# Класове и обекти в C++

Любомир Чорбаджиев
Технологическо училище "Електронни системи"
Технически университет, София
lchorbadjiev@elsys-bg.org
*Revision* : 1.2

$Date: 2007/11/04 13:35:14 $

## Съдържание

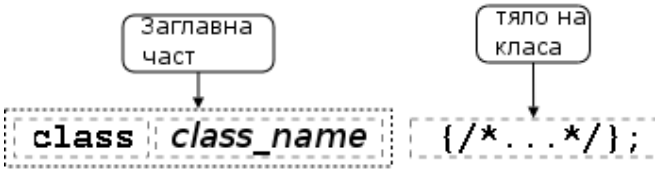
# 1 Дефиниране на класове и обекти в C++

Механизмът на класовете в C++ позволява на програмистите да дефинират техни собствени типове от данни. Поради тази причина, класовете често се наричат *типове, дефинирани от потребителя*.

Класовете в C++ разполагат с изключително богати възможности, което позволява дефинираните от потребителя типове да бъдат точно толкова мощни и изразителни, колкото и вградените в езика типове.

## 1.1 Дефиниция на клас

Дефиницията на клас в езика C++ се състои от две части — *заглавна част* и *тяло на класа*. *Заглавната част на класа* се състои от ключовата дума **class**, последвана от името на класа. *Тялото на класа* се разполага след заглавната част и е затворено във фигурни скоби.




Дефиницията на класа трябва да бъде последвана от точка и запетая или от списък от декларации. Например:

```
class Point {/*...*/};
class Rectangle {/*...*/} r1, r2;
```

## 1.2 Тяло на класа

В тялото на класа се дефинира списъкът от членове на класа и нивото на достъп до тях. Класовете имат два вида членове: *член-променливи* и *член-функции*.

Тялото на класа дефинира област на действие. Декларирането на членове на класа в тялото на класа въвежда имената им в областта на действие на класа.

## 1.3 Обекти

Дефиницията на класа може да се разглежда като шаблон, по който се създават обекти. Дефинирането на клас създава нов тип в областта на видимост, в която е направена дефиницията. За да се дефинира обект от даден клас, трябва да се дефинира променлива от съответния тип.

След като типът, определен от даден клас, бъде дефиниран, то този тип може да се използва по два начина:

- като се използва ключовата дума **class** последвана от името на класа;
- като се използва само името на класа.

```
class Point p;
Point p2;

Point* ptr=&p;
Point& ref=p;
```

## 1.4 Вложени класове

Клас може да бъде дефиниран в рамките на друг клас. Такъв клас се нарича *вложен* клас.

Дефиницията на вложен клас може да бъде направена в публичната, скритата или защитената секция на обграждащия клас. Ако вложеният клас е дефиниран в публичната секция на класа, то той може да се използва както в рамките на класа, в който е дефиниран, така и навсякъде в програмата. Трябва да се има предвид обаче, че когато вложен клас се използва извън рамките на класа, в който е дефиниран, е необходимо неговото име да се специфицира допълнително с името на обграждащия клас.

Например, в следващия фрагмент е дефиниран публичен вложен клас `Bar`. Когато използваме този клас в рамките на обграждащия клас е достатъчно като име на класа да използваме `Bar` — виж ред 5. Когато вътрешен клас се използва извън рамките на обграждащия го клас, неговото име трябва да се квалифицира с името на обграждащия клас — виж ред 8.

```
1 class Foo {
2 public:
3   class Bar {/*...*/};
4 private:
5   Bar bar_;
6   //...
7 };
8 Foo::Bar bar;
```

## 2 Член-променливи и член-функции

Членовете на класа се дефинират в рамките на тялото на класа. Класовете могат да имат два вида членове: *член-променливи* и *член-функции*.

### 2.1 Член-променливи

Дефинирането на член-променливите на класа е аналогично на дефинирането на променливи. За да стане една променлива член на класа, то тя трябва да бъде дефинирана в областта на действие на класа. С други думи, ако една променлива се дефинира в тялото на даден клас, то тя се превръща в член-променлива на класа.

Например, променливите `x_` и `y_`, дефинирани в ред 2 и 3, стават член-променливи на класа `Point`, защото са дефинирани в неговото тяло.

```
1 class Point {
2   double x_;
3   double y_;
4 };
```

Аналогично променливите `x_`, `y_`, `width_` и `height_` са член-променливи на класа `Rectangle`.

```
1 class Rectangle {
2   double x_, y_, width_, height_;
3 };
```

За разлика от нормалните променливи обаче член-променливите не могат да бъдат инициализирани при тяхното дефиниране.[1] Следователно, следният фрагмент съдържа грешка в ред 2:

[1] Това правило има изключение. Ако член-променливата е статична и константна, то тя може да бъде инициализирана още при нейното дефиниране.

```
class Foo {
  int bar_=42;
};
```

Причината за тази разлика е, че при дефинирането на член-променлива не се заделя памет. Заделянето на памет и инициализирането на член-променливите се извършва едва при създаването на обект от дадения клас. Поради това за инициализиране на член-променливите на обектите от даден клас се грижи специализирана член-функция — *конструктор* — която се вика автоматично при създаването на всеки обект.

## 2.2 Член-функции

*Член-функциите* реализират множеството от операции, които могат да се извършват върху обектите от даден клас. Идеологията на обектно-ориентираното програмиране налага правилото, че всички операции върху член-променливите трябва да се извършват от член-функциите на класа.

За да стане една функция член на класа, тя трябва да бъде декларирана в тялото на класа. Например функцията `set_x()`, декларирана в следващия фрагмент, е член-функция на класа `Point`.

```
class Point {
//...
  void set_x(double x);
};
```

Член-функциите могат да се дефинират в тялото на класа.

```
class Point {
  double x_, y_;
public:
  void set_x(double x){x_=x;}
};
```

## 2.3 Модификатори за достъп до членовете на класа

*Капсулирането (скриването на информацията)* е механизъм който предпазва вътрешното представяне на данните. В обектно-ориентирането програмиране капсулирането е основна техника, която се използва за разпределяне на отговорностите между различните части на програмата.

Класовете в C++ имат силно развит механизъм за скриване на информацията. В основата му са спецификаторите за достъп — **public**, **private** и **protected**. Чрез спецификаторите за достъп тялото на класа се разделя на секции — *публична* (**public**), *скрита* (**private**) и *защитена* (**protected**). В зависимост от това в коя секция е дефиниран даден член на класа, той се превръща в *публичен*, *скрит* или *защитен*.

*Публичните членове* на класа са достъпни от всички точки на програмата. *Скритите членове* на класа са достъпни само в член-функциите на класа и в *приятелите* на класа. *Защитените членове* се държат като публични за членовете на производните класове и като скрити за всички останали точки на програмата.

Като пример нека разгледаме класа `Point`, дефиниран по следния начин:

```
1 class Point {
2   double x_, y_;
3 public:
4   void set_x(double x){x_=x;}
5 };
6 Point p1, p2;
7 p1.set_x(10.0);
8 p2.x_=10.0;// грешка
```

В ред 7 променяме състоянието на обекта `p1` като се обръщаме към публичен член на класа — член-функцията `set_x()`. Като резултат стойността на скритата член-променлива `p1.x_` става 10. Ако се опитаме обаче директно да променим стойността на скрита член-променлива (вж. ред 8), то компилаторът ще даде съобщение за грешка.

## 2.4 Обекти

Дефиницията на класа може да се разглежда като шаблон, по който се създават обекти. При дефиниране на променлива от типа на даден клас се създава обект (екземпляр, инстанция) от класа. Всеки обект притежава собствено копие на всички нестатични член-променливи на класа. При създаването на обект се създават и екземпляри на всички член-променливи на класа, които стават "собственост" на създадения обект.

За разлика от член-променливите, всички обекти си поделят само едно копие на член-функциите на класа. Независимо от броя на обектите в програмата има само едно копие на член-функциите на класа.

Нека като пример разгледаме класа `Point` дефиниран по следния начин:

```
1 class Point {
2   double x_, y_;
3 public:
4   void set_x(double x) { x_=x;}
5   double get_x(void) {return x_;}
6 };
```

Нека са дефинирани два обекта `p1` и `p2` от типа `Point`. Всеки от тези обекти притежава собствено копие от нестатичните член-променливи `x_` и `y_`. Когато методът `get_x()` (или `set_x()` се извиква чрез обекта `p1`, то използваната в метода член-променливата `x_` принадлежи на обекта `p1`. Аналогично за `p2`.

```
1 Point p1, p2;
2 p1.set_x(10);
3 p2.set_x(20);
4 p1.get_x();
5 p2.get_x();
```

Като се използва методът `set_x()`, се модифицира стойността на член-променливата `x_` за съответния обект — в ред 2 член-променливата `x_` на обекта `p1` получава стойност `10`, а в ред 3 член-променливата `x_` на обекта `p2` получава стойност `20`.

## 2.5 Структури и класове

В езика `C++` структурите и класовете са тясно свързани. Съгласно определението структурата е клас, за който по подразбиране всички членове са публични. Това означава, че следните две дефиниции са еквивалентни:

```
class s {
public:
  //...
};
```

```
struct s {
  //...
};
```

Във всяко друго отношение структурите се държат като класове – за тях е възможно да се дефинират член-функции, конструктори и деструктори. Например, следните дефиниции са еквивалентни:

```
class Foo1 {
  int bar_;
public:
  Foo1(int bar);
  int get_bar(void);
};
```

```
struct Foo2 {
private:
  int bar_;
public:
  Foo2(int bar);
  int get_bar(void);
};
```

## 2.6 Приятели на клас

В C++ механизмите за контрол на достъпа до членовете на класа могат да бъдат заобиколени като се използват приятелските класове и функции. Механизмът на *приятелите* позволява да се даде достъп до скритата част на класа на определени функции и класове.

За да се дефинира, че дадена функция или клас са приятелски се използва ключовата дума **friend**. Например, в следната дефиниция функцията `dump()` се декларира като приятелска за класа `Point`.

```
1 class Point {
2   double x_,y_;
3
4   friend void dump(const Point& p);
5 public:
6   //...
7 };
```

Тази дефиниция предоставя на функцията `dump()` достъп до скритите и защитените членове на класа `Point`. Например, в следващия фрагмент в ред 2 напълно коректно се използват директно скритите член-променливи на класа `Point`.

```
void dump(const Point& p) {
  cout << p.x_ << " " << p.y_ << endl;
}
```

## 2.7 Указател this

Всяка член-функция има достъп до указател, който е насочен към обекта, за който тази член-функция е извикана. Това е указателят **this**.

Например, нека са дефинирани два обекта `p1` и `p2` от типа `Point`. Ако член-функцията `get_x()` е извикана за обекта `p1`, т.е. използван е следният оператор:

```
p1.get_x();
```

то указателят **this** ще бъде насочен към обекта `p1`. Следните две дефиниции на член-функцията `get_x()` са еквивалентни:

```
class Point {
  double x_,y_;
public:
  double get_x() {
    return x_;
  }
  //...
};
```

```
class Point {
  double x_,y_;
public:
  double get_x() {
    return this->x_;
  }
  //...
};
```

Програмистът може да използва указателя **this** в член-функциите, за да се обръща към член-променливите (както в предходния пример), но това е излишно.

Най-често указателят **this** се използва, когато даден метод трябва да върне препратка към обекта, за който е извикан.

Например, нека е дефиниран обект `p` от типа `Point` и нека за този обект е използван следния израз:

```
p.set_x(1.0).set_y(2.0);
```

Този израз се изчислява отляво надясно, т.е. най-напред се изпълнява методът `set_x()` и към резултата от този метод се прилага методът `set_y()`.

```
(p.set_x(1.0)).set_y(2.0);
```

За да е възможно подобно поведение е необходимо методът `set_x()` да връща препратка към обекта, чрез който е извикан.

```
1  class Point {
2    double x_;
3    double y_;
4  public:
5    Point& set_x(double x) {
6      x_=x;
7      return *this;
8    }
9    Point& set_y(double y) {
10     y_=y;
11     return *this;
12   }
13 };
```

## 2.8 Дефиниране на член-функции

Когато една член-функция не е дефинирана в тялото на класа, то тя трябва да бъде дефинирана извън тялото на класа. При дефинирането на член-функция извън тялото на класа е необходимо да се укаже нейното пълно име. Като пример нека разгледаме член-функцията `set_x()` на класа `Point`:

```
1  class Point {
2    double x_, y_;
3  public:
4    Point& set_x(double x);
5    //...
6  };
```

Когато дефинираме член-функцията `set_x()` извън тялото на класа `Point` трябва да използваме пълното име на функцията, което включва и името на класа, на който тази функция е член. Пълното име на тази член-функция е `Point::set_x()`.

```
Point& Point::set_x(double x) {
  x_=x;
  return *this;
}
```

## 2.9 Статични член-променливи

Когато една член-променлива на класа не е част от обектите на класа тя се нарича *клас-променлива*. За всеки един клас, този вид променливи имат само по едно копие, независимо от броя на обектите от съответния клас.

В езика C++ за дефинирането на клас-променливи се използва ключовата дума **static**. Поради това в C++ променливите от този вид се наричат *статични променливи*.

Статичните член-променливи се създават и инициализират независимо от това дали се създават обекти. Освен това статичните член-променливи имат само по едно копие за всички обекти от класа. Това означава, че инициализацията на статичните член-променливи не може да се включи в конструкторите на класа. Механизмът за инициализация на статичните член-променливи трябва да се различава от инициализацията на нестатичните член-променливи.

Нека като пример да разгледаме класа `DeepThought`, в който е дефинирана статична член-променлива `ANS`.

```
class DeepThought {
  static int ANS;
  //...
};
int DeepThought::ANS=-1;
```

Инициализацията на статичната член-променлива `ANS` се извършва извън тялото на класа. За целта се използва пълното име на член-променливата `DeepThought::ANS`.

```
int DeepThought::ANS=-1;
```

## 2.10 Статични член-функции

Член-функция, която не се свързва с обектите на класа, се нарича *клас-функция*. При извикването на клас-функция не се използва обект от класа, а самият клас.

В C++ за дефиниране на клас-функции се използва ключовата дума **static**. Поради това обикновено клас-функциите в C++ се наричат *статични* член-функции.

Като пример нека разгледаме класа `DeepThought`, в който е дефинирана статична член-функция `find_the_answer()`.

```
1 class DeepThought {
2   int foo_;
3 public:
4   static void find_the_answer(void);
5 };
```

Статичните член-функции не се свързват с конкретен обект от класа при извикването им. Това означава, че в рамките на една член функция вие нямате директен достъп до нестатичните член-променливи (вж. ред 2).

```
1 void DeepThought::find_the_answer(void) {
2   foo_=8; // грешка! /*@@*/
3   ...
4 }
```

Статичните член-функции имат пълен достъп до всички член-променливи на класа. В рамките на тялото на статичната член-функция имаме пълен достъп до скритите и защитени член-променливи на класа.

```
1 class DeepThought {
2   int foo_;
3 public:
4   static void find_the_answer(void) {
5     DeepThought some_thoughts;
6     some_thoughts.foo_=42; // OK!
7     ...
8   }
9 };
```

Статичните член-функции не се свързват с конкретна инстанция на класа. За да се извика статична член-функция не е необходим обект от класа. Статичните член функции се викат директно, като се използва пълното име на член-функцията:

```
1 DeepThought::find_the_answer();
```

## 2.11 Статични членове: пример

```
#include <iostream>
using namespace std;
class DeepThought {
public:
  static int ANSWER;
  static void find_the_answer(void);
};

int DeepThought::ANSWER=-1;

void DeepThought::find_the_answer(void) {
  // some deep calculations
  ANSWER=42;
}

int main(void) {
  DeepThought::find_the_answer();
  cout << "The␣answer␣is:␣"
       << DeepThought::ANSWER << endl;
  return 0;
}
```

## 2.12 Константни член-функции

Член-функциите могат да бъдат дефинирани като *константни*. Когато една член-функция е константна, това означа, че в рамките на такава функция не може да се променя състоянието на обекта (на неговите член-променливи).

За да се дефинира една член-функция като константна се използва ключовата дума **const**:

```
1 class Foo {
2   int bar_;
3 public:
4   int get_bar(void) const { return bar_; }
5 };
```

Член-функцията `get_bar()`, дефинирана в ред 4, е константна член-функция.

Когато една член-функция е дефинирана като константна, тя не може да променя състоянието на обекта, за който е извикана. В ред 4 на следващия фрагмент е допусната грешка.

```
1 class Foo {
2   int bar_;
3 public:
4   void bar(void) const { bar_++; }// грешка
5 };
```

# 3 Конструктори и деструктори

## 3.1 Конструктори

Член-променливите не могат да се инициализират при тяхната дефиниция. Причината за това е, че при дефинирането на класа не се създават екземпляри на член-променливите му. Това се случва едва когато се създаде обект от дефинирания клас. За всеки създаден обект се създават екземпляри на всички член-променливи на класа, които са свързани със създадения обект. Поради това инициализирането на член-променливите трябва да се извърши при създаване на обекти.

За инициализиране на обектите от даден клас се използва специализирана член-функция, която се нарича *конструктор*. При създаването на всеки обект се вика конструктор, който инициализира член-променливите на обекта. Извикването на конструктора се извършва автоматично при създаването на обект.

Името на конструктора съвпада с името на самият клас. Например, в следващия фрагмент е дефиниран клас `Point`. Конструкторът на този клас е деклариран в ред 4:

```
1 class Point {
2   double x_,y_;
3 public:
4   Point(double x, double y); // конструктор
5   //...
6 };
```

Ако конструкторът има аргументи, то те трябва да се предадат при създаването на обекта. Например:

```
1 Point p1 = Point(1.0,1.0);
2 Point p2(2.0,2.0);
```

```
3 Point p3; // грешка
4 Point p4(4.0); // грешка
```

Има възможност за един клас да се дефинират няколко конструктора, които се различават по аргументите, които им се предават.[2]

Конструктор, който се извиква без аргументи се нарича *конструктор по подразбиране*. Например, в следващия фрагмент в ред 3 е деклариран конструктор, който се извиква с два аргумента, а в ред 4 — конструктор по подразбиране:

```
1 class Point {
2 public:
3   Point(double x, double y);
4   Point(void);
5 };
```

Кой конструктор ще се извика в даден конкретен случай се решава според аргументите, които са предадени при извикването на конструктора. Например, в следващия фрагмент, в ред 1 ще извика конструкторът на `Point`, който приема два аргумента, а в ред 2 ще се извика конструкторът по подразбиране.

```
1 Point p1(1.0,1.0);
2 Point p2;
```

## 3.2 Конструктор по подразбиране

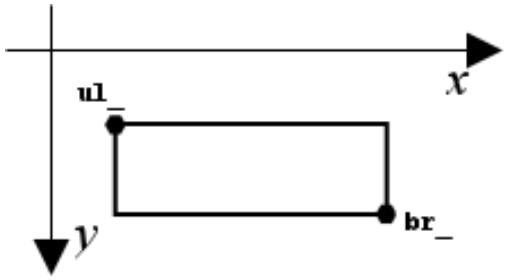
*Конструктор по подразбиране* се нарича конструктор, който може да се извика без да му се предават аргументи.

Ако програмистът не дефинира *никакъв* конструктор на класа, то компилаторът ще генерира конструктор по подразбиране. Поведението на генерирания от компилатора конструктор по подразбиране е да извика конструкторите по подразбиране на всички член-променливи на класа.

## 3.3 Инициализация на член-променливи в конструктора

Нека имаме клас `Point`, дефиниран по следния начин:

[2] В `C++` е допустимо да се дефинират няколко функции с едно и също име, които се различават по броя и типа на аргументите. За да реши коя точно функция трябва се извика, компилаторът използва списъка на предадените аргументи.




Фигура 1: Клас `Rectangle`

```
class Point {
  double x_, y_;
public:
  Point(double x, double y) {
    x_=x; y_=y;
  }
  Point(void) {
    x_=0.0; y_=0.0;
  }
  Point& set_x(double x) { x_=x; return *this; }
  Point& set_y(double y) { y_=y; return *this; }
  double get_x(void) {return x_;}
  double get_y(void) {return y_;}
};
```

Като използваме така дефинирания клас `Point` искаме да дефинираме клас `Rectangle`. Обектите от този клас имат две член-променливи, които съответстват на горния ляв `ul_` и долния десен `br_` ъгъл на правоъгълника. През тези две точки може да се построи само един правоъгълник, чиито страни са успоредни на координатните оси, както е показано на фигура 1.

Нека разгледаме следната дефиниция на класа `Rectangle`:

```
class Rectangle {
  Point ul_, br_;
public:
  Rectangle(double x,double y,double w,double h){
    ul_.set_x(x).set_y(y);
    br_.set_x(x+w).set_y(y+h);
  }
```

```
8 };
```

Обърнете внимание, че при влизане в тялото на конструктора в ред 5, член-променливите вече са създадени. При тяхното създаване е извикан конструкторът по подразбиране на класа `Point`, а в тялото на конструктора ние променяме стойностите на $x$ и $y$-координатите на `ul_` и `br_`.

По-добрият вариант е да накараме компилатора когато създава обектите `ul_` и `br_` да извика не конструктора по подразбиране, а конструктора на `Point` с два аргумента.

Този проблем е общ и неговото решение в `C++` е въвеждането на специален синтаксис, който позволява указването на конструкторите, които трябва да се извикат при създаването на член-променливите на класа.

За нашия конкретен пример, конструкторът на класа `Rectangle` ще изглежда по следния начин:

```
1 class Rectangle {
2   Point ul_, br_;
3 public:
4   Rectangle(double x,double y,double w,double h)
5     : ul_(x,y),
6       br_(x+w,y+h)
7   {}
8 };
```

Преди тялото на конструктора, с двоеточие започва изброяването на конструкторите, които трябва да се извикат за всяка една член-променлива. В ред 5 и ред 6 са указани конструкторите, които трябва да се извикат за член-променливите `ul_` и `br_` съответно.

## 3.4 Конструктори и деструктори

Основната задача на конструктора е да инициализира обекта за да могат член-функциите на обекта да работят правилно. Коректната инициализация на даден обект понякога включва заделянето на динамична памет, отварянето на файлове или използването на някакъв друг ресурс.

Нека разгледаме следния пример:

```
1 class Foo {
2   int size_;
3   int* bar_;
4 public:
```

```
5   Foo(int size)
6     : size_(size), bar_(new int[size])
7   {}
8   //...
9 };
10 int bar() {
11   Foo foo(100);
12   //...
13   return 42;
14 }
```

При създаването на обекта `foo` в ред 11 динамично се заделя памет за масив от 100 цели. Когато променливата `foo` излезе от областта на действие в ред 13, то тя ще се унищожи, заедно с всички нейни член-променливи. Динамично заделената памет, обаче, ще остане неосвободена.

Подобни ситуации се често срещани в практиката. За тяхното решаване е необходимо да има член-функция, която да се държи противоположно на конструкторите — да се извиква автоматично при унищожаване на обектите от даден клас. В `C++` член-функцията, която се вика автоматично при унищожаването на обект от даден клас се нарича *деструктор*.

## 3.5 Деструктори

Деструкторът е член-функция, която се вика автоматично при унищожаване на обект от даден клас, независимо от причините поради които обектът се унищожава — излизане от областта на видимост или освобождаване на динамично създаден обект. Основната задача на деструктора е да освободи ресурсите, използвани от обекта.

За да се дефинира деструктор на даден клас, трябва да се дефинира член-функция, чието име е образувано от името на класа, като пред него е поставена тилда '`~`'. Например, името на деструктора на класа `Foo` е `~Foo()`. Деструкторите нямат тип на резултата и аргументи. За разлика от конструкторите, всеки клас може да има само един деструктор.

Нека се върнем на примера с класа `Foo`. В конструктора на този клас динамично се заделя памет за масив от цели числа. Това означава, че когато създаваме обекта `foo` в ред 14 динамично се заделя памет за масив от 100 цели числа. Когато обектът `foo` излезе от областта на видимост в ред 16 тази памет трябва да се освободи.

Поради тази причина за класа `Foo` се дефинира деструктор `~Foo()`, който освобождава динамично заделената памет — виж ред 8. Тогава,

когато обектът `foo` излезе от областта на видимост в ред 16, автоматично ще се извика неговият деструктор, който от своя страна ще освободи динамичната памет.

```
1  class Foo {
2    int* bar_;
3    int size_;
4  public:
5    Foo(int size)
6      : size_(size), bar_(new int[size])
7    {}
8    ~Foo(void) {
9      delete [] bar_;
10   }
11   //...
12 };
13 int bar() {
14   Foo foo(100);
15   //...
16   return 42;
17 }
```

## 4 Капсулиране на данните: пример

Разгледаните механизми за дефиниране на класове позволяват на програмистите по естествен начин да скриват вътрешната организация на обектите. Манипулирането на обектите от даден клас може да се извършва само чрез публичните методи на класа.

Нека като пример разгледаме дефинирането на клас `Stack`, моделиращ поведението на абстрактния тип стек. Основните операции, които могат да се извършват с един стек са две:

- добавяне на нов елемент в стека — `push()`;
- изваждане на последния добавен елемент от стека — `pop()`.

Поради тази причина, често стекът се нарича FILO (First In, Last Out) — първи влязъл, последен излязъл.

Възможните начини за реализиране на стек са няколко. Най-простият вариант е да се използва масив, като ако е необходимо стекът да е ди-

намичен,[3] може да се използва динамичен масив. Друг вариант за реализирането на стека е да се използва списък — например двусвързан списък.

Независимо от начина на реализация обаче, операциите които могат да се извършват със стека трябва да се едни и същи — `push()` и `pop()`.

## 4.1 Пример: стек, реализиран чрез динамичен масив

```
1  #include <iostream>
2  #include <exception>
3  using namespace std;
4  class Stack {
5    const static int chunk_=2;
6    int size_;
7    int *data_;
8    int top_;
9  public:
10   Stack(void)
11     : size_(chunk_),
12       data_(new int[chunk_]),
13       top_(-1)
14   {}
15   ~Stack(void) {
16     delete [] data_;
17   }
18   void push(int v) {
19     if(top_>=(size_-1)) {
20       resize();
21     }
22     data_[++top_]=v;
23   }
24   int pop(void) {
25     if(top_<0){
26       throw exception();
27     }
28     return data_[top_--];
29   }
```

[3] Броят на елементите, които могат да се поберат в масива да не е фиксиран по време на компилацията на програмата.

```
30 private:
31   void resize(void) {
32     cout << "Stack::resize() called..." << endl;
33     int *temp=data_;
34     data_=new int[size_+chunk_];
35     for(int i=0;i<size_;i++)
36       data_[i]=temp[i];
37     delete [] temp;
38     size_+=chunk_;
39     cout << "Stack::resize() new size is <"
40          << size_ << ">..." << endl;
41   }
42 };
43 int main(void) {
44   Stack st;
45   st.push(1);
46   st.push(2);
47   st.push(3);
48   try {
49     cout << st.pop() << endl;
50     cout << st.pop() << endl;
51     cout << st.pop() << endl;
52   } catch(const exception& e) {
53     cout<<"exception() catched in pop..."<<endl;
54   }
55   return 0;
56 }
```

## 4.2 Пример: стек, реализиран чрез двусвързан списък

```
1  #include <iostream>
2  using namespace std;
3
4  class Stack {
5    struct Elem {
6      Elem* next_;
7      Elem* prev_;
8      int data_;
9
10     Elem(int v)
```

```
11        : next_(0),
12          prev_(0),
13          data_(v)
14      {}
15    };
16
17    Elem* first_;
18    Elem* last_;
19    void appendElem(Elem* newElem) {
20      if(last_==0) {
21        first_=newElem;
22        last_=newElem;
23      }
24      else {
25        last_->next_=newElem;
26        newElem->prev_=last_;
27        last_=newElem;
28      }
29    }
30    Elem* removeLastElem(void) {
31      if(last_==0)
32        return 0;
33      if(last_==first_) {
34        Elem* res=last_;
35        last_=0;
36        first_=0;
37        return res;
38      }
39
40      Elem* res=last_;
41      last_=res->prev_;
42      last_->next_=0;
43
44      res->prev_=0;
45      res->next_=0;
46      return res;
47    }
48  public:
49    Stack(void)
50      : first_(0),
```

```
51        last_(0)
52    {}
53
54    ~Stack(void) {
55      while(last_!=0){
56        Elem* el=removeLastElem();
57        delete el;
58      }
59    }
60    void push(int val) {
61      Elem* newElem=new Elem(val);
62      appendElem(newElem);
63    }
64
65    int pop(void) {
66      Elem* elem=removeLastElem();
67      if(elem==0){
68        cout << "ERROR: stack empty..." << endl;
69        return 0;
70      }
71      int res=elem->data_;
72      delete elem;
73      return res;
74    }
75  };
76
77  int main(void) {
78    Stack st;
79    st.push(1);
80    st.push(2);
81
82    cout << st.pop() << endl;
83    cout << st.pop() << endl;
84    cout << st.pop() << endl;
85    return 0;
86  }
```