`Дефиниране на класове. Обекти. Конструктори. Деструктори. Указатели към обекти на класове. Масиви и обекти. Създаване и разрушаване на обекти и класове. Инициализиране на обекти на класове. Наследяване.

Обект — обединение между данните и функциите, предназначени за обработването на тези данни.
Метод — предоставя услуга, характерна за даден обект.
Съобщение — заявка за изпълнение на даден метод.
Клас — шаблон, по който се създават обекти.
Инстанция — обект, който принадлежи на даден клас.
Капсулация — скриване на вътрешното устройство на обектите.
Наследяване — механизъм, който позволява повторно използване на спецификациите на класовете.
Йерархия от класове — дървовидна структура, която отразява наследствените взаимоотношения между класовете.
Полиморфизъм — възможността на различни класове да отговарят на едни и същи съобщения по различен начин.

КЛАС
ДЕФИНИРАНЕ НА КЛАСОВЕ

Най-важната функционална възможност на C++ е класът.
Класът е механизмът, който се използва за създаване на обекти.

Декларация на клас:

```
class    име на клас {
private //функции и променливи
public //функции и променливи
} списък от обекти;
```

Списък от обекти не е задължително да се декларира веднага с декларацията на класа, това може да стане и по-нататък.
Функциите и променливите, които са декларирани в рамките на декларацията на един клас, се наричат членове на този клас.
По подразбиране private са частни, т.е. те са достъпни само за други членове на този клас.
За да се декларират публично членовете на един клас се използва ключовата дума public, следвана от двоеточие. Те са достъпни както за членовете на този клас, така и за всички други части от програмата съдържаща този клас.

Пример 1:

```
class myclass {
int a; //това е private променлива
public:
  void set_a(int num); //public функция set_a, декларирана с прототипа си
  int get_a();  //public функция get_a, декларирана с прототипа си
};
```

Функциите, декларирани като част от клас се наричат член-функции.
Тъй като променливата **a** е **private**, тя не е достъпна извън **myclass**. Но тъй като **set_a()** и **get_a()** са членове на **myclass**, то те имат достъп до **a**. Тъй като **set_a()** и **get_a()** са **public**, те могат да бъдат извиквани от всяка част на програмата която съдържа **myclass.**
Въпреки, че **set_a()** и **get_a()** са декларирани от **myclass** те не са дефинирани.

Дефиниране на член функция

За да дефинирате дадена член-функция, трябва да свържете типа на класа с името на функцията. Това се прави, като разделте името на класа и името на функцията с двойно двуеточие(**::**)
Двойното двоеточие **::** се нарича **оператор за определяне областта на видимост.**

Пример: Дефиниране на функциите set_a() и get_a():

```
void myclass::set_a(int num)
{
  a = num;
}
int myclass::get_a()
{
  return a;
}
```

2 ОБЕКТИ

Обектът е инстанция на даден клас. Обектът притежава идентичност (уникален екземпляр).
Класът дефинира както интерфейса, така и реализацията на множество обекти. Класът определя поведението на всички негови обекти.

След като обектът се създаде, той не може да променя класа към който принадлежи.

Създаване на обект - за да създадете обект, използвайте името на класа като спецификатор за типа.

Пример: Следващият ред създава 2 обекта от тип **myclass**
myclass ob1, ob2;

Запомнете: Декларацията на клас е логическа абстакция , която дефинира нов тип. Тя определя как ще изглежда един обект от този тип. Декларацията на обект създава физическа единица от такъв тип. Тоест, един обект заема памет, докато дефиницията на тип(клас) – не. След като веднъж е създаден обект от клас, вашата програма може да се обръща към неговите public членове посредством оператора (.)

Пример 1 :Напишете програма, която задава стойности на член-променливата a на класовете ob1 и ob2, след като извежда стойностите на a за всеки обект.

```
#include <iostream>
using namespace std;
class myclass {
  int a; //частна променлива
public:
  void set_a(int num); //декларира функция set_a
  int get_a(); //декларира функция get_a
};
void myclass::set_a(int num)  //дефиниране на член функцията set_a()
{
  a = num; //тяло на функцията set_a
}
int myclass::get_a() //дефиниране на член функцията get_a()
{
  return a; //тяло на функцията get_a
}
int main()  //главна функция
{
  myclass ob1, ob2;  //декларира обектите ob1 и ob2
  ob1.set_a(10); //задава на a от ob1  стойност 10
  ob2.set_a(99); //задава на a от ob2  стойност 99
  cout << ob1.get_a() << endl; //изкарай на екран 10
  cout << ob2.get_a() << endl; //изкарай на екран 99
  return 0;
```

```
}
```
Резултат:
10
99

Запомнете: Всеки обект от даден клас притежава свое собствено копие от всяка променлива, декларирана в този клас.

Пример 2: В същата програма от Пример 1 се опитайте да получите достъп до private променливата int a:

```
#include <iostream>
using namespace std;
class myclass {
  int a; //private за myclass
public:
  void set_a(int num); //декларира функция set_a
  int get_a(); //декларира функция get_a
};
void myclass::set_a(int num) //дефиниране на член функцията set_a()
{
  a = num; //тяло на функцията set_a
}
int myclass::get_a() //дефиниране на член функцията get_a()
{
  return a; //тяло на функцията get_a
}
int main()
{
  myclass ob1, ob2;

 ob1.a = 10; // ГРЕШКА! Не може да се извърши достъп до private a
  ob2.a = 99; //ГРЕШКА! Не може да се извърши достъп до private a
  cout << ob1.get_a() << "\n";
  cout << ob2.get_a() << "\n";
  return 0;
}
```
Резултат:
BUILD FAILED (exit value 2, total time: 916ms)

Пример 3:Както могат да съществуват public член-функции, така могат да съществуват и public член-променливи.
В тази програма a е декларирана в public частта на myclass и така обръщението към нея е възможно от която и да е част на програмата

```
#include <iostream>
using namespace std;
class myclass {
public:
   int a; //сега a е public
  // така, че   set_a() и  get_a() не са необходими !!!
};
int main()
{
  myclass ob1, ob2;
  ob1.a = 10; //сега имаме директен достъп до a
  ob2.a = 99; //сега имаме директен достъп до a
  cout << ob1.a << "\n";
  cout << ob2.a << "\n";
  return 0;
}
```

Пример 4: Програмата създава клас card, който има структура на библиотечен картон. Класа съхранява името, автора и броя на наличните екземпляри от книгата. Името и автора са променливи от тип низ, а броя на наличните екземпляри – целочислен тип.
Тук е използвана public член-функция store, за да съхрани информация за книгата, и public член-функция show, която показва информацията. Добавянето на функцията main демонстрира класа.

```
#include <iostream>
#include <cstring>
using namespace std;
class card {
  char title[80]; // име на книгата
  char author[40]; // автор
  int number; // наличност
public:
  void store(char *t, char *name, int num); //член-функция store
  void show(); //член-функция show
};
void card::store(char *t, char *name, int num) //дефин. на чл.ф. store
{
  strcpy(title, t);
  strcpy(author, name);    //тяло на чл.ф. store
  number = num;
}
void card::show() //дефиниране на чл.ф show
{
  cout << "Title: " << title << "\n";
```

```
  cout << "Author: " << author << "\n";     //тяло на чл.ф. show
  cout << "Number on hand: " << number << "\n";
}
int main()   //сега демонстрираме класа
{
  card book1, book2, book3; //създаваме обектите

  book1.store("Dune", "Frank Herbert", 2); //извикваме функцията store
  book2.store("The Foundation Trilogy", "Isaac Asimov", 2);
  book3.store("The Rainbow", "D. H. Lawrence", 1);
  book1.show(); //извикваме функцията show
  book2.show();
  book3.show();
  return 0;
}
```

Резултат:

```
Title: Dune
Author: Frank Herbert
Number on hand: 2
Title: The Foundation Trilogy
Author: Isaac Asimov
Number on hand: 2
Title: The Rainbow
Author: D. H. Lawrence
Number on hand: 1
```

Пример 5: Създайте клас, който да съдържа име и адресна информация. Съхранявайте информацията в символни низове, които са private членове на класа. Включете public член-функция, която да запазва името и адреса. Включете и public член-функция, която да извежда името и адреса (наречете тези функции store() и display().)

```
#include<iostream>
#include<cstring>
using namespace std;
class addr{   //създаване на класа
   char name[40]; //private членове
   char street[40];
   char city[30];
   char zip[10];
public: //public член-функции
   void store(char *n,char *s, char *c, char *z); //чл.ф-я запазваща име и адрес
   void display(); //чл.ф-я извеждаща име и адрес
```

```
};
void addr::store(char* n, char* s, char* c, char* z)  /*дефиниране на член ф-я запазваща името и адреса*/
{
   strcpy(name, n);
   strcpy(street,s);
   strcpy(city,c);
   strcpy(zip,z);
}
void addr::display()/*дефиниране на член ф-я извеждаща името и адреса*/

{
   cout<<name<<" ";
   cout<<street<<" ";
   cout<<city<<" ";
   cout<<zip<<" ";
}
int main()
{
   addr a;  //създаване на обект а –копие на класа
   a.store("Kaufland","bul. Ovcha Kupel 136", "Sofia", "1306");
   a.display();

return 0;
}
```

Изход:

Kaufland bul. Ovcha Kupel 136 Sofia 1306

Присвояване на обекти

Един обект може да бъде присвояван на друг при условие, че и двата обекта са от един и същи тип / по принцип, когато един обект се присвоява на друг, се извършва побитово копиране на всички данни./

Пример 6:

```
 // пример за присвояване на обекти
#include <iostream>
using namespace std;
class myclass {
  int a, b;
public:
  void set(int i, int j) { a = i; b = j; }
  void show() { cout << a << ' ' << b << "\n"; }
};
int main()
{
```

```
  myclass o1, o2;
  o1.set(10, 4); /*обектът o1 притежава член-променливите a и b със стойности 10 и 4*/
    o2 = o1; /*присвоява o1 на o2, т.е ст-та на o1 a се присвоява на o2 и текущата ст-т на o1.b се присвоява на o2.b*/
  o1.show();
  o2.show();
  return 0;
}
```

Резултат:
10 4
10 4

Предавне на обекти към функции

Обектите могат да бъдат подавани като аргументи на функции по същия начин, по който се предават и всички други типове данни. Просто декларирайте параметъра на функцията като тип клас и след това използвайте обект от този клас като аргумент при извикването на функцията. Обектите, както и останалите типове данни по подразбиране се подават на функцията по стойност.

Пример 7: Тази програма демонстрира подаването на обект към функция

```
#include <iostream>
using namespace std;
class samp {
  int i;  //целочислена променлива i
public:
  samp(int n) { i = n; }
  int get_i() { return i; }
};
int sqr_it(samp o) //връща стойността на o.i на квадрат
{
  return o.get_i() * o.get_i(); //квадрат
}
int main()
{
  samp a(10), b(2);  //обекти a и b
  cout << sqr_it(a) << endl;
  cout << sqr_it(b) << endl;
  return 0;
}
```

Резултат:
100
4

Връщане на обекти като резултат от функция

За тази цел първо трябва да дефинирате функцията като функция, която връща резултат от тип клас, също така като резултат трябва да върнете обект от декларирания тип, като използвате обикновената конструкция return.

Пример 8: Тази програма демонстрира функция, която като резултат връща един обект

```
// Връща обект
#include <iostream>
#include <cstring>
using namespace std;

class samp {
  char s[80];
public:
  void show() { cout << s << endl; }
  void set(char *str) { strcpy(s, str); }
};
samp input() //връща обект от тип samp
{
  char s[80];
  samp str;
  cout << "Enter a string: ";
  cin >> s;
  str.set(s);
  return str;
}
int main()
{
  samp ob;
  ob = input(); //присвоява върнатия обект на ob
  ob.show();
  return 0;
}
```

Резултат:
Enter a string: het464gbdbd

het464gbdbd

3 . КОНСТРУКТОРИ И ДЕСТРУКТУРИ. СЪЗДАВАНЕ И РАЗРУШАВАНЕ НА ОБЕКТИ НА КЛАСОВЕ

Конструктори –има същото име като класа, към който принадлежи и не притежава тип на връщан резултат.
Конструктора е необходим за инициализация при решаване на реални задачи. Конструкторът на един клас се извиква всеки път когато се създава обект от този клас.

Пример 1:

```
#include <iostream>
using namespace std;

class myclass {
  int a; //private променлива
public:
  myclass(); //  декларира конструктор
  void show(); // декларира функцията show()
};
myclass::myclass() //деф. конструктора –той няма тип на връщан резулт!
{
  cout << "In constructor\n"; //тяло на конструктора
  a = 10;
}
void myclass::show() //дефинира функцията show()
{
  cout << a;  //тяло на функцията show() отпечатва a  на екрана
}
int main() //демонстрира класа
{
  myclass ob; //конструктора се извиква при създаване на обекта ob
  ob.show();  //функц. за отпечатване на a на екарана става чрез констр.
  return 0;
}
```

Резултат: а която има стойност 10 се инициализира от конструктура

```
In constructor
10
```

За глобалните обекти, конструкторът се извиква веднъж – когато започне изпълнението на програмата , за локалните обекти-всеки път, когато се изпълнява конструкцията за деклариране на променлива.

Деструктор: противоположен на конструктора

Тази функция се извиква, когато се унищожава обект. Това е наложително, защото при създаването си обекта заделя памет, и тя трябва да се унищожи при освобождаването на обекта. Името на деструктура на един клас е името на класа предшествано от ~

Пример 2: Тази програма демонстрира деструктур

```
#include <iostream>
using namespace std;
class myclass {
  int a;
public:
  myclass(); // конструктор
  ~myclass(); // деструктор
  void show();
};
myclass::myclass()
{
  cout << "In constructor\n";
  a = 10;
}
myclass::~myclass() //дефинира деструктора
{
  cout << "Destructing...\n";  //тялото на деструктора
}
void myclass::show()
{
  cout << a << "\n";
}
int main()
{
  myclass ob;
  ob.show();
  return 0;
}
```

Резултат:

```
In constructor
10
```

Деструктурът на един клас се извиква при унищожаването на обект от този клас.

Локалните обекти се унищожават, когато излязат извън областта на видимост.

Глобалните обекти се унищожават когато завърши изпълнението на програмата.

Програма 3 . Тази програма използва обект от класа **timer**, за да определи времето между създаването на един обект от класа timer , за да определи времето между създаването на един обект от класа **timer** и неговото унищожаване. Изминалото време се извежда, когато се извика деструктура на обекта. Можете да използвате подобен обект, за да измерите времето за изпълнение на една програма или времето , изразходвано от една функция в даден блок.
Само се уверете, че обектът излиза от областта на видимост в момента, в който искате да приключи времевия интервал.

```
#include <iostream>
#include <ctime>
using namespace std;

class timer {
  clock_t start;
public:
  timer(); // конструктур
  ~timer(); //деструктур
};
timer::timer()
{
  start = clock();
}
timer::~timer()
{
  clock_t end;
  end = clock();
  cout << "Elapsed time: " << (end-start) / CLOCKS_PER_SEC << "\n";
}
int main() //демонстрира констр. и деструкт.
{
  timer ob;
  char c;

  cout << "Press a key followed by ENTER: ";
  cin >> c;

  return 0;
}
```

Резултат:
Press a key followed by ENTER: d
Elapsed time: 0

Тази програма използва стандартната библиотечна функция **clock(),** която връща броя на тактовите импулси от началото на изпълнението на програмата. Разделяйки тази стойност на **CLOCKS_PER_SEC,** получавате резултата в секунди

Програма 4. Създайте клас stopwatch, който емулира хронометър. Използвайте конструктор, за да установите началната стойност 0 за измереното време. Добавете член-функцията start() и stop(), които съответно да стартират и да спират таймера. Включете и член функцията show(), която да показва измереното време. Също така използвайте и деструктора, който автоматично да показва измереното време при унищожаване на обекта от тип stopwatch (за по-лесно извеждайте времето в секунси).

```
// Stopwatch емулатор
#include <iostream>
#include <ctime>
using namespace std;

class stopwatch {
  double begin, end;
public:
  stopwatch(); //конструктор
  ~stopwatch(); //деструктор
  void start();
  void stop();
  void show();
};

stopwatch::stopwatch()
{
  begin = end = 0.0;
}

stopwatch::~stopwatch()
{
  cout << "Stopwatch object being destroyed...";
  show();
}
void stopwatch::start()
{
  begin = (double) clock() / CLOCKS_PER_SEC;
}
```

```
void stopwatch::stop()
{
  end = (double) clock() / CLOCKS_PER_SEC;
}
void stopwatch::show()
{
  cout << "Elapsed time: " << end - begin;
  cout << "\n";
}
int main() //демонстрира
{
  stopwatch watch;
  long i;
  watch.start();
  for(i=0; i<320000; i++) ; // time a for loop
  watch.stop();
  watch.show();
  return 0;
}
```

Резултат:
Elapsed time: 0.015
Stopwatch object being destroyed...Elapsed time: 0.015

Програма 5: Показва реда на създаване и унищожаване на конструкторите на 2 базисни класа

```
#include <iostream>
using namespace std;

class BaseClass1 {
public:
  BaseClass1() { cout << “Constructing BaseClass1\n”; }
  ~BaseClass1() { cout << “Destructing BaseClass1\n”; }
};

class BaseClass2 {
public:
  BaseClass2() { cout << “Constructing BaseClass2\n”; }
  ~BaseClass2() { cout << “Destructing BaseClass2\n”; }
};

class DerivedClass: public BaseClass1, public BaseClass2 {
public:
```

```
  DerivedClass() { cout << “Constructing DerivedClass\n”; }
  ~DerivedClass() { cout << “Destructing DerivedClass\n”; }
};

int main()
{
  DerivedClass ob;

  return 0;
}
```

Резултат:

```
Constructing BaseClass1
Constructing BaseClass2
Constructing DerivedClass
Destructing DerivedClass
Destructing BaseClass2
Destructing BaseClass1
```

4. КОНСТРУКТУРИ С АРГУМЕНТИ

Възможно е да се задават аргументи на конструктурите . За тази цел добавете подходящи параметри към декларациите и дефинициите на конструкторите

Пример 1: Демонстрира как конструктура **myclass** приема един параметър. Стойността, предадена на **myclass**, се използва за инициализиране на променливата **a**.

```
#include <iostream>
using namespace std;
class myclass {
  int a;
public:
  myclass(int x); // конструктур с аргумент (int x)
  void show();
};
myclass::myclass(int x)
{
  cout << "In constructor\n";
  a = x;
}
void myclass::show()
{
```

```
  cout << a << "\n";
}
int main()
{
  myclass ob(4);
  ob.show();
  return 0;
}
```

Резултат:

In constructor
4

Обърнете внимание как е деклариран обектът **ob** в **main().**
Стойността **4,** посочена в скобите след **ob(),** е аргументът, който се предава на **myclass** за инициализиране на **a**

За разлика от конструктурите, деструктурите не могат да приемат аргумент, тъй като не съществува механизъм, чрез който да се предават аргументи на обект, който се унищожава.

Пример 2: Тази програма демонстрира често срещано явление – конструктор да получава повече от един аргумент /в този случай 2/

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  myclass(int x, int y); // конструктур
  void show();
};
myclass::myclass(int x, int y)
{
  cout << "In constructor\n";
  a = x;
  b = y;
}
void myclass::show()
{
  cout << a << ' ' << b << "\n";
}
int main()
{
  myclass ob(4, 7);
```

```
  ob.show();
  return 0;
}
```

Пример 3: Тази програма показва как да предадете аргумент на конструктора на един производен клас

```
#include <iostream>
using namespace std;
class base {
public:
  base() { cout << "Constructing base class\n"; }
  ~base() { cout << "Destructing base class\n"; }
};

class derived : public base {
  int j;
public:
  derived(int n) {
    cout << "Constructing derived class\n";
    j = n;
  }
  ~derived() { cout << "Destructing derived class\n"; }
  void showj() { cout << j << '\n'; }
};

int main()
{
  derived o(10);
  o.showj();
  return 0;
}
```

Резултат:
Constructing base class
Constructing derived class
10
Destructing derived class
Destructing base class

Пример 4: Използвайки следния скелет на програма, създайте конструктори **car**() и **truck**(). Нека те да предават подходящи аргументи на **vehicle**().

В добавка, нека **car**() инициализира **passengers** според показаното в програмата, **truck**() инициализира **loadlimit**, както е показано при създаването на обекта

```
#include <iostream>
using namespace std;

// базисен клас за  различни типове превозни средства
class vehicle {
  int num_wheels;
  int range;
public:
  vehicle(int w, int r)
  {
    num_wheels = w; range = r;  //wheels-бр. колела ; range-km/резервоар
  }
  void showv()
  {
    cout << "Wheels: " << num_wheels << endl;
    cout << "Range: " << range << endl;
  }
};
class car : public vehicle {
  int passengers;
public:
  car(int p, int w, int r) : vehicle(w, r)
  {
    passengers = p;
  }
  void show()
  {
    showv();
    cout << "Passengers: " << passengers << '\n';
  }
};
class truck : public vehicle {
  int loadlimit;  //kg натоварване
public:
  truck(int l, int w, int r) : vehicle(w, r)
  {
    loadlimit = l;
  }
  void show()
  {
```

```
   showv();
   cout << "loadlimit " << loadlimit << '\n';
  }
};
int main()
{
  car c(5, 4, 500);
  truck t(30000, 12, 1200);

  cout << "Car: \n";
  c.show();
  cout << "\nTruck:\n";
  t.show();
  return 0;
}
```

Резултат:
Car:
Wheels: 4
Range: 500
Passengers: 5

Truck:
Wheels: 12
Range: 1200
loadlimit 30000

5.НАСЛЕДЯВАНЕ

Наследяване – механизъм, по който един клас може да наследи свойствата на друг.
Базов клас /base/– наследяваният клас, чиито свойства се наследяват
Производен клас /derived/ - класът, който наследява основните характеристики на базовият клас и добавя свои собствени.

```
class B { //дефинира базов клас
  int i;
public:
  void set_i(int n);
  int get_i();
};

class D : public B { //дефинира производен клас
  int j;
```

```
public:
  void set_j(int n);
  int mul();
};
```

Пример 1: Това е програма, използваща класовете **B** и **D**

```
// Прост пример за наследяване
#include <iostream>
using namespace std;

class base { //дефинира базов клас
  int i;
public:
  void set_i(int n);
  int get_i();
};
class derived : public base { //дефинира производен клас
  int j;
public:
  void set_j(int n);
  int mul();
};
void base::set_i(int n) //задаване на стойности на i в класа base
{
  i = n;
}
int base::get_i() //връщане на стойност на i в класа base
{
  return i;
}
void derived::set_j(int n) //задаване на стойност на j в класа derived
{
  j = n;
}

int derived::mul() //връщане на стойността на i от base умнож. с j
{
  //производният клас може да извиква public член-функции на базовия клас
  return j * get_i();
}

int main()
```

```
{
  derived ob;

  ob.set_i(10); // зареждане на i в base
  ob.set_j(4);  // зареждане на j в derived

  cout << ob.mul();  // извежда 40

  return 0;
}
```

Резултат:
40
Погледнете дефиницията на mul(). Обърнете внимание, че тя извиква get_i(), която е член на базовия клас B а не на D и не е свързана с никой конкретен обект. Това е възможно, защото public членовете на B стават public членове на D. Причината, поради която mul() се обръща към get_i(), вместо директно да опита достъп до i , е, че private членовете на базовия клас(в случая i) си остават private за него и не са достъпни от производния клас. Причината, поради която private членовете на един клас не са достъпни от производните класове се крие в капсулирането

Пример 2. Тази програма дефинира базов клас, който описва определени характеристики на плод. Класът е наследен от два производни класа **Apple** и **Orange**, които добавят специфични характеристики за съответните класове.

```
#include <iostream>
#include <cstring>
using namespace std;
enum yn {no, yes};
enum color {red, yellow, green, orange};
void out(enum yn x);
char *c[] = { "red", "yellow", "green", "orange"};
class fruit { //общ клас за плод fruit
// in this base, all elements are public
public:
  enum yn annual;
  enum yn perennial;
  enum yn tree;
  enum yn tropical;
  enum color clr;
  char name[40];
};
class Apple : public fruit {  // производен клас Apple
  enum yn cooking;
```

```
  enum yn crunchy;
  enum yn eating;
public:
  void seta(char *n, enum color c, enum yn ck, enum yn crchy,
            enum yn e);
  void show();
};
class Orange : public fruit {  //производен клас orange
  enum yn juice;
  enum yn sour;
  enum yn eating;
public:
  void seto(char *n, enum color c, enum yn j, enum yn sr,
            enum yn e);
  void show();
};
void Apple::seta(char *n, enum color c, enum yn ck,
                 enum yn crchy, enum yn e)
{
  strcpy(name, n);
  annual = no;
  perennial = yes;
  tree = yes;
  tropical = no;
  clr = c;
  cooking = ck;
  crunchy = crchy;
  eating = e;
}
void Orange::seto(char *n, enum color c, enum yn j,
                  enum yn sr, enum yn e)
{
  strcpy(name, n);
  annual = no;
  perennial = yes;
  tree = yes;
  tropical = yes;
  clr = c;
  juice = j;
  sour = sr;
  eating = e;
}
void Apple::show()
{
```

```
  cout << name << " apple is: " << "\n";
  cout << "Annual: ";  out(annual); //едногодишно растение
  cout << "Perennial: ";  out(perennial);  //многогодишно растение
  cout << "Tree: "; out(tree);  //расте на дърво
  cout << "Tropical: "; out(tropical);  //тропично
  cout << "Color: " << c[clr] << "\n";  //има цвят
  cout << "Good for cooking: "; out(cooking);  //става за готвене
  cout << "Crunchy: ";  out(crunchy);  //хрупкаво е
  cout << "Good for eating: ";  out(eating);  //става за ядене
  cout << "\n";
}
void Orange::show()
{
  cout << name << " orange is: " << "\n";
  cout << "Annual: ";  out(annual); //едногодишно
  cout << "Perennial: ";  out(perennial);  //многогодишно
  cout << "Tree: ";  out(tree);   //расте на дърво
  cout << "Tropical: ";  out(tropical);   //тропично
  cout << "Color: " << c[clr] << "\n";  //цвят
  cout << "Good for juice: ";  out(juice);  //сочно
  cout << "Sour: ";  out(sour);  //кисело
  cout << "Good for eating: ";  out(eating);   //става за ядене
  cout << "\n";
}
void out(enum yn x)
{
  if(x==no) cout << "no\n";
  else cout << "yes\n";
}
int main()
{
  Apple a1, a2;
  Orange o1, o2;
  a1.seta("Red Delicious", red, no, yes, yes);
  a2.seta("Jonathan", red, yes, no, yes);
  o1.seto("Navel", orange, no, no, yes);
  o2.seto("Valencia", orange, yes, yes, no);
  a1.show();
  a2.show();
  o1.show();
  o2.show();
  return 0;
}
```

Резултат:

Red Delicious apple is:
Annual: no
Perennial: yes
Tree: yes
Tropical: no
Color: red
Good for cooking: no
Crunchy: yes
Good for eating: yes

Jonathan apple is:
Annual: no
Perennial: yes
Tree: yes
Tropical: no
Color: red
Good for cooking: yes
Crunchy: no
Good for eating: yes

Navel orange is:
Annual: no
Perennial: yes
Tree: yes
Tropical: yes
Color: orange
Good for juice: no
Sour: no
Good for eating: yes

Valencia orange is:
Annual: no
Perennial: yes
Tree: yes
Tropical: yes
Color: orange
Good for juice: yes
Sour: yes
Good for eating: no

Програмата демонстрира нещо важно за наследяването: базовият клас не принадлежи единствено на един производен клас, а може да бъде наследен от произволен брой класове.

МАСИВИ И ОБЕКТИ. ИЗПОЛЗВАНЕ НА УКАЗАТЕЛИ КЪМ ОБЕКТИ. ПСЕВДОНИМИ

1. МАСИВИ И ОБЕКТИ

Обектите представляват променливи и имат същите свойства и възможности, както и всеки друг тип променливи. Следователно е абсолютно приемливо да се създават масиви от обекти.

Пример 1: Тази програма създава масив от 4 елемента –обекти от тип **samp**, след което записва в променливата **a** на всеки елемент стойността от 0 до 3.
Обърнете внимание как се извикват член-функциите за всеки отделен елемент от масива. Името на масива **ob** е индексирано, след това се използва оператора за обръщение към член на масива (.) точка, последван от името на извиканата член-функция.

```
#include <iostream>
using namespace std;

class samp {  //клас samp
  int a;  //променлива a
public:
  void set_a(int n) { a = n; } //задава функциите
  int get_a() { return a; }
};
int main()
{
  samp ob[4]; //създава масив от 4 елемента от тип samp
  int i;
  for(i=0; i<4; i++) ob[i].set_a(i);  //обръщение към члена на масива
  for(i=0; i<4; i++) cout << ob[i].get_a( ); /*последван от името на
функцията*/
  cout << "\n";
  return 0;
}
```

Резултат:
0123

Пример 2: Ако даден клас има конструктор, то инициализирането на масив с обекти от този клас може да се осъществи по начина показан по-долу. Тук ob е един инициализиран масив:

```
#include <iostream>
```

```
using namespace std;
class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};
int main()
{
  samp ob[4] = { -1, -2, -3, -4 };
  int i;
  for(i=0; i<4; i++);
 cout << ob[i].get_a() << endl;
  return 0;
}
```

Резултат:

-1 -2 -3 -4

Всъщност синтаксисът, показан в инициализационния списък, представлява съкратен запис на следната по-дълга форма:

```
samp ob[4] = { samp(-1), samp(- 2),
               samp(-3), samp(- 4) };
```

Използваната в програмата форма се среща по-често.

Пример 3. Можете да създавате и многомерни масиви от обекти. Тази програма създава двумерен масив от обекти и ги инициализира

```
#include <iostream>
using namespace std;

class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4][2] = {
    1, 2,
    3, 4,
```

```
   5, 6,
   7, 8
  };
  int i;
  for(i=0; i<4; i++) {
    cout << ob[i][0].get_a()<<" ";
    cout << ob[i][1].get_a() << endl;

  }
   return 0;
}
```

Резултат:

1 2
3 4
5 6
7 8

Както знаете, един конструктор може да приема повече от един аргумент. Когато инициализираме масив от обекти, чиито конструктори приемат повече от един аргумент, трябва да използвате алтернативната форма за инициализация.

Пример 4: В тази програма конструктурът на **samp** приема два аргумента. В този случай масивът ob е деклариран и инициализиран в main(), като се използват директни извиквания към конструктура на samp. Това е необходимо, защото формалния синтаксис на C++ позволява едновременно подаване само на един аргумент от разделен със запетая списък.

```
#include <iostream>
using namespace std;

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; }
  int get_a() { return a; }
  int get_b() { return b; }
};

int main()
{
  samp ob[4][2] = {
    samp(1, 2), samp(3, 4),
    samp(5, 6), samp(7, 8),
```

```
    samp(9, 10), samp(11, 12),
    samp(13, 14), samp(15, 16)
  };

  int i;

  for(i=0; i<4; i++) {
    cout << ob[i][0].get_a() << ' ';
    cout << ob[i][0].get_b() << "\n";
    cout << ob[i][1].get_a() << ' ';
    cout << ob[i][1].get_b() << "\n";
  }
  cout << "\n";
  return 0;
}
```

Резултат:

```
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
```

2. УКАЗАТЕЛИ КЪМ ОБЕКТИ

До сега се обръщахте към членовете на един обект посредством оператора точка (.). Това е правилния метод, но има и друг метод – посредством указател.

Указателят в C++ работо с оператора стрелка (->), а не с точка. Декларирането на указател към обект е абсолютно същото, както деклариране на указател към всеки друг тип променлива. Указава се с името на класа и името на променливата със звездичка пред него.

За да вземе адреса на даден обект, пред неговото име се поставя оператора & по същия начин, по който се взима адреса на променливата.

Пример 1: Тази програма използва указател към обект

Обърнете внимание как декларацията **myclass*p;** създава указател към обект от **myclass**. Важно е да се разбере, че създаването на един указател към обект не създава обект, а просто указател към такъв. Адреса на ob се поставя в p посредством следната конструкция:
p=&ob;

```
#include <iostream>
using namespace std;

class myclass {
  int a;
public:
  myclass(int x); // конструктор
  int get();
};

myclass::myclass(int x)
{
  a = x;
}

int myclass::get()
{
  return a;
}

int main()
{
  myclass ob(120);  // създава обект
  myclass *p;  // създава указател към обект
  p = &ob; // поставяне на адреса на ob в p
  cout << "Value using object: " << ob.get()<<endl;
  cout << "Value using pointer: " << p->get();
  return 0;
}
```

Реазултат:
Value using object: 120
Value using pointer: 120

Забележка: В един от следващите уроци ще разгледаме някои специални свойства сврзани с указателите

Пример 2: Тази програма е пример за адресна аритметика. Както показва изходът от програмата, при всяко инкрементиране на **p**, указателят ще сочи към следващ обект от масива.

```
// Указатели към обекти
#include <iostream>
using namespace std;

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; } //конструктор
  int get_a() { return a; }
  int get_b() { return b; }
};
int main()
{
  samp ob[4] = {
    samp(1, 2),
    samp(3, 4),
    samp(5, 6),
    samp(7, 8)
  };
  int i;
  samp *p;
  p = ob; // взима първият адрес на масива
  for(i=0; i<4; i++) {
    cout << p->get_a() << " ";
    cout << p->get_b() << "\n";
    p++;  // преминава към следващия обект
  }
  cout << "\n";
  return 0;
}
```

Резултат:

1 2
3 4
5 6
7 8

3. УКАЗАТЕЛЯТ *THIS*

this е специален указател, който се предава автоматично към всяка **член-функция** !!!, когато към нея се извърши обръщение. Това е указател към обекта, който генерира обръщението.

Пример: Както знаете, когато една член функция се обръща към даден член на класа, тя го прави директно, без да определя члена със спецификатор за клас или обект.
Тази програма създава прост инвентарен клас:

```
// демонстрация на указателя this
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
  char item[20]; //член променлива item (указва се директно)
  double cost;  //член променлива cost (указва се директно)
  int on_hand;  //член променлива on_hand (указва се директно)
public:
  inventory(char *i, double c, int o) //конструктор
  {
     strcpy(item, i);
     cost = c;
     on_hand = o;
   }
   void show(); //член функция show()
};
void inventory::show()
{
  cout << item;
  cout << ": $" << cost;
  cout << "  On hand: " << on_hand << "\n";
}
int main()
{
  inventory ob("wrench", 4.95, 4);
  ob.show();
  return 0;
}
```

Резултат:
wrench: $4.95 On hand: 4

Както се вижда, в рамките на конструктура inventory() и на член-функцията show(), член-променливите по-горе се указват директно. Това е така, защото една член-функция може да бъде извикана само във връзка с обект. Следователно компилаторът знае към данните на кой обект се прави извикването.

Указателя this има няколко приложения, като едно от тях е много полезно – при предефинирането на оператори.

Домашна работа:
Зад.3 В следната програма променете всички възможни обръщения към членовете на класа така, че да използват явно указателя this:

```
#include <iostream>
using namespace std;
class myclass {
  int a, b;
public:
  myclass(int n, int m) { a = n; b = m; }
  int add() { return a+b; }
  void show();
};
void myclass::show()
{
  int t;
  t = add(); // call member function
  cout << t << "\n";
}
int main()
{
  myclass ob(10, 14);
  ob.show();
  return 0;
}
```

В езика С за динамично управление на паметта се използват функциите malloc() и free().
Работата на malloc() е да задели парче от динамичната памет, а с помощта на free() заделеното парче памет се освобождава.
В езика С++ програмистите могат да използват тези функции, но тяхното използване е неудобно и трябва да се избягва. Проблемът е, че при използването на функцията malloc() не се извикват

конструктори.

ИЗПОЛЗВАНЕ НА NEW И DELETE /за заделяне и освобождаване на памет/

До този момент, винаги когато трябваше да се задели памет, използвахме malloc(), а освобождаването на заделена памет извършвахте с free(). Това са стандартните C функции за динамично заделяне на памет. Въпреки, че тези функции са достъпни и в C++, C++ предлага по-удобен и безопасен метод за заделяне и освобождаване на памет.

В C++ може да заделяте памет с new и да я освобождавате с delete.

Тези оператори имат следната обща форма:

указател-към-тип = new спецификатор-на-тип; //заделяне на памет

delete указател-към-тип; //освобождаване на памет

new е оператор, който връща указател към динамично заделена памет, достатъчно голяма, че да съхрани един обект от този тип
delete е оператор, който освобождава тази памет, когато тя вече не е необходима
Ако извикате **delete** с невалиден указател, ще се унищожи системата за динамично заделяне на памет и програмата ви ще увисне.

Ако няма достатъчно свободна памет в отговор на заявката за динамично заделяне на памет, се извършва едно от следните действия:
а/ new връща **null** указател
б/ new генерира изключение /грешка, която може да бъде обработвана по структуриран начин/

Предимства на new и delete:

- **new автоматично заделя достатъчно памет за съхраняване на посочения тип /не се нуждаете от sizeoff например, за да изчислите броя на необходимите байтове/**
- **new автоматично връща указател от посочения тип /не трябва явно да преобразувате типа, както при употребата на malloc()**
- **new и delete могат да бъдат предефинирани, което ви позволява да имплементирате вашата собствена система за динамично заделяне на памет**
- **възможно е да инициализирате динамично зададен обект**

Пример: Тази програма заделя памет за целочислена стойност

```
// прост пример за  new и delete.
#include <iostream>
using namespace std;
int main()
{
  int *p;
  p = new int; // заделяне на памет за int стойност
  if(!p) {
    cout << "Allocation error\n";
    return 1;
  }
  *p = 1000;
  cout << "Here is integer at p: " << *p << "\n";
  delete p;  // освобождаване на паметта
  return 0;
}
```

Реазултат:
Here is integer at p: 1000

Обърнете внимание, че върната от new стойност се проверява, преди да се използва. Тази проверка изма смисъл само ако вашият компилатор имплементира **new** така, че операторът да връща **null** при неуспех.

Домашна работа:
Зад.4 Създайте клас, който съдържа име на човек и телефонен номер. С помощта на new заделете динамично памет за един обект от този клас и впишете вашето име и телефонен номер в полетата на този обект.

new и delete има няколко допълнителни свойства:

на един динамично заделен обект може да се зададе начална стойност, като се използва следната форма:

указател = new тип (начална стойност);

указател = new тип [размер]; //за масив

след изпълнението на втората конструкция за масив, указател ще сочи към началото на масив от размер на брой елементи от посочения тип.

Поради технически причини, не е възможно да се инициализира динамично заделен масив.
За да освободите динамично заделен масив, използвайте следната форма на delete:

delete [] указател;

Това кара компилатора да извика деструктора за всеки елемент на масива. Това не води до многократно освобождаване на сочената от указателя памет. Паметта, сочена от указателя се освобождава само веднъж.

Пример: Тази програма заделя памет за целочислена стойност и инициализира тази памет. Тя извежда стойност 9, което е началната стойност зададена в паметта, към която сочи **p**

```
// програма за инициализиране на динамична променлива
#include <iostream>
using namespace std;

int main()
{
  int *p;

  p = new int(9); // задава начална стойност 9

  if(!p) {
    cout << "Allocation error\n";
    return 1;
  }

  cout << "Here is integer at p: " << *p << "\n";

  delete p;  // освобождаване на паметта

  return 0;
}
```

Резултат:
Here is integer at p: 9

ПСЕВДОНИМИ

Псевдонимът е неявен указател, който играе ролята на друго име за дадена променлива.
Съществуват 3 начина за предаване на псевдоним:

- **може да бъде предаван на функция /най-важната употреба/**
- **може да бъде връщан като резултат от функция**
- **може да бъде създаван независим псевдоним**

За да обясним използването на псевдоним като указател, ще използваме програмата по-долу в двата варианта – уаказател и псевдоним

1/ с указател:

```
#include <iostream>
using namespace std;

void f(int *n);  // използва указател като параметър
int main()
{
  int i = 0;
  f(&i);
  cout << "Here is i's new value: " << i << '\n';
  return 0;
}
void f(int *n)
{
  *n = 100; // слага 100 в аргумента сочен от n
}
```

В езика С този начин е единственият за извикване на адрес. В С++ обаче, може напълно да се автоматизира този процес, чрез използване на псевдоними. Това е показано в програмата по-долу:

2/ с псевдоним

```
#include <iostream>
using namespace std;

void f(int &n); // деклариране на псевдоним за параметър

int main()
{
  int i = 0;
  f(i);
  cout << "Here is i's new value: " << i << '\n';
```

```
  return 0;
}
void f(int &n) //сега f() използва псевдоним за параметър
{
   n = 100; /* не е необх. * , слага 100 в аргумента, използван за извикване
на  f() */
}
```

Псевдонима се декларира чрез поставяне на **&** пред името на променливата или параметъра. Точно така и n е декларирана като параметър на **f().** Тъй като **n** е псевдоним е недопустимо да се използва оператора *. Вместо това, при всяко извикване на **n** в тялото на **f(),** той автоматично се счита за указател към аргумента, който е използван при извикването на **f().** Това означава, че конструкцията **n=100**; всъщност зарежда стойността **100** в променливата, използвана за извикване на **f().** Следователно, при извикването на **f()** не е необходимо да поставяте **&** пред аргумента. Тъй като в декларацията на **f()** е казано, че за аргумент функцията приема псевдоним, то адресът на този аргумент автоматично се предава на **f**().

ПРЕДЕФИНИРАНЕ НА ФУНКЦИИ
ПРЕДЕФИНИРАНЕ НА ОПЕРАТОРИ

1. ПРЕДЕФИНИРАНЕ ФУНКЦИИ

/предефиниране на конструктор на клас е често срещано явление, предефиниране на деструктур е невъзможно.

Съществуват 3 причини, които налагат предефиниране на конструктор:

- **с цел постигане на гъвкавост**
- **с цел поддържане на масиви**
- **с цел създаване конструктори за копиране**

Забележка: Трябва да се има в предвид, че трябва да има конструктор за всеки начин, по който може да създадете обект от дадения клас. Ако една програма се опита да създаде обект по начин, за който няма съответен конструктор, ще се генерира грешка по време на компилация. Точно за това предефинирането на конструктори е често срещано явление в C++.

Предефинираните конструктури се използват най-често, за да осигурят възможност за избор дали при създаването си даден обект да се инициализира или не.

Пример 1: В тази програма **o1** се инициализира , а **o2** не.

Ако премахнете конструктура с празния списък от аргументи, програмата няма да се компилира, защото не съществува конструктор, който да отговаря за създаване на неинициализиран обект от тип **samp**.
Обратното също е вярно: ако премахнете конструктура, приемащ параметри, програмата няма да се компилира, защото няма конструктур за инициализиран обект. Така, че и двата конструктура са необходими, за да се компилира коректно програмата.

```
#include <iostream>
using namespace std;
class myclass {
  int x;
public:
  // предефиниране на конструктура по два начина
  myclass() { x = 0; } // без инициализация
  myclass(int n) { x = n; } // с инициализация
  int getx() { return x; }
};
int main()
{
  myclass o1(10); // декларация с начална стойност
  myclass o2; // декларация без инициализация
  cout << "o1: " << o1.getx() << '\n';
  cout << "o2: " << o2.getx() << '\n';
  return 0;
}
```

Резултат:
o1: 10
o2: 0

Друга често срещана причина за предефиниране на конструктор е да се позволи съществуването в една програма едновременно на самостоятелни обекти и масиви от обекти.

Например, имайки в предвид класа **myclass** от горния пример, и двете декларации са верни:

```
myclass ob(10);
myclass ob[5];
```

Осигурявайки конструктор с параметри и конструктор без параметри, вашата програма позволява създаването на обекти, които се инициализират или не, в зависимост от конкретния случай.

Пример 2: Тази програма декларира два масива от тип myclass, като единият се инициализира, а другият не.
В този пример всички елементи на o1 се задават със стойност 0 от констуктура . Елементите на o2 се инициализират, както е показано в програмата.

```
#include <iostream>
using namespace std;
class myclass {
  int x;
public:
  // предефиниране на конструктур по два начина
  myclass() { x = 0; } // без инициализация
  myclass(int n) { x = n; } // с инициализация
  int getx() { return x; }
};
int main()
{
  myclass o1[10]; // декларация на масив без инициализация
  // декларация с инициализация
  myclass o2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int i;
  for(i=0; i<10; i++) {
    cout << "o1[" << i << "]: " << o1[i].getx();
    cout << '\n';
    cout << "o2[" << i << "]: " << o2[i].getx();
    cout << '\n';
  }
  return 0;
}
```

Резултат:

```
o1[0]: 0
o2[0]: 1
o1[1]: 0
o2[1]: 2
o1[2]: 0
o2[2]: 3
o1[3]: 0
o2[3]: 4
o1[4]: 0
o2[4]: 5
o1[5]: 0
o2[5]: 6
o1[6]: 0
o2[6]: 7
```

o1[7]: 0
o2[7]: 8
o1[8]: 0
o2[8]: 9
o1[9]: 0
o2[9]: 10

Една от най-важните форми на предефиниране на конструктор е **конструкторът за копие**.
Както показват предишните примери, проблеми възникват, когато трябва да се предаде или върне обект от функция. Създаването и използването на конструктур за копие е един от методите това да се избегне.
Когато дефинирате конструктор за копие, можете да зададете какво точно трябва да се прави, когато се задава копие на даден обект.
C++ дефинира две ситуации, в които стойността на един обект се дава на друг.
Първата е присвояването. Втората е инициализацията
Конструктурът за копие се отнася само за инициализацията и той не се прилага при присвояване.
При инициализация, компилаторът автоматично осигурява точно побитово копие (подразбиращ се конструктор за копие)

Най-общата форма на конструктор за копие е:

```
име-на-клас(const име-на-клас &обект){
//тяло на конструктура
}
```

обект –псевдоним на обект, използван за инициализиране на друг обект
Например, ако имаме клас **myclass()** и обект от този клас **y,** то следната конструкция ще извика конструктора за копие на **myclas()**

```
myclass x = y; // y явно инициализира x
func1(y);      // y предаден като параметър
y = func2();   // y приема върнат обект
```

В първите два случая на конструктура за копие ще се предаде псевдоним на y.
В третия на конструктора за копие се предава псевдоним към обекта, върнат от **func2()**.

Пример 3: Тази програма показва как един конструктур предотвратява проблемите, свързани с предаването на определен вид обекти на функции

```
// тази програма има грешка
#include <iostream>
#include <cstring>
#include <cstdlib>
```

```
using namespace std;

class strtype {
  char *p;
public:
  strtype(char *s);
  ~strtype() { delete [] p; }
  char *get() { return p; }
};

strtype::strtype(char *s)
{
  int l;

  l = strlen(s)+1;

  p = new char [l];
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }

  strcpy(p, s);
}

void show(strtype x)
{
  char *s;

  s = x.get();
  cout << s << "\n";
}

int main()
{
  strtype a("Hello"), b("There");

  show(a);
  show(b);

  return 0;
}
```

В тази програма, когато **strtype** обектът се предава на **show**(), се създава побитово копие (тъй като не е дефиниран конструктор за копие) и то се задава на параметъра **x**. По този начин, когато функцията върне резултат, **x** излиза от областта на видимост и се унищожава. Това разбира се предизвиква извикването на деструктура, който освобождава **x.p**. Но освободената памет е същата, която все още се използва и от обекта, участвал в обръщението към функция. В резултат на което възниква грешка.
Решението на проблема е за класа **strtype** да се дефинира конструктор на копе, който да заделя памет за копието при неговото създаване. Този подход се използва в следващата програма:

```
/* тази програма използва конструктор за копие за да даде възможност
на strtype обектите да бъдат подавани към функция */
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
public:
  strtype(char *s); // конструктор
  strtype(const strtype &o); // конструктор за копие
  ~strtype() { delete [] p; } // деструктор
  char *get() { return p; }
};

// „нормален” конструктор
strtype::strtype(char *s)
{
  int l;

  l = strlen(s)+1;

  p = new char [l];
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }

  strcpy(p, s);
}

// конструктор за копие
```

```
strtype::strtype(const strtype &o)
{
  int l;

  l = strlen(o.p)+1;

  p = new char [l]; // заделя памет за ново копие
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }

  strcpy(p, o.p); // копира низ в копието
}

void show(strtype x)
{
  char *s;

  s = x.get();
  cout << s << "\n";
}

int main()
{
  strtype a("Hello"), b("There");

  show(a);
  show(b);

  return 0;
}
```

Когато **show()** приключва своето изпълнение и x излиза от областта на видимост, паметта, сочена от **x.p**. (която ще бъде освободена) е различна от паметта, която все още се използва от обекта, предаден на функцията.

ПРЕДЕФИНИРАНЕ НА ФУНКЦИЯ ЧРЕЗ OVERLOAD

В първоначалната версия на C++ за предефиниране на функция се изискваше ключовата дума overload.
Тази ключова дума вече е отживелица, но при някои по-стари програми все още може да се срещне.

2. ПРЕДЕФИНИРАНЕ НА ОПЕРАТОРИ

Предефинирането на оператори наподобява предефинирането на функции. Това е просто разновидност на предефинирането на функции, но с някои допълнителни правила.
Когато предефинирате даден оператор, той не губи първоначалното си значение. Вместо това, той придобива допълнителен смисъл, свързан с класа, за който е бил дефиниран.
За да предефинирате един оператор, вие създавате операторна функция. Тази операторна функция е най-често член-функция или приятелска функция на класа, за който е дефинирана.
Но съществува малка разлика между член-функция оператор и приятелска функция оператор.

2.1 предефиниране на оператор чрез използване на член-функция

обща форма:

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

тип-резултат –най-често същият като типа на класа, за който е дефиниран операторът
-операторът, който се предефинира (например, ако предефинираме операторът „+”, то името на функцията ще бъде **operator+**)
списък-аргументи – различно, в зависимост имплементацията и типа на предефинирания оператор
Повечето оператори могат да бъдат предефинирани.
Не могат да бъдат предефинирани операторите на предпроцесора и няколко други като напр. „?”

Пример 1: Тази програма предефинира оператора + за класа coord. Този клас поддържа X,Y координатна система.

```
// предефинира + за класа coord
#include <iostream>
using namespace std;

class coord {
```

```
  int x, y; // координатни стойности
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2);
};
coord coord::operator+(coord ob2) // предефинира + за класа coord

{
  coord temp; /* временният обект е необходим с цел да не се променят нито един от
операндите*/
  temp.x = x + ob2.x;
  temp.y = y + ob2.y;
  return temp;
}
int main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;
  o3 = o1 + o2; // добавя два обекта – извиква се оператор+()
  o3.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";
  return 0;
}
```

Резултат:
(o1+o2) X: 15, Y: 13

Нека да разгледаме по-подробно тази програма:
Функцията **operator+** връща обект от тип **coord**, който съдържа сумата от **X** координатите в **x** и **Y** координатите в **y** за двата операнда.
Обърнете внимание на употребата на временния обект **temp** в тялото на **operator+().** Той съдържа резултата и точно той се връща като резултат. Забележете също, че нито един операнд не е променен. Причината за съществуването на **temp** е лесно разбираема. В тази ситуация операторът **+** е предефиниран по начин, който е съобразен с неговата нормална аритметична употреба. Тук беше важно да не се променя нито един от операндите. Например, когато прибавите 10 към 4 (10+4) резултатът е 14, но нито 10 нито 4 се променят. Точно затова е нужен временния обект.
operator+() връща като резултат обект от тип coord, за да може резултатът от сумирането на coord обекти да бъде използван в по-големи изрази.

2.2 Предефиниране на оператор чрез използване на приятелска функция

Както знаете една приятелска функция не получава this указател. Не можете да използвате приятелска функция за да предефинирате оператора за присвояване, този опеатор може да бъде използван самп от операторна член-функция.

Пример 2: В тази програма **operator+()** е предефиниран за класа **coord** с помощта на приятелска функция

```
// предефинира + за класа coord с помощта на приятелска функция
#include <iostream>
using namespace std;

class coord {
  int x, y; // координатни стойности
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator+(coord ob1, coord ob2);
};
// предефинира използвайки приятелска функция
coord operator+(coord ob1, coord ob2)
{
  coord temp;

  temp.x = ob1.x + ob2.x;
  temp.y = ob1.y + ob2.y;

  return temp;
}

int main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 + o2; // добавя два обекта –извиква се operator+()
  o3.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

Резултат:

(o1+o2) X: 15, Y: 13

Обърнете внимание, че левият операнд се предава на първия параметър, а десният на вторият.

ШАБЛОНИ И ОБРАБОТКА НА ИЗКЛЮЧЕНИЯ

1. ШАБЛОНИ

С помощта на шаблоните може да създавате родови функции и класове. При тях типът на данните се предава като параметър, което дава възможност да използвате една функция или клас с няколко различни типа данни, без да е необходимо да пишете явно специфична версия за всеки отделен тип данни. По този начин шаблоните дават възможност да създадете преизползваем код.

Шаблонът се нарича още универсална функция.
Тази функция се създава с помощта на ключовата дума **template.**
Template създава шаблон, който описва работата на функцията и оставя компилатора да попълни, където е необходимо подробностите

```
template<class Ттип> бр-тип име-на-функция(списък с параметри)
{
//тяло на функцията
}
```

Ттип – е името на типа на данни, използвани от функцията
Въпреки, че употребата на ключовата дума **class** за указване на универсален тип в декларацията на шаблони е традиционна, може да използвате също ключовата дума **typename**.

Пример: Тази програма създава универсална функция, която разменя стойностите на двете променливи, с които е извикана. Тъй като процесът на размяна на две стойности е независим от типа на стойностите, то е добре той да се реализира с помощта на универсална функция

```
// пример за шаблонна функция
#include <iostream>
using namespace std;
// това е шаблонна функция
```

```
template <class X> void swapargs(X &a, X &b)
{
  X temp;
  temp = a;
  a = b;
  b = temp;
}
int main()
{
  int i=10, j=20;
  float x=10.1, y=23.3;
  cout << "Original i, j: " << i << ' ' << j << endl;
  cout << "Original x, y: " << x << ' ' << y << endl;
  swapargs(i, j); // разменя int стойности
  swapargs(x, y); // разменя float стойности
  cout << "Swapped i, j: " << i << ' ' << j << endl;
  cout << "Swapped x, y: " << x << ' ' << y << endl;
  return 0;
}
```

Резултат:

Original i, j: 10 20
Original x, y: 10.1 23.3
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1

Ключовата дума template бе използвана за дефиниране на универсална функция.
Редът **template <class X> void swapargs(X &a, X &b)** казва на компилаторът две неща: създава се шаблон и започва дефиниция на универсална функция. Тук X е универсален тип , който се използва като плейсхолдър. След template частта се декларира функцията swapargs(), като се използва X за тип на стойностите, които ще се разменят. В main() функцията swapargs() се извиква за два различни типа данни: int и float. Тъй като swapargs() е универсална функция, компилаторът автоматично създава две версии на swapargs(0- една, която ще разменя целочислени стойности и друга, която ще разменя стойности с плаваща запетая.

2. **Обработка на изключения**

Обработката на изключения представлява една подсистема на C++, която позволява структурирано и контролирано да обработвате грешки, които се появяват по време на изпълнение.
С помощта на системата за обраб. на изключения в C++ вашата програма може автоматично да извика функция за обработка на грешката при появата на такава. По принцип предимството на системата за обработка на изключения е , че тя автоматизира голяма част от кода за обработка на грешки, който преди това се е писал на ръка във всяка по-голяма програма.

Обработката на изключения в C++ е изградена върху три ключови думи – **try, catch и throw.** В най-общи линии изразите на програмата, които искате да следите за изключения се съдържат в **try** блок. Ако в **try** блока възникне грешка, то се генерира изключение(като се използва **throw).** Изключението се прихваща с помощта на **catch** и се обработва.
Всеки израз, който генерира изключение, трябва да е бил изпълнен в рамките на **try.**

Обща форма:

```
try{
        //try блок
}
catch(тип1 арг){
        //catch блок
}
……….
……….

catch(типN арг){
        //catch блок
}
```

Блокът **try** трябва да обхваща частта от вашата програма, която желаете да следите за грешки. Тя може да бъде сведена до няколко израза в една функция или да бъде разпростряна така, че да обхваща в **try** блок кода на функцията **main** (което означава на практика цялата функция да бъде следена).
Когато се генерира изключение, то се прихваща от съответстващата му **catch** конструкция. В един try блок е възможно да има повече от една catch конструкция. Използваната catch конструкция се определя от типът на изключението. Или казано с други думи, ако типът на на данните, указани в catch конструкцията, съвпада с типа на изключението, съответната catch

конструкция се изпълнява (а останалите се прескачат). Когато се прихване изключение, неговата стойност се получава в арг.

Обща форма на throw конструкцията е:

throw изключение;

Конструкцията throw трябва да бъде изпълнен или от вътрешността на try блока, или от всяка функция, която се извиква от кода в рамките на try блока, изключение е генерираната стойност.
Ако генерирате изключение, за което не съществува съответна catch конструкция, то е възможно да се стигне до ненормално приключване на програмата.

Пример:

```
// прост пример с обработка на изключение
#include <iostream>
using namespace std;

int main()
{
  cout << "start\n";

  try { // начало на try блока
    cout << "Inside try block\n";
    throw 10; // генерира изключение
    cout << "This will not execute";
  }
  catch (int i) { // прихващане на изключение
    cout << "Caught One! Number is: ";
    cout << i << "\n";
  }
  cout << "end";
  return 0;
}
```

Резултат:
start
Inside try block
Caught One! Number is: 10
end

Както виждате от програмата, в нея има **try** блок, съдържащ три израза и конструкция **catch(int i),** която обработва целочислени изключения. В блока **try** само два от трите израза ще бъдат изпълнени: първият cout израз и изразът **throw**. След като се генерира изключението, контролът се предава на конструкцията **catch** и блокът **try** се прекъсва. Казано с други думи, **catch** конструкцията не се извиква. По-скоро изпълнението на програмата се прехвърля към нея. Така изразът **cout**, който се намира след **throw**, никога няма да се изпълни.
След като конструкцията catch се изпълни, програмата продължава своето изпълнение с израза след **catch**. Все пак често блокът catch завършва с извикване на **exit**(), **abort**() или някоя друга функция, която води до приключване на програмата, тъй като обработката на изключения често се използва точно за обработката на катастрофални грешки.