **void print()** за извеждане на име и възраст на обект от клас **CPerson** и да се предефинира оператор за извеждане на обект от клас **CPerson**;
* Да се декларира клас **CCar**, който описва кола с нейната марка и година на производство и наследява клас **CPerson**. Да се дефинира конструктор на класа, деструктор, константен метод **void print()** за извеждане на марката и годината на производтво на кола и име и възраст на нейния собственик (обект от клас **CPerson**) и да се предефинира оператор за извеждане на обект от клас **CCar**;
* Да се дефинират обекти от класовете **CPerson** и **CCar**. Да се изведат тези обекти като се използват и двата възможи начина.

**Задача 8** Да се напише програма, в която **:**

* Да се декларира клас **CPerson**, който описва човек с име и възраст. Да се дефинира конструктор на класа, деструктор, константен метод **void print()** за извеждане на име и възраст на обект от клас **CPerson**;
* Да се декларира клас **CStudentID**, който описва студент с неговия факултетен номер. Да се дефинира конструктор на класа, константен метод **void print()** за извеждане на факултетния номер на студент;
* Да се декларира клас **CStudent**, който описва студент чрез съответния му курс и наследява класовете **CPerson** и **CStudentID**. Да се дефинира конструктор на класа, констатнен метод **void print()** за извеждане на име, възраст, факултетен номер и курс на студента;
* Да се дефинира обект от клас **CStudent**. Да се изведат име, възраст, факултетен номер и курс на студента.

**Задача 9**  Да се направи клас **CFigure**. Да се направят класове **CTriangle**, **CRectangle** и **CCircle**, които да наследяват **CFigure**. За всеки от класовете да се дефинират методи за лице и периметър, които във **CFigure** да са **virtual**. Да се направи клaс **CSemiCircle**, който наследява **CClrcle** и да се дефинират същите методи и за него.

**Задача 10**  Даден е клас:

class **CPerson**{

char \* name;

int age;

public:

virtual void print()=0; // извежда по подходящ начин всички данни за човека

};

Да се напишат:

* клас **Студент**, който да предефинира виртуалната функция и да има данни и за факултетен номер;
* клас **Деца**, който да предефинира виртуалната функция и да има данни и за имената и на неговите деца като масив от стрингове;
* клас **CDerived**, който наследява **Студент** и **Деца** и има всичките им данни и предефинира виртуалната функция;

**Задача 1**

#include <iostream>

#include <iomanip>

using namespace std;

class CPoint

{

public:

CPoint(){setPoint(0.0, 0.0);};

CPoint(float, float);

void setPoint(float, float);

float getX()const {return x;};

float getY()const {return y;};

void print()const;

private:

float x;

float y;

};

CPoint::CPoint(float a, float b)

{ setPoint(a, b);}

void CPoint::setPoint(float a, float b)

{

x=a;

y=b;

}

void CPoint::print()const

{

cout<<"Точката има координати (x,y): ("<<getX()

<<", "<<getY()<<")"<<endl;

}

void main()

{

cout<<setiosflags(ios::fixed | ios::showpoint)<<setprecision(2);

CPoint point1;

point1.print();

point1.setPoint(2.1, 3.2);

point1.print();

CPoint point2(12.0, 23.0);

point2.print();

}

Резултат от изпълнението на програмата **:**

Точката има координати (x,y): (0.00, 0.00)

Точката има координати (x,y): (2.10, 3.20)

Точката има координати (x,y): (12.00, 23.00)

**Задача 10**

#include<iostream>

#include<string.h>

using namespace std;

class Person

{

public:

virtual void print()=0;

};

class Student:public Person

{

protected:

int fn;

char \*name;

int age;

public:

Student(char \*s,int a,int f)

{

name=new char [strlen(s)+1];

strcpy(name,s);

age=a;

fn=f;

}

void print(){ cout<<name<<age<<fn;}

};

class Child:public Person

{

protected:

char \*cname;

public:

Child(char \*c)

{

cname=new char [strlen(c)+1];

strcpy(cname,c);

}

void print(){ cout<<cname;}

};

class CDerived:public Student,public Child

{

public:

CDerived(char \*s,int a,int fn,char \*c):Student(s,a,fn),Child(c){}

void print(){ cout<<name<<age<<fn<<cname;}

};

void main()

{

cout<<endl;

Student s("Ivan",13,23);

Person \*p=&s;

CDerived s1("Dani",23,45,"Ani");

Person \*ptr=(Student\*)&s1;

p->print();

cout<<endl;

s1.print();

cout<<endl;

ptr->print();

}

Header

Class B{

Public:

≡ / методи

Private: / използва се от само от методите

Dublereal; на класа и е скрит

} ---------- // --------- imag ; свойства

В () ;

В (double, double) ;

Където има В е за конструктора

Real imag

Void Add (B);

# include “име на файла”

име на (страница 2-15) не се разчита

void B:: Add( C Complexa)

операция за

разрешаване на област на действие

real = real + a.real

imag +=a. imag

}

В print ();

B ::B () { void

Real = imag = 0;

}

B:B (double r, double i)}

Real= r;

Imag = I;

}

(a+ib) (c+ id) + ac + iad +ibc – bd

I\*I = -1

B = b+c – d\*a

b.Add (с) ;

b. Snb (d);

b. Mp (a);

в Add / Sub (d). /Mp (a); /конкатенация слепване)

(Complex & (Complex: Sdd \_ (){

This → real= this →real +a.real

Return \*this;

}

Class SRat {

cin >> a >>b>>c;

Приятелски ф-ии на класа

Frend istram&operator << \_

(istream& in, SRat&a)}

Време in>>r.a>>r.b;

Оригинално копие от пеметта

Returniu

}

Предефиниране

Friend ostream & operator <<

(costerem& out, SRat&r) {

Псевдоним на cont

Ont <<r.a << “/” <<r.b;

Return out ;

} op. +

\_\_\_\_\_\_

Class Crat {

Public:

CRat (int aa, int bb=1) {

a= aa ;

b=bb;

do Norm ();

}

Унарни операции

{ CRat& operatpr + () { return \* this;}

{ CRat operatpr – () { return CRat (-a,b) ;

Връщаме копие

CRat& operatpr +( CRat& в) }

Return CRat (a\*rb + b\*ra, b \* rb))}

CRat operatpr \*( CRat& r ){

Return CRat (a\*ra , b\* rb);}

CRat operatpr / (CRat& r) {

Return CRat (a\*rb + b\*ra); }

Int operator == (CRat&r){

If (a\*r.b = = b\*r.a)

Return 1;

Else

Return 0 ;}

Int operator ! = (CRat&r){ return ? (\*this ==r);

If (a\*r.b ! = b\*r.a

Return 1;

Else

Return((CRat&r){

Int operator > (CRat&r){ return r>\*this ;

If (a\*r.b ! >b\*r.a

Else

Return 0;}

Работа с вектори

Class Cloint {

Protected :float x,y;

Public (Point Cfloat, float );

Void print () const {

Cont << x<< ‘\_’ << u<< endl; }

(CPoint:: CPoint Cfloat x 1, float y1 );

{x=x1; y = yi;)

Class CVector; public CPoin

{public;CVectorCfloat.float); / иниц. На кординатите на точка

CVect oroperator + (const CVector &) ;

CVect oroperator - (const CVector &) ;

Float oroperator \* (const CVector &) ;

Void point () cost {cont <<’(<<x<<;<<y<<’)’ \_ <<endl: }

(CVectot :: CVectot (float x 1, float y1 ) : CPoint *(базов клас)* (x 1, y 1)

{ } / няма собствени данни

CVectot CVectot : : operator ( const CVector & v)

{retur CVectot (x v.x, y v.y) ; }

Float CVectot :: operator \* (cont CVectot &v)

{ return ( x\*v.x + y\*v.y ) ; }

Void main ()

{ CPoint p1 (2,5);

CVectot p1 (2.5);

CVectot v1 (1.3), v2 (2.4) ;

CVectot \*pv;

CPoint \* pp = & p1;

CVectot v3;

V 3 = v1+v2 ;

V3.printl ) ;

Pp → print; }

Class cPerson {

Friend ostream & operator << ( ostream &, CPerson &);

Protected : char \* name;

Unsigned age;

Public : Person (char \*, unsigned) ;

Void print () const;

~ person () {delete name ; }

};

Person :: Person (char\*s, unsigned a)

{ name = new char [stream (s) +1];

Strcpylname, s);

Age = a;

}

Void person :: print () const

{ cont << name;

Cont << age; }

Ostream & operatpr << ( ostream & ont, CPerson &p)

{ return ont << o.name << p. age; }

Class CCar : public CPerson

Friend ostream & operator << (ostream &, CCar &);

Protected : char \* marka;

Unsigned page;

Public : CCar (char\*, unsigned, char\*, unsigned) {

Void print () const;

~ Ccar () { delete marka; }

};

CCar :: C Car (vhar\*s, unsigned age, char\* m , unsigned a) : \_\_

Cperson (s,age) { marka = new char [ stream (m) + 1];

Strcpy (marka, m);

Page = a;

}

Void CCar :: print () cost

{ cont << marka;

Cont << page; }

Ostream & operator << (ostream & ont, CCar & p)

{return ont << p.marka << p. oage; }

Void main ()

{CPerson p ( “ Иван”, 18);

CCar c ( …………..);

Cont << p;

Cont << c;

}

\_\_\_\_\_\_\_\_\_\_

Клас дата – месец, година, ден – 3 пром. От тип int създаване на поток за въвеждане и извецдане

Статичен клас (дата)

CPrate {

Public : CPrate (int d, int m, int y) {

Day = d;

Mondth = m;

Year = y: }

CDate operatpor + (int d) {

For [i=0, i<s; I ++)

Add \_ day ();

Return \* this: }

Private ;

Int year; mont, day;

Static in A [13]

Day \_ per \_ Mounth [13] / дни на месеца

}

CDate : A [] = { 0.31,28,31,30 ………}

Add \_ day () {

If (day< A [ month] ) ако не е края на месеца и 28 февруари

Day ++;

Else

If (( month == 2) && ( leap\_year () )) / за месец февруари и пров. За висок. Година

if (day == 28 ) day ++

else day = 1;

month ++;

else if (month ==2)

{ day = 1;

Month ++ ;)

Else if (month = = 12) {

Day = 1;

Month = 1;

Year ++;

}

Else (day =1; month ++ ; }

Оператор за присвояване

A=s

Cpate a,b

Продължителност на прогрмата

Operator + (CData &right)

Year = right. Year; при статична памет

Month = right. Month;

Day = right,day:

}

Operatpr = (CDate & right ) { За динамична памет

If (this !=& right)

Year = …………………………………различно място в паметта

} else cont<

CDate operatpr ++ () {

Add \_ day ();

Return \*this ; }

CDate operator ++

Return \* thi; {

CDate operator ++ ( int ) {

CDate temp;

Temp = \* this;

Add \_ Day()

Return \* temp:

}

Множествено наследяване

* Наследява се повече от 1 базов клас

Clas име :public CBase1, pribate CBase2

{ тяло }

Конструктора не се наследява

Зад. 8

Class CPerson {

Frien ostream & operator << (ostream &, CPerson &)

Protected: