ПЕ - конспект

1. Обекти и класове. Клас определение. Основни понятия.

С програмни обекти се моделират реални обекти от действителността, от които се извличат най-съществените признаци и дейности. Признаците на реалните обекти се моделират с променливи, а дейностите - с функции. Без да претендираме за изчерпателност, нека дадем няколко групи от обекти, които са подходящи за моделиране с програмни обекти. Състояние и фукционалност на обекта - Всеки един обект се описва с някакви характеристики, които формират състоянието на обекта, и извършва определен вид дейности, които определят неговата функционалност. Например едно комплексно число представлява набор от две реални числа - неговата реална и имагинерна стойности, z = a + bi. Същото число може да се описва още с модула си (абсолютната стойност) и неговата фаза - z = |z|exp(iq). Т.е. променливи в програмния обект "комплексно число" ще бъдат реалната и имагинерната част, но програмистът може да добави още две променливи - модула и фазата. Комплексните числа се ссъбират и изваждат и се умножават и делят едно на друго, т.е. в обекта е необходимо да има функции, които извършват тези дейности. Всъщност, това е възможно само с обектно-ориентираното програмиране и се състои в дефинирането на нови оператори +, -, * и /, които извършват действията с обектите.

Класове - В ООП обектите са конкретна реализация (an instance or an instantiation) на класовете. Класът е вид (тип) данни, а обектите са негова конкретна реализация, подобно на променливата a от декларацията int a , която променлива е реализация (т.е. конкретен представител, за който се отделя памет в програмата) на типа int; за разлика от класовете в C++ типът int е предефиниран тип, т.е. не е необходимо да се дефинира от програмиста. Един клас се специфицира с ключовата дума class и тялото на спецификацията, заградено с големи скоби и завършващо с точка и запетая. С помощта на обектите могат да се дефинират нови типове данни. Също така могат да се определят и нови операции с тези нови типове данни, а също и ново значение на операторите +, -, ++, --, * и /. Веднъж специфицирани класовете могат да се наследяват и променят като се добавят нови функционалности към тях, което изключително много улеснява програмистите и спомага за повторното използване на вече изготвените и тествани класове.

В дефиницията на класа има три ключови думи - public, protected и private. Последната ключова дума, означава "частни" и ограничава достъпа чрез обектите до данните (или функциите), които са дефинирани след нея. Тези полета (данни и/или функции) са достъпни само в класа. Първата ключова дума ("публични"), обратно, позволява тези функции и/или данни да са достъпни чрез обектите. Именно функциите в тази част позволяват да се осъществи достъп до данните в защитената част. Втората ключова дума означава "защитени" и нейната функция е нещо средно между другите две.

Конструторът има същото име като класа и след него има списък на параметри, затворени в кръгли скоби. Конструктурът се използва за първоначална инициализация на обектите от този клас - т.е. даване на стойност на променливите на обектите, както и някои други операции, като създавене на динамични променливи в хийпа (the heap). Деструкторът започва с тилда, ~, и има същото име като класа; между тилдата и името няма разстояние. Деструкторът се грижи за освобождаване на заетата динамична памет от съответния обект.

2. Методи и параметри. Член данни. Обекти.

Обекти са операции, които могат да бъдат приложени. Методите могат да имат параметри за подаване на допълнителна информация, необходима за изпълнението. При повече наблюдение в повечето случаи обекта може да бъде създаден от един клас, обекта има атрибути – стойности, съхранявани в полета. Класът определя кои области на обекта да има, но всеки обект има свои собствени стойности за състоянието на обекта. Всеки клас има свързан с него изходен код,който определя неговите данни, полета и методи.

Пример за декларация на клас:

```
class Point {
        public:
                double x;
                double y;
        };
```

Потребителски дефиниран тип е декларация на данни, които се използват, когато типът е инстанция на даден обект, както и набор от операции, необходими с цел манипулация.

Декларация на обект:

```
Point ob1;
ob1.x=3.0;
ob1.y=4.0;
```

Членовете на структура са по подразбиране публично достъпни.

Ако по подразбиране има конструктор, обект от тип клас трябва да се инициализира. Ако не, то има подходяш изпълнител на обекта. Ако не е по подразбиране, конструктора се произвежда от съставителя. Обектът може да бъде член на един сложен обект, ако: клас обекта притежава конструктор без параметри; обект на клас не притежава конструктор; Ако сложен обект има конструктор, включително стойности за инициализиране на неговите член променливи.

3. Програмиране точка: класове и обекти-ВЪВЕДЕНИЕ. От структурата С до дефиницията на клас.

С програмни обекти се моделират реални обекти от действителността, от които се извличат най-съществените признаци и дейности. Признаците на реалните обекти се моделират с променливи, а дейностите - с функции. Без да претендираме за изчерпателност, нека дадем няколко групи от обекти, които са подходящи за моделиране с програмни обекти. Състояние и фукционалност на обекта - Всеки един обект се описва с някакви характеристики, които формират състоянието на обекта, и извършва определен вид дейности, които определят неговата функционалност. Например едно комплексно число представлява набор от две реални числа - неговата реална и имагинерна стойности, z = a + bi. Същото число може да се описва още с модула си (абсолютната стойност) и неговата фаза - z = |z|exp(iq). Т.е. променливи в програмния обект "комплексно число" ще бъдат реалната и имагинерната част, но програмистът може да добави още две променливи - модула и фазата. Комплексните числа се ссъбират и изваждат и се умножават и делят едно на друго, т.е. в обекта е необходимо да има функции, които извършват тези дейности. Всъщност, това е възможно само с обектно-

ориентираното програмиране и се състои в дефинирането на нови оператори +, -, * и /, които извършват действията с обектите.

Класовете позволяват да дефинираме нови типове данни. Един клас представлява описание на тип, включващ едновременно данни и функции, които ги обработват. Данните се наричат член-променливи, а функциите – член-функции. Дефиницията на един клас включва декларация на класа и дефиниции на член-функциите.

Декларацията на класа има следния синтаксис:

```
class  <име-на-клас>{
        // Декларации на член-променливи
        // Декларации на член-функции
};
```

Дефиницията на член-функция има следния синтаксис:

```
<връщан-резултат> <име-на-клас>::<функция>(вх. парам.)
{
        // Тяло на функцията
}
```

Класове - В ООП обектите са конкретна реализация (an instance or an instantiation) на класовете. Класът е вид (тип) данни, а обектите са негова конкретна реализация, подобно на променливата a от декларацията int a , която променлива е реализация (т.е. конкретен представител, за който се отделя памет в програмата) на типа int; за разлика от класовете в C++ типът int е предефиниран тип, т.е. не е необходимо да се дефинира от програмиста. Един клас се специфицира с ключовата дума class и тялото на спецификацията, заградено с големи скоби и завършващо с точка и запетая. С помощта на обектите могат да се дефинират нови типове данни. Също така могат да се определят и нови операции с тези нови типове данни, а също и ново значение на операторите +, -, ++, --, * и /. Веднъж специфицирани класовете могат да се наследяват и променят като се добавят нови функционалности към тях, което изключително много улеснява програмистите и спомага за повторното използване на вече изготвените и тествани класове.

В дефиницията на класа има три ключови думи - public, protected и private. Последната ключова дума, означава "частни" и ограничава достъпа чрез обектите до данните (или функциите), които са дефинирани след нея. Тези полета (данни и/или функции) са достъпни само в класа. Първата ключова дума ("публични"), обратно, позволява тези функции и/или данни да са достъпни чрез обектите. Именно функциите в тази част позволяват да се осъществи достъп до данните в защитената част. Втората ключова дума означава "защитени" и нейната функция е нещо средно между другите две.

4. Constructors

Конструторът има същото име като класа и след него има списък на параметри, затворени в кръгли скоби. Конструктурът се използва за първоначална инициализация на обектите от този клас - т.е. даване на стойност на променливите на обектите, както и някои други операции, като създавене на динамични променливи в хийпа (the heap).

Конструктурът на един клас се извиква всеки път, когато се създава обект от този клас. Всякаква инициализация на данни, която е нужно да се извършим за даден обект, може да се изпълни автоматично от функцията конструктор. Той има същото име като името на класа, към който принадлежи, и не притежава тип на връщан резултат. Общият вид на конструктора е:

```
<име-на-клас>::<име-на-клас>(<аргументи>){
        //Тяло на конструктор
}
```

Може да създаваме повече от един конструктори, но те трябва да се различават по техните аргументи.

5. Destructors

Деструкторът започва с тилда, ~, и има същото име като класа; между тилдата и името няма разстояние. Деструкторът се грижи за освобождаване на заетата динамична памет от съответния обект.

Деструкторът се извиква автоматично при разрушаването на обект от класа. Деструктора може да извежда съобщение за разрушаване на обекта или да освобождава заделената за обекта динамична памет. Деструкторът има същото име като класа, но предшествано от символа ~. Деструкторът не може да връща резултат и не може да има аргументи:

```
~<име-на-клас>::<име-на-клас>(){
        //Тяло на деструктор
}
```

6. Променливи и оператори в класове. Основните типове данни. Деклариране на променливи.

В C++ има седем предефинирани основни типове данни.

char	символ (character)	'a', 'B', '$', '9', '\n'
short	малки цели числа	7, 30,000, -222
int	средни цели числа	също като short или като long
long	големи цели числа	1,234,567,890, -987,654,111
float	малки реални числа	13.7, 199.99, -6.2, 0.000231
double	големи реални числа	7,553.393.95,47, -0.046754364
long double	извънаредно големи реални числа	9,312,455,679,012,354.436

Тип символ (char). Променливите или константите от този тип съхраняват точно един символ. Променливите от тип символ се използват за съхранение на изключително малки числа, по-точно от -128 до 127, и заемат 8 бита (1 байт) в паметта на компютъра.

Целочислени типове (integers). Тези променливи позволяват съхранението на по-големи целочислени числа. Числовите константи, които стоят отдясно на изразите за присвояване, се пишат без разделителните запетаи (както е прието в някои страни - 243,345,566) или без интервалите между всеки три цифри (както понякога у нас се означават големите числа -- 243 345 566.

Неотрицателни целочислени типове (unsigned integers). При добавяне на unsigned пред числата се получават целочислен тип, който може да има само неотрицателни стойности (т.е. вкючително нула). Само с unsigned се означа типа unsigned int. По подразбиране, без думата unsigned целочислените типове са със знак, но те могат да се означат така и с използване на думата signed, но това на практика никога не се прави от програмистите.

Типове с плаваща запетая (floating point). Те се използват за означаване на реални числа, т.е. числа които имат целочислена част отляво на десетичния знак и дробна част, отдясно на десетичния знак, например 5.237, -123.98709, -.000234, 9876543. и т.н. Тези числа освен по този начин могат да се запишат и в експоненциална форма като 5.237E0, -1.2398709e2, -2.34E-4 и 9.876543e6>. Е идва от exponent и няма значение дали е с малка или голяма буква, то просто означава 10 на едикоя си степен.

7. Указатели и връзки. Константи. Изброен тип.

Всеки обект в езика (променлива, константа, масив, елемент на масив, структура, …) се съхранява в определени клетки на паметта. Прието е да се казва, че тези клетки се намират на определен „адрес“. Променлива, чиято стойност е адрес в паметта, се нарича указател. Стойността на указателя посочва местоположението на дадена променлива и позволява косвен достъп до стойността й. Указателите се дефинират също както обикновените променливи, но в началото на името се добавя знака „*“. Например:

```
// Дефинира указател към променлива от тип int
int *p;
```

Важно условие, е че типа данни в дефиницията на указателя трябва да съответства на типа данни на променливата, към която ще сочи. Адреса на дадена променлива може да се изведе чрез знака „&“.При добавянето или изваждането на цяло число към/от указател всяка единица всъщност е равна на броя байтове, определени в паметта за зададения тип на елемента от данните. дресната аритметика води до смислени резултати само когато данните имат взаимна връзка помежду си. Важно е данните да са подредени последователно в паметта и да са с еднаква мащабируемост. Това обяснява защо указателят трябва да се дефинира строго като указател от определен тип данни. Използването на масиви от указатели и символни низове води до съществено спестяване на памет.

Константи:

const <тип> <име на променлива> = <стойност>;

Удобни са при дефиниране на размерност на масиви, заделяне на памет и в всякакви подобни операции, където използваме една и съща постоянна величина на множество места в програмата. Важно е да се каже, че е задължително константите да се инициализират в

момента на деклариране си. За дефиниране на указатели към константи има една особеност – трябва указателят да се дефинира като указател към константен тип.

Изброен тип:

Изброен тип се използва за променливи, които могат да вземат само определен набор от стойности.

Спецификацията започва с ключовата дума enum, последвана от името на типа и заградени в големи скоби имената на стойностите на типа, които са разделени със запетайки. Накрая спецификацията завършва с точка и запетая. Типът се нарича изброен, защото всички стойности, които може да приема променлива от този тип, се изброяват в големите скоби. Пр:

enum days_of_week {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

Дефинирането на променливи от тип enum е като с всеки един тип в C++ - първо се изписва името на типа и след него се изписват имената на променливите, разделени със запетайка.

8. C + + операторите и на изразяване.

Всички изрази дават резултат
x= a+b;
Оператори за присвояване
a=0;
аритметични оператори
+, -, *, /,%, + =, -=, *=, /=, % =
a = a + 5; е равна на + = 5;
+ + или --
- Postfix или prefix
релационни и логически оператори
<> = <= ==! =
връщащи истина или лъжа
& &
||
!
побитови оператори
&, | , ^ (изключителни или) , ~ , >>, <<
за char, short ,int ,long стойности
INT a = 5; / / 101
A = A <<2 / / 10 100, които са 20

Логически оператори (Logical Operators). Т.н. бинарни логически оператори съставляват основата на формалната логика. Общо могат да се съставят 8 (8 = 23) такива оператора, но най-важните са логическото и (and), логическото или (or), изключващото или (xor), и логическото следствие (implication, импликация). Първите два бинарни логически оператора се използват директно в повечето програмни езици, включително в C++, и са разгледани по-долу. Изключващото или (xor) има стойност истина (True) само когато първият операнд има стойност истина, а вторият - лъжа (False), или обратно. Ако двата операнда имат еднакви стойности, то изключващото или дава лъжа. Логическото следствие (implication) има стойност лъжа само ако първият операнд (който играе ролята на причина) е истина, а вторият (който играе ролята на следствие) е лъжа. Това е така, защото очакваме, че ако се изпълни причината, то да настъпи следствието. В останалите случаи импликацията има стойност истина. По-принцип всички

възможни осем логически операции могат да се представят с логическото и (and), логическото или (or) и логическото не (not). Последният логически оператор е унарен (действа на един операнд) и превръща стойност истина в лъжа, както и лъжа в истина. Логически оператори са :&& (И), ||(ИЛИ), ! (НЕ).

По страшинство операторите се нареждат : първо * , /, %, после - + и -, след това <, >, <=, >=, ==, !=, след това && || и накрая = .

9. Подаване на данни чрез параметри. Accessor методи. Mutator методи.

Стойностите се съхраняват в полета (и други променливи) чрез прехвърляне на твърдения: променлива - израз;

Една променлива има единична стойност, така всички предишни стойности се губят.

Accessor методите са методи за изпълнение на поведението на обектите. Accessors-ите предоставят на информация за даден обект. Имат структура, състояща се от заглавието и тялото. Заглавния ред определя името на метода, а тялото отчета на метода. Пример:

```
public int getPrice() {
return price;
}
```

Mutator методи: имат сходна структура с Accessor методите: заглавие и тяло. Използват се за промяна състоянието на даден обект. Постигат го чрез промяна на стойността на една или повече области. Обикновено съдържат задачи отчети и получават параметри. Пример:

```
public void insertMoney{int amount)
{
balance = balance + amount;
}
```

10. Функции. Деклариране прототипа на функция. Дефиниране на тялото функция. Обръщане към функции. Справяне с локални и глобални променливи. Определяне и използване на претоварени функции

Функцията е основната програмна единица. Можем да разглеждаме функциите като логическа част от по-голяма програма. Този фрагмент може да се изпълнява многократно в различни части на програмата. Функциите се дефинират по следият начин:

```
тип_на_функцията име(входни_параметри)
{
        оператори;
        return резултат;
}
```

Типа на връщания резултат трябва да съвпада с типа на самата функция. Когато няма нужда от връщане на резултат се използва тип на функцията void.

Конструктор и деструктор са член функции, които винаги присъстват. Можете да добавите специално предназначение като ги декларирате като член-функции.

Претоварване на функции имаме, когато повече от 1 функция със същото име и различни например параметър списък имат двойна използваемост или ако имаме разлика между връщания вид на резултата и получавате грешка на компилатора.

Променливите, дефинирани в тялото на функция се наричат локални променливи. Промените по тях важат само за тялото на самата функция. Глобални са всички променливи, които са дефинирани извън тялото на функциите. Когато дадена функция промени такава променлива, то промяната се отразява навсякъде.

Локални променливи

• Полетата са един вид променлива.

- Те съхраняват ценности чрез живота на даден обект.
- Те са достъпни в целия клас.

• Методите могат да включват променливи с по-кратък живот.

- Те съществуват само докато методът се изпълнява.
- Те са достъпни само в рамките на метода.

Глобалните променливи се декларират в началотона програмата и се ползват в цялата програма.

11. Член- функции. Глобални функции

Класовете позволяват да дефинираме нови типове данни. Един клас представлява описание на тип, включващ едновременно данни и функции, които ги обработват. Данните се наричат член-променливи, а функциите – член-функции. Дефиницията на един клас включва декларация на класа и дефиниции на член-функциите. Дефиницията на член-функция има следния синтаксис:

```
<връщан-резултат> <име-на-клас>::<функция>(вх. парам.)
{
        // Тяло на функцията
}
```

Конструктор и деструктор са член функции, които винаги присъстват. Можете да добавите специално предназначение като ги декларирате като член-функции.

Глобалната функция е обявена извън декларацията на класа. Глобалната функция може да има име, отговарящо на името на класа на член функцията, ако се различава по параметър от списъка.

12. Презаредени функции

Замяната на функции или т.н. презаредени функции е едно превъзходно удобство в C++. Тази концепция позволява да се използва едно и също име за различни функции, чрез различаването им чрез набора от параметри, с които функциите се извикват.

Претоварване на функции имаме, когато повече от 1 функция със същото име и различни например параметър списък имат двойна използваемост или ако имаме разлика между връщания вид на резултата и получавате грешка на компилатора. Също така, използването

на няколко функции (това се нарича name proliferation - разпространение на имената) не само способства за повече програмистки грешки, но затруднява програмиста, който трябва да помни и да се съобразява с различните имена. Допълнително, описването на тези няколко функции в помощния файл или в ръководството ще заеме повече място и труд.

13. Локални променливи. public срещу private.

♦ public атрибутите (полета, конструктори, методи) са достъпни за други категории.
♦ полета не трябва да са public.
♦ private атрибутите са достъпни само в рамките на един и същи клас.
♦ само методи, които са предназначени за други класове трябва да са public.
Информация крие
• Данните, принадлежащи към един обект са скрити от други обекти.
• Бъдете наясно, че обект може да направи нещо, но не как да го направи.
• Информацията крие повишено ниво на независимост.

14. взаимодействие между обекти. Абстракция и Модулация

• Области, параметри и локални променливи всички са променливи.
• полета съществуват за да определят живота на даден обект.
• Параметрите се използват за получаване на стойности в конструктор или метод.
• локалните променливи се използват за краткотрайно временно складиране.
Абстракция, е способността да се игнорира информация на части, които обръщат повече внимание на по-високо ниво на проблема.
Модулация е процес разделящ цялото в точно определени части, които могат да бъдат построени и разгледани поотделно, и които си взаимодействат в строго определени начини.

15. Дисплей-часовник пример: обект и клас схема; проект и кода на елементите.

he clock display
11:03
One four-digit display?
Or two two-digit displays?
03
Implementation - NumberDisplay

```
public class NumberDisplay {
private int limit;
private int value;
Constructor and methods omitted.
}
```

Implementation -ClockDisplay

```
public class ClockDisplay {
```

```
private NumberDisplay hours;
private NumberDisplay minutes;
Constructor and methods omitted.
}
```
Пример:
```
SomeObject obj ;
int i;
Fragments of Source code: NumberDisplay
public NumberDisplay(int rollOverLimit) {
limit = rollOverLimit; value = 0;}
public void increment(){
value =   (value +1)   % limit;}
public String getDisplayValue() {
if(value <  10)
return "0" + value;
else
return “ ” + value;
}
public class ClockDisplay{
private NumberDisplay hours;
private NumberDisplay minutes;
private String displayString;
public ClockDisplay ()
{
hours = new NumberDisplay(24);
minutes = new NumberDisplay(60);
updateDisplay() ; }}
public void timeTick() {
minutes.increment();
if(minutes.getValue()   == 0) {
hours.increment();
}
updateDisplay() ;
}
private void updateDisplay() {
displayString =hours.getDisplayValue() + ":" + minutes.getDisplayValue();
}
```

16. **Група обекти. Колекции и итератори. Пример: проект "бележник", обект структура; използва за събиране в кода**

```
public class Notebook
{
public NotebookO {
```

```
notes = new Arrayl_ist<String>();
}

public class Notebook i
private ArrayList<String> notes;



public void storeNote(String note) i
notes.addCnote);
}
public int numberOfNotesO {
return notes.size();
}
}
```

Изтегляне на обект

```
public void showNote(int noteNumber) i
if(noteNumber < 0) {
}
else if(noteNumber < numberOfNotesO) {
System.out.println(notes.get(noteNumber));
}
else {
}
}
```

17. Проектиране на класове в обектно-ориентирана програма.

Софтуер промени

- Софтуерът не е като роман, който е написан веднъж и след това остава непроменен.
- Софтуер може да бъде удължен, коригирани, поддържане, пренесли, адаптиран ...
- Работата се извършва от различни хора, с течение на времето (често десетилетия).

18. Код - качествена оценка. Съединител. Сближаване.

Има две важни концепции за качеството на кода:

- Свързване
- Кохезионния

Свързването се отнася до връзките между отделните звена на програмата. Ако два класа зависят тясно по много подробности за всяка друга страна, ние казваме, че са тясно свързани. Ние се стремим към разхлабването на връзките.

Загубата на свързване позволява да се разбере един клас без четенето на други, промяна на един клас без да засяга други и по този начин се подобрява поддръжката.

Сближаване се отнася до броя и разнообразието на задачите, за които един-единствен елемент е отговорен. Ако всеки блок отговаря за една логическа задача, ние казваме, че е с високо сближаване. Сближаване се отнася за класове и методи. Стремим се към високо сближаване. Високото сближаване прави по-лесно различаването какво е клас или метод, използва описателни имена и повтаря класове и методи. Сближаване от методи –методът, следва да отговарят за една и само една ясно определена задача. Сближаване от класове – класът би трябвало да представлява по-добре дефиниран обект.

19. **Код за качество: код на дублирането. Отговорност - задвижвано от дизайна. Пример за дизайн RDD.**

Кодът на дублиране

• е показател за лошо проектиране,

• прави поддръжката по-трудна,

• може да доведе до въвеждането на грешки по време на поддръжката.

Пример за свързване :

• Промяна на програма за добавяне на нови посоки

• Никога не се правят публични полета!

Responsibility-driven design (RDD)

• Въпрос: къде трябва да се добави нов метод (кой клас)?

• Всеки клас трябва да отговаря за манипулиране на данните си.

• В клас, който е собственик на данни следва да отговаря за преработката му.

• RDD води до ниски свързвания.

Локализиране на промяната

• Една от целите за намаляване на свързването и отговорността е задвижвана от дизайна и е да се локализира промяната.

• Когато е необходима промяна, тъй като някои класове могат да бъдат засегнати.

Сближаване от методи- Всеки един метод, следва да отговарят за една и само една точно определена задача. Сближаване от класове- Всеки клас трябва да представлява един-единствен, точно определена структура(domain).

20. **Дизайн въпроси в обектно-ориентирания подход. Дизайн насоки.**

Дизайн насоки

• Методът е твърде дълъг, ако го прави повече от една логическа задача.

• Класът е прекалено сложен, ако тя представлява повече от един логически субект.

Опитайте се да:

• Тествате с най-различни входове

• Уверете, че всичко е в съответствие със стандартите

• Търсете дизайнерски проблеми: кодекс дублиране, дълги методи, Magic номера, Метод в изгнание, Интерфейс грижа, липса на общ харектер.

Преглед

• Програмите са непрекъснато променят.
• Важно е да се направи тази промяна възможна.
• Качеството на кода изисква много повече от просто изпълнение правилно наведнъж.
• кодекс трябва да бъде разбираема и Maintainable.
• Добрият код избягва дублирането, показва високо сближаване, ниски съединения.
• кодиран стил (коментира, именуване, оформление и т.н.)
• Има голяма разлика в размера на работата, необходими за промяна лошото структуриране и добре структуриран код.

21. **Класове и обекти. Класове в заглавието и кода. Създаване и унищожаване на обекти.**
    Организиране на класове в заглавието (декларация) и изходни (дефиниция) файлове (части):
    •class name { body }
    • секции в тялото: public:
    private:
    protected:
    • данни за членовете и функция членове
    • правилата за достъп
    • # include директивата и хедър файлове
    • double colon syntax ::
    • наследство :

    Дефинирането на клас е по-добре в заглавния файл. Изпълнение на клас е в. CPP файла с #include отчета. Класът трябва да има конструктор. Конструкторът има същото име като на класа и може да има или няма параметри. Конструктор, без параметър и недействителни връщане по подразбиране е конструктор. Ако няма конструктор по подразбиране се определя в компилатора. Класа трябва да има деструктор. На деструктор не може да връща стойност или да предприеме всякакви параметри изрично и мълчаливо. Обектът е създаден с помощта на New и унищожени от Delete. Създаването и разрушаването могат да се направят мълчаливо, когато е необходимо.
    Клас членове се определят със ключовата дума static. Те се наричат с предимство на клас (име на класа:: functionname ()). Обект членове са определени без static.

22. **Defining constructors and destructors. Defining class members**
    Конструктурът на един клас се извиква всеки път, когато се създава обект от този клас. Всякаква инициализация на данни, която е нужно да се извършим за даден обект, може да се изпълни автоматично от функцията конструктор. Той има същото име като името на класа, към който принадлежи, и не притежава тип на връщан резултат. Общият вид на конструктора е:

    <име-на-клас>::<име-на-клас>(<аргументи>){
    //Тяло на конструктор

```
}
```

Може да създаваме повече от един конструктори, но те трябва да се различават по техните аргументи.

Деструкторът започва с тилда, ~, и има същото име като класа; между тилдата и името няма разстояние. Деструкторът се грижи за освобождаване на заетата динамична памет от съответния обект.

Деструкторът се извиква автоматично при разрушаването на обект от класа. Деструктора може да извежда съобщение за разрушаване на обекта или да освобождава заделената за обекта динамична памет. Деструкторът има същото име като класа, но предшествано от символа ~. Деструкторът не може да връща резултат и не може да има аргументи:

```
~<име-на-клас>::<име-на-клас>(){
        //Тяло на деструктор
}
```

Всяка променлива в C се характеризира с три атрибута:

1. Тип на променливата – определя стойностите, които променливата може да приема и съответно необходимото количество памет, което трябва да се задели
2. Област на действие – мястото в програмата, където тя е „видима“, например разгледаните вече глобални променливи имат по-широка област на действие от локалните
3. Период на активност – част от времето за изпълнение на програмата, през което ще се съхранява последната текуща стойност. Не се посочва явно при описанието на променливите. Този клас памет е отделен за „външни“ или още „глобални“ променливи. Тези променливи се записват в областта за данни на програмата и стойностите им се пазят до самия и край.

23. Defining relationships between different objects in an application

Дефиниране на клас е по-добре в заглавния файл. Изпълнение на клас е в. CPP файла с #include отчета. Класът трябва да има конструктор. Конструкторът има същото име като на класа и може да има или няма параметри. Конструктор, без параметър и недействителни връщане по подразбиране е конструктор. Ако няма конструктор по подразбиране се определя в компилатора. Класа трябва да има деструктор. На деструктор не може да връща стойност или да предприеме всякакви параметри изрично и мълчаливо. Обектът е създаден с помощта на New и унищожени от Delete. Създаването и разрушаването могат да се направят мълчаливо, когато е необходимо.

Клас членове се определят със ключовата дума static. Те се наричат с предимство на клас (име на класа:: functionname ()). Обект членове са определени без static.

24. Наследяване. Подтипизиране. Замяна. Полиморфни променливи.

Когато един клас наследява друг се използва общата форма:

```
class <производен-клас>:<тип-достъп> <име-базов-клас> {
        //... };
```

Типът достъп е една от трите ключови думи: public, private или protected. Ако е необходимо да се предаде аргумент на конструктора на базовия клас, то всички необходими аргументи за базовия и за производния клас се предават на конструктора на производния клас. Като се използва разширената форма на декларация на конструктора на производния клас, могат да се предадат съответните аргументи и към базовия клас:

```
<производен конструктор>(списък аргументи):<базов-клас>(списък аргументи) {
        //тяло на конструктора на производния клас
}
```

Полиморфични променливи - Обект променливи са полиморфни (Те могат да притежават обекти с повече от един вид.). Те могат да притежават обекти на декларираните видове, или подтипове на декларираните типове.

Понятието „полиморфизъм“ идва от биологията. То описва „съществуване на морфологично различни индивиди в границите на един вид“. Това например са пчелите и търтеите в пчелния кошер – те са от един вид, но имат съвсем различни функции. В обектно-ориентираното програмиране под „полиморфизъм“ се разбира същото – различни обекти (индивиди) от един и същи вид (базов клас) извършват различни дейности. Тук се разбира, че индивидите наследяват характерните черти на вида, но променят някои от неговите функционалности.

25. **Пример за база данни с различни предмети в него. Проект, обектния модел, клас диаграма. Код реализация.**

"База данни за мултимедийни развлечения" Database of Multimedia Entertainment (DoME)

• магазини подробности за CD-та и видео

- CD: заглавие, изпълнител, "песни, игрово време, got it, коментар

-Video: заглавие, режисьор, игрово време, got it, коментар

• позволява (по-късно) да се търси информация или да отпечатват списъци.

В примера имаме 2 DoME обекта: CD и Video.

Код на CD:

```
public class CD {
private String title;
private String artist;
private String comment;

CD(String theTitle, String theArtist) {
title = theTitle;
artist = theArtist; comment = "  " ;
}

void setComment(String newComment) {  ...  }

S tr i ng getCommen t() {  …}
```

```
void print() {   …   }
…
}
```
Код на Видео:
```
public class Video {
private String title;
 private String director;
private String comment;

Video(String theTitle,  String theDirect)
{
title = theTitle; director = theDirect; comment = 11 " ;
}
 void setComment(String newComment) {  ...  }
S tr i ng ge tCommen t() {   ...   }
void print() {   ...   }
}
```
Код на базата данни:
```
class Datebase (
private ArrayList<CD> cds;
private ArrayList<Video> videos;
…
public void list()
{
for(CD cd: cds)
{ cd.print() ;
System.out.println();
}

for{Video video   :   videos  )   { videos.print();
System.out.println();
}
}
}
```
Основни проблеми:

• код на дублирането

- CD и видео класовете са много сходни {голяма част са еднакви)
- прави поддръжката трудно / повече работа
- въвежда опасност от грешки поради неправилна поддръжка

• код на дублирането и в клас на базата данни

26. Пример за база данни с различни предмети в него, подобрена версия с наследството. Суперкласа. Подкласа. Наследяване и конструктори.

При наследяването имаме

• Определяне на суперкласа: Позиция

• дефиниране на подкласовете за Video и CD

• суперкласата определя общи атрибути

• на подкласовете се наследяват атрибутите на суперкласовете

• на подкласовете добавяме собствени атрибути

```
public class Item
{
private String title;
private int playingTime;
private Boolean gotIt;
private String comment;
}
```

Superclass

```
public class Item {
private String title;
 private int playingTime;
private boolean gotlt;
private String comment;
}
```

Subclasses

```
public class CD extends Item {
private String artist;
 private int numberOfTracks;
}
public class Video extends Item {
private String director;
}
public class Item
{
private String title;
 private int playingTime;
private boolean gotlt;
 private String comment;
public I tern. (String theTitle,  int time)
{
title = theTitle;
playingTime = time;
gotlt = false;
```

```
 comment = "";
}
```

27. **Creating deeper hierarchies.**

```
public class Database
{
private ArrayList items;
public Database()
{
items = new ArrayList();
}
public void addItem(Item theItem)
{
items.add(theItem);
}
…
}

public void list() {
for (Item item :  items )   { item.print () ;
System, out .println () ; }}
```

28. **Подтипизиране. Подкласа и подтипизиране. Подтипизиране и оценка. Подтипизиране и параметър, минаваща**

Подтипизиране
Първо, имахме: public void addCD (CD theCD)
public void addVideo (видео theVideo)
Сега, ние имаме: public void addItem (Item theItem)
Викаме тези методи с:

```
Video MyVideo = new Video (...);
Database. addItem (MyVideo);
```

Класове определят вида.Подкласовете определят подтипа. Обекти на подкласа могат да се използват, когато предмет на supertypes са задължителни. (Това се нарича заместване.)

```
public class Database
{
public void addItern{Item theItem) {
…
}
}
Video video = new Video(...);
 CD cd = new CD (...);
```

```
database.addItern(video);
 database.addItem(cd);
```

29. Polymorphic variables.

• Обект променливи са полиморфни.(Те могат да притежават обекти с повече от един вид.)

• Те могат да притежават обекти от декларираните видове или подтипове на декларираните типове.

Понятието „полиморфизъм“ идва от биологията. То описва „съществуване на морфологично различни индивиди в границите на един вид“. Това например са пчелите и търтеите в пчелния кошер – те са от един вид, но имат съвсем различни функции. В обектно-ориентираното програмиране под „полиморфизъм“ се разбира същото – различни обекти (индивиди) от един и същи вид (базов клас) извършват различни дейности. Тук се разбира, че индивидите наследяват характерните черти на вида, но променят някои от неговите функционалности.

30. Наследяване в ООП. Значението на наследството в обектно-ориентираното програмиране. Определяне на базовия клас. Определянето на извлечения клас. Достъп до база-членове на клас от производния клас.

BankAccount базов клас действа като хранилище за общите член функции и членове на данни. Разрешение за генериране на "банкова сметка " не може да се създаде само с получени клас сметки. За да се постигне този клас BankAccount е "абстрактен" клас.

Опитът да се създадете BankAccount обект с "new" води до грешка!

```
#include "CurrentAccount.h"
#include "SavingsAccount.h"
int main(void)
{ Console::WriteLine(S"Testing the CurrentAccount");
CurrentAccount * current = new CurrentAccount("Emily", 100);
current->Credit(500);
current->Debit(600); // Should be accepted
current->Debit(1); // Should be declined
Console::WriteLine(S"\nTesting the SavingsAccount");
SavingsAccount * savings = new SavingsAccount("Thomas");
savings->Credit(500);
savings->Debit(50); // Should be accepted
savings->Debit(46); // Should be declined
return 0;
}

void BankAccount::Credit(double amount)
{ balance += amount;
Console::Write(S"After credit, new balance is: "); }
void BankAccount::Debit(double amount)
```

```
{ if (CanDebit(amount))
{ balance -= amount;
Console::Write(S"Debit succeeded, new balance is: ");}
else
{ Console::Write(S"Debit refused, balance is still: ");}
}
```

За извикване на функция или данни от супер класа (както BankAcount), ние можем да напишем:

```
__super:: function (..);
```

В управляван код можете да определите клас като "sealed". Тогавава е класът не може да да бъде наследен. Това е полезна мярка за сигурност. Наследството може да се направи чрез интерфейси, както и чрез главни класове. Интерфейсът е подобен на този клас, но всички главни-функции в него са чисто виртуални. Така че, една istantiation на интерфейс не може да се направи! Интерфейсите са полезни за определяне на общи възможности за различни класове с различни реализации.

```
__gc __interface IStorableAsXml
{
void ReadFromXmlFile(String *XmlFileName);
 void WriteToXmlFile(String *XmlFileName);
};
...
__gc __sealed class SavingAccount: public BankAccount, public IStorableAsXml
{
public:
...
}
```

31. С помощта на виртуални дума за постигане на полиморфизъм. Дефиниране на абстрактни класове и абстрактни методи

Трябва да има базов клас може и с някои (виртуални) функции, с различно значение в бъдещето.

```
class Orbiter
{
private:
double m_mass;
XY m_prior, m_current, m_thrust;
public:
 Orbiter( XY current, XY prior, double mass);
XY GetPosition() const;
void Fly();
};
```

Обогатен вариант с display () метод.

```
class Orbiter
{
protected:
double m_mass;
XY m_current, m_prior, m_thrust;
public:
Orbiter( XY current, XY prior, double mass)
{ m_current = current; m_prior = prior; m_mass = mass; }
XY GetPosition() const;
void Fly();
virtual void Display() const;
};
```

Следния пример се приема, че orbiterArray [] съдържа препратки към обекти на класовете, производни на Orbiter:

```
extern Orbiter* orbiterArray[];
for( int i = 0; i < MAX; i++)
{
orbiterArray[i] ->Fly();
orbiterArray[i] -> Display();
}
```

Идея за чисто виртуални функции (ако вече съществуват в един клас, това е забранено за изграждане на обект на този клас).

```
virtual void Display() const =0;
class Planet : public Orbiter
{ public:
Planet( XY current, XY prior, double mass) :Orbiter( current, prior, mass) {}
void Display() const;
};
class SpaceShip : public Orbiter
{ private: double m_fuel; XY m_orientation;
public:
SpaceShip( XY current, XY prior, XY thrust, double mass, double fuel, XY orientation)
: Orbiter(current, prior, mass)
{m_orientation = orientation; m_fuel = fuel;
m_thrust = thrust;
}
void Display() const;
...
}
```

32. Наследяване и полиморфизъм в една програма. Пример

Трябва да има базов клас може и с някои (виртуални) функции, с различно значение в бъдещето.

```
class Orbiter
{
private:
double m_mass;
XY m_prior, m_current, m_thrust;
public:
 Orbiter( XY current, XY prior, double mass);
XY GetPosition() const;
void Fly();
};
```

Обогатен вариант с display () метод.

```
class Orbiter
{
protected:
double m_mass;
XY m_current, m_prior, m_thrust;
public:
Orbiter( XY current, XY prior, double mass)
{ m_current = current; m_prior = prior; m_mass = mass; }
XY GetPosition() const;
void Fly();
virtual void Display() const;
};
```

Следния пример се приема, че orbiterArray [] съдържа препратки към обекти на класовете,производни на Orbiter:

```
extern Orbiter* orbiterArray[];
for( int i = 0; i < MAX; i++)
{
orbiterArray[i] ->Fly();
orbiterArray[i] -> Display();
}
```

Идея за чисто виртуални функции (ако вече съществуват в един клас, това е забранено за изграждане на обект на този клас).

```
virtual void Display() const =0;
class Planet : public Orbiter
{ public:
Planet( XY current, XY prior, double mass) :Orbiter( current, prior, mass) {}
void Display() const;
```

```
};
class SpaceShip : public Orbiter
{ private: double m_fuel; XY m_orientation;
public:
SpaceShip( XY current, XY prior, XY thrust, double mass, double fuel, XY orientation)
: Orbiter(current, prior, mass)
{m_orientation = orientation; m_fuel = fuel;
m_thrust = thrust;
}
void Display() const;
...
}
```

33. Virtual ключова дума. Виртуални функции и призова виртуални функции от базовите класове.

Виртуалните функции могат да се поставят в базовия клас, както и извън класа.
if Orbiter class contains Orbiter::Fly()
и тази функция използва ъглов момент за неговото изчисление, който е специфичен за всеки получен клас. Така че това е удобно да го обяви в Orbiter така:
protected:
virtual XY GetSpecificCalculation() const = 0;
и на всеки получен клас е длъжен да предостави на своята по-горна функция.

34. Вградени обекти . Copy constructors.

• както конструкторът по подразбиране, конструктурът Copy е член функция и компилаторът генерира
• целта на копиращия конструктор е да се направи нов обект от същия клас, от съществуващ обект, който се предава като аргумент.
• Копиращия вътрешноредов конструктор за XY клас изглежда така:

```
XY( const XY& xy)
{
x = xy.x; y = xy.y;
}
```

• автоматично генерирания копиращ конструктор просто е член копие на всички данни
• за комплексни класове (с памет разпределяне и т.н.) е добра практика да се пишат собствени копиращиконструктори
• нотация XY и казва - съставител минава на адреса на обекта XY като аргумент, а не копие от обект
• Изпозването на един копиращ конструктор:

```
XY alpha(1.0,2.0);
XY beta = alpha;
```

XYgamma(alpha);
Копиращия конструктор се използва в параметър, преминаване - когато официално има замяна на параметър се извършва инициализация
void f(XY xy);
XY alpha(2.0,' 3.0);
func(alpha); - копие до списъкът с аргументите

```
struct string{ char *p;
int size;
string(int sz) { p = new char[size = sz];}
~string() {delete p;}
};
void f() {
string s1(10);
string s2(20);
s1 = s2; }
struct string{ char *p;
int size;
string(int sz) { p = new char[size = sz];}
void operator = (string&)
~string() {delete p;}
};
void string::operator= (string& a) {
if(this == &a) return;
delete p;
p = new char[size = a.size]; // new ‘p’
strcpy(p, a.p); }
```

35. Оператори за присвояване

Това е като копие на конструктора, освен че работи на съществуващ обект и не създава нов. Компилаторът генерира оператори подразбиране и генерира повикване към тях. Ако ви се налага да пишете собствен оператор за присвояване, той ще изглежда така:

```
const XY& operator=(const XY& xy)
{
x=xy.x;
y=xy.y;
return *this;
}
```

Използването на оператора е възможно по този начин (поради връщането XY):

```
xy1 = xy2 = XY(4.5, 5.0);
```

Референтните данни са скрити параметри. Полезно е, ако функцията ще използва параметър за промяна на променливата във викащата програма. Така че референцията няма да бъде

константа. Искаме да избегнем копиране на голям обект в извикването на функцията stack. Така че, референтната ще бъде константа.

```
void Show(const XY& xy) {
printf("x=%f, y= %f\n", xy.GeX(), xy.GetY());}
```

36. Референтен параметри (const срещу non-const). Как препратките работят в C + + .

В момента има следния код заявление за изграждане на обект от тип planet:

```
XY current(1000.0, 2000.0);
XY prior(900.1, 1000.2);
Planet Earth(current, prior, 2.7E+8);
```

Да не забравяме, че имаме следните декларации клас за използваните обекти:

```
class XY{
public:
double x,y;
XY() {x =0.0; y = 0.0;}
XY(double a, double b;) {x = a; y = b;}
XY(const XY& xy)
{ x = xy.x;
y = xy.y;
}
const XY& operator=(const XY& xy)
{ x = xy.x;
y = xy.y;
return *this;
}};
class Planet : public Orbiter
{
public:
Planet (XY current, XY prior, double mass)
:Orbiter(current,prior,mass){}
void Display();
}
class Planet : public Orbiter
{ public:
Planet(XY& current, XY& prior, double mass)
: Orbiter(current, prior, mass) {}
void Display();
};
```

Забележки и подобрения:

- сега Orbiter и Planet конструктори използват XY препратки.
- Orbiter конструктор е различна: инициализация на данни членовете се различава.
- C + + позволява синтаксис като m_mass (маса), дори и за вграждане на видове
- сега, вместо две повиквания по подразбиране XY конструктор и две повиквания към

Оператори за присвояване (както в предишния слайд) компилаторът генерира две повиквания към XY копиращ конструктор само (преди m_mass (маса))
• всички отчети след ":" включително повиквания към базовия клас и строители се изпълняват преди тялото на конструктора

37. Returning references. returning reference from a function

• Функцията може да върне препратка (еквивалент на връщане на указателя)

```
const double& XY::GetConstX() const {return x};
```

• Така обявени, функцията връща const позоваване на XY обект и може да бъде използва само от дясната страна на прехвърлянето.

```
my.GetConstX () = 1.0;          / / грешно!
```

Грешно е следното:

```
int *GetInt()
{
int result = (int) (rand() / 1000);
return &result;                 // don't do this!!
}
```

функцията връща указател към стека, който ще бъде използван другаде, след като функцията върне резултат. Равносилна грешка е:

```
int& GetInt()
{
int result = (int)(rand() / 1000);
return result;
}
```

38. Constructing embedded objects. Destructing embedded objects

Constructing embedded objects.

1. Компилатора на обекта е декларация. И той знае общата памет необходима за обект Spaceship и q разпределя.
2. Всички вградени обекти (m_current, m_prior, m_thrust) са изградени
3. строителя Orbiter е наименуван
4. на m_orrientation вградени обект е constucted
5. конструктора на Spaceship функцията е именуван

• Това е най-правилния списък за строителство

• Дизайнът на класа и синтаксиса на конструктора Spaceship се определя точно кои конструктори (по подразбиране, явни или копиращи) да именува

Destructing embedded objects.

Нека Spaceship да трябва да бъде унищожен:
Той е получен от клас Orbiter и вградени обекти (като XY), определени както в базовия клас и с произведени от него класове. И така:
1. Spaceship се нарича деструктора
2. m_orientation вграден обект е унищожен
3. Orbiter се нарича деструктора

4. m_current, m_prior и m_thrust вградени обекти са унищожени
5. паметта за Spaceship е освободена

class SpaceShip : public Orbiter
{private: double m_fuel; XY m_orientation;
public:
SpaceShip(XY current, XY prior, XY thrust, double mass, double fuel, XY orientation)
: Orbiter(current, prior, mass)

• деструктори не се наследяват. Компилаторът генерира по подразбиране, деструктор за всеки клас, ако не изрично пише един. Тогава, получен клас деструктор винаги призовава своя деструктор от базовия клас.
Унищожаването на производни класове ще бъдат непълни по този начин.
• Ако в производния клас на деструктора е обявен за виртуална:
virtual ~ Orbiter () {}
компилаторът генерира по подразбиране за деструктор на клас Spaceship, например, първо ще унищожи всички елементи, собственост на Spaceship и след това призовава Orbiter деструктора

39. Allocating objects on the heap & Object interrelationships

• някои обекти са отпуснати от стека и унищожени, когато излязат от обхват
• разпределяне от heap означава, че обектът е достижим.
• операторите New I Delete

```
double* mybuff = new double[1024];          // for storage
Planet* pFarth = new Planet(...);           // for object
…. delete pFarth;
```

• отнасящи се до обекти чрез указатели

```
pEarthFly();
Orbiter* pMy = new Planet(……);
and then: pMyDisplay()  // Orbiter virtual function.
                                    //Display() from Planet class is called
```

Указателят може да бъде извън обхват или унищожен.

40. Virtual destructors

41. Object interrelationships – Friend classes and friend functions

Ще дойде време, когато ще искате една функции да има достъп до private членовете на един клас, без всъщност тя да е член на този клас. С тази цел C++ поддържа приятелските (friend) функ¬ции. Една приятелска функция не е член на даден клас, но тя има достъп до неговите private елементи. Две са причините, в които приятелските функции са полезни - при предефинирането на оператори и при създаването на опре¬делени видове входно-изходни функции. Ще трябва да изчакате, за да видите употребата на приятелските функции в такива ситу¬ации. Все пак съществува и трета

причина за употреба на прия¬телските функции - когато искате една функция да има достъп до private членовете на два или повече различни класа. Точно та¬зи употреба е разгледана тук.
Една приятелска функция се дефинира като обикновена фун¬кция, която не е член на клас. Но в декларацията на класа, за който тя ще бъде приятелска, е включен прототипът на функция¬та, предшестван от ключовата дума friend.
Важно е да се разбере, че една приятелска функция не е член на класа, за който тя е приятелска функция. Т.е. не е възможно да се извика приятелска функция посредством името на обекта и оператор за достъп до член на клас (точка или стрелка).

42. Static class members

STATIC ЧЛЕНОВЕ НА КЛАС

Член-променливите на един клас може да бъдат декларирани ка¬то static (статични). С тяхна помощ можете да се справите с дос¬та заплетени ситуации. Декларирайки една член-променлива ка¬то static, разрешавате да съществува само едно единствено копие на тази променлива- без значение колко обекта от този клас са създадени. Всички обекти използват тази променлива. Запомне¬те, че за обикновената член-променлива се създава копие за все¬ки нов обект. (Тоест, всеки обект притежава собствено копие на обикновените член-променливи от класа.), static член-променли¬ва обаче има само едно копие, което се използва от всички обек¬ти на класа. Същата static променлива се използва и от всички класове, производни на класа, който я съдържа. Въпреки че на пръв поглед изглежда странно, static член-про- менливата съществува преди създаването на обект от класа. Всъщ¬ност static член-променливата представлява глобална променлива, чиято област на видимост е ограничена до класа, в който е деклари¬рана. В един от следващите примери ще видите, че е възможно да получите достъп до static член-променлива без да използвате обект.

43. Operator Overloading. What operator overloading is? Which classes must support operator overloading? What can and can't be overloaded?

В C++ всяка съществуваща унарна или бинарна операция, в която участва поне един обект от даден клас, може да бъде предефинирана от програмиста. Това дава възможност класовете да бъдат интерпретирани, като нови типове данни, които могат да се използват по начин аналогичен на базовите типове. Предефинирането на операции се осъществява чрез операторни функции. Операторната функция е член-функция или приятелска функция на класа, за който тя е дефинирана. Общата форма на една член-функция оператор е следната:

```
<тип-резултат> <име-на-клас>::operator<#>(<аргументи>)
{
        //изпълнявана операция
}
```

Типът на връщания резултат най-често е същият като типа на класа, за който е дефиниран операторът. Операторът, който се предефинира, замества # в конструкцията. Когато една операторна член-функция предефинира бинарен оператор, функцията ще приема само един параметър. Този параметър ще получава обекта, който е от дясната страна на оператора. Обектът

от лявата страна е този, който генерира обръщението към операторната функция и точно той се предава неявно на функцията посредством указателя this.
Операторната функция operator=() връща *this, т.е. тя връща обекта, на който се присвоява стойност. Връщайки като резултат *this, предефинираният оператор за присвояване позволява на обектите да бъдат използвани в поредица от присвоявания. Например:

o3 = o2 = o1;

При предефиниране на операциите + и – използвахме псевдоними да предаване на стойността. В други случаи може да се наложи да ги предавате по стойност (както направихме с оператора =).
При унарните оператори ако оператора се намира пред операнда (напр. ++o3), то се извиква функцията operator#(). Ако оператора е след операнда (напр. o3++), то се извиква функцията operator#(int i). В този случай на „i“ винаги ще се предава стойност 0.

45. Example: overloading arithmetic operators

В C++ всяка съществуваща унарна или бинарна операция, в която участва поне един обект от даден клас, може да бъде предефинирана от програмиста. Това дава възможност класовете да бъдат интерпретирани, като нови типове данни, които могат да се използват по начин аналогичен на базовите типове. Предефинирането на операции се осъществява чрез операторни функции. Операторната функция е член-функция или приятелска функция на класа, за който тя е дефинирана. Общата форма на една член-функция оператор е следната:

```
<тип-резултат> <име-на-клас>::operator<#>(<аргументи>)
{
 //изпълнявана операция
}
```

Типът на връщания резултат най-често е същият като типа на класа, за който е дефиниран операторът. Операторът, който се предефинира, замества # в конструкцията. Когато една операторна член-функция предефинира бинарен оператор, функцията ще приема само един параметър. Този параметър ще получава обекта, който е от дясната страна на оператора. Обектът от лявата страна е този, който генерира обръщението към операторната функция и точно той се предава неявно на функцията посредством указателя this.
Операторната функция operator=() връща *this, т.е. тя връща обекта, на който се присвоява стойност. Връщайки като резултат *this, предефинираният оператор за присвояване позволява на обектите да бъдат използвани в поредица от присвоявания. Например:

o3 = o2 = o1;

При предефиниране на операциите + и – използвахме псевдоними да предаване на стойността. В други случаи може да се наложи да ги предавате по стойност (както направихме с оператора =).
При унарните оператори ако оператора се намира пред операнда (напр. ++o3), то се извиква функцията operator#(). Ако оператора е след операнда (напр. o3++), то се извиква функцията p