

# Кратък обзор на езика C++

Любомир Чорбаджиев<sup>1</sup>  
lchorbadjiev@elsys-bg.org

<sup>1</sup>Технологическо училище “Електронни системи”  
Технически университет, София

29 март 2009 г.



# Съдържание

- 1 Структури
- 2 Класове
- 3 Пространство от имена (*namespace*)
- 4 Входно изходни операции
- 5 Обработка на изключения

# Дефиниране на структура

- *Структурата* представлява съвкупност от една или повече променливи, които могат да от различни типове. Дефиницията на структура има следния синтаксис:
  - Заглавна част, която се състои от ключовата дума **struct** последвана от името на структурата.
  - Тяло, в което се описват членовете на структурата. Тялото на дефиницията е оградено от фигурни скоби и задължително трябва да бъде последвано от точка и запетая ';'.



# Пример за дефиниране на структура

- В следващия фрагмент е дефинирана структурата `person`:

```
1 struct person {  
2     char* name;  
3     int age;  
4 };
```

- Името на структурата `person` се превръща в име на тип и могат да се дефинират променливи.

```
1 person somebody;
```

- За инициализирането на структура се използва запис, подобен на инициализацията на масив.

```
1 person anybody={"pesho",18};
```

# Достъп до членове на структура

- Достъпът до членовете (полетата) на структурата се осъществява с използването на оператора `.` (точка).

```
1 somebody.name="ivan";  
2 somebody.age=16;
```

- Когато достъпът до структурата се извършва чрез указател, то членовете на структурата са достъпни чрез оператора `->`.

```
1 void dump(person* ptr) {  
2     cout << ptr->name << endl  
3         << ptr->age << endl;  
4 }
```

# Класове

- В езика C++ има няколко начина за дефиниране на типове от потребителя. Едната възможност е да се използват разгледаните вече структури **struct**. Другата възможност е да се използват класове.
- Механизмът на класовете в C++ разполага с изключително богати възможности, което позволява дефинираните от потребителя типове да бъдат точно толкова мощни и изразителни, колкото и вградените в езика типове.

# Дефиниция на клас

- Дефиницията на клас в езика C++ се състои от две части — *заглавна част* и *тяло*.



- Пример:

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

- 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*.

# Член-променливи

- За да стане една променлива член на класа, то тя трябва да бъде дефинирана в тялото на класа.
- Пример:

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

- Пример:

```
1 class Rectangle {  
2     Point bl_, ur_  
3 };
```



# Член-променливи

- Член-променливите не могат да бъдат инициализирани при тяхното дефиниране.

```
1 class Foo {  
2     int bar_=42; // Грешка!  
3 };
```

- 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.

# Член-функции

- *Член-функциите* реализират множеството от операции, които могат да се извършват върху обектите от даден клас.
- За да стане една функция член на класа, тя трябва да бъде декларирана в тялото на класа.
- Член-функциите могат да се дефинират в тялото на класа.

```
1 class Point {  
2     ...  
3     void set_x(double x);  
4     int get_x() {return x_;}  
5 };
```

# Модификатори за достъп

- *Капсулирането (скриването на информацията)* е механизъм който предпазва вътрешното представяне на данните.
- Класовете в C++ имат силно развит механизъм за скриване на информацията. В основата му са спецификаторите за достъп — **public**, **private** и **protected**.
- *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.

# Модификатори за достъп: пример

```
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
```

# 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.
- При дефиниране на променлива от типа на даден клас се създава обект (екземпляр, инстанция) от класа. Всеки обект притежава собствено копие на член-променливите на класа.

# Обекти

```
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 };
```

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

# Структури и класове

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

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

```
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);  
};
```

# Конструктори

- Член-променливите не могат да се инициализират при тяхната дефиниция. Инициализирането на член-променливите трябва да се извърши при създаване на обекти.
- За инициализиране на член-променливите 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.



# Конструктори

- Името на конструктора съвпада с името на самият клас.

```
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:

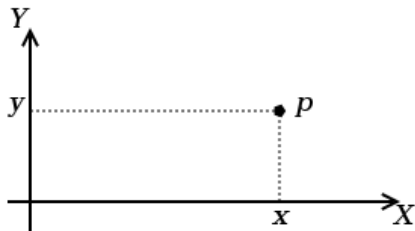
```
1 Point p1 = Point(1.0, 1.0);  
2 Point p2(2.0, 2.0);  
3 Point p3; // грешка  
4 Point p4(4.0); // грешка
```

# Конструктори

- Има възможност за един клас да се дефинират няколко конструктора, които се различават по аргументите, които им се предават.
- Конструктор, който се извиква без аргументи се нарича *конструктор по подразбиране*.

```
1 class Point {  
2 public:  
3     Point(double x, double y);  
4     Point(void);  
5 };  
6 ...  
7 Point p1(1.0,1.0);  
8 Point p2;
```

# Пример: точка в равнината



**Фигура:** Декартови координати на точка в равнината

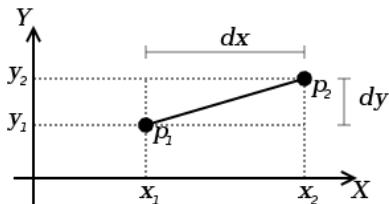
# Пример: точка в равнината

```
1 #include <cmath>
2 using namespace std;
3
4 class Point {
5     double x_, y_;
6 public:
7     double get_x() {return x_;}
8     double get_y() {return y_;}
9     void set_x(double x) {x_=x;}
10    void set_y(double y) {y_=y;}
```

# Пример: точка в равнината

```
1 Point(double x=0.0, double y=0.0) {
2     x_=x;
3     y_=y;
4 }
5
6 void add(Point other) {
7     x_+=other.x_;
8     y_+=other.y_;
9 }
10
11 void sub(Point other) {
12     x_-=other.x_;
13     y_-=other.y_;
14 }
```

# Пример: точка в равнината



```

1  double distance(Point other) {
2      double dx=x_-other.x_;
3      double dy=y_-other.y_;
4      return sqrt(dx*dx+dy*dy);
5  }
6  };

```

# Пример: точка в равнината

```
1 Point add(Point p1, Point p2) {
2     Point result(p1.get_x(),p1.get_y());
3     result.add(p2);
4     return result;
5 }
6
7 Point sub(Point p1, Point p2) {
8     Point result(p1.get_x(), p2.get_y());
9     result.sub(p2);
10    return result;
11 }
12
13 double distance(Point p1, Point p2) {
14     return p1.distance(p2);
15 }
```

# Основни операции със стек

- Основните операции, които могат да се извършват с един стек са:
  - добавяне на нов елемент в стека — `push()`;
  - изваждане на последния добавен елемент от стека — `pop()`.
- Често стекът се нарича FILO (First In, Last Out) — първи влязъл, последен излязъл.



# Реализация на стек

```
1 const int STACK_SIZE=10;  
2 class Stack {  
3     int data_[STACK_SIZE];  
4     int top_;  
5 public:  
6     Stack() {  
7         top_=0;  
8     }
```

# Реализация на стек

```
1 void push(int val) {
2     if(top_ < STACK_SIZE) {
3         data_[top_++] = val;
4     }
5 }
6 int pop(void) {
7     if(top_ > 0) {
8         return data_[--top_];
9     }
10    return 0;
11 }
```

# Реализация на стек

```
1  bool is_empty() {  
2      return top_==0;  
3  }  
4  bool is_full() {  
5      return top_==STACK_SIZE;  
6  }  
7  };
```

# Използване на стек

```
1 int main(int arch, char* argv[]) {  
2     char* msg="Hello!";  
3     char buff[10];  
4     Stack st;  
5     for(char* p=msg;*p!='\0';p++)  
6         st.push(*p);  
7     char* p=buff;  
8     while(!st.is_empty())  
9         *p+=st.pop();  
10    *p='\0';  
11    return 0;  
12 }
```

# Пространство от имена

- Пространствата от имена (*namespaces*) са въведени в C++ като поддръжка на така нареченото модулно програмиране.
- По същество пространствата от имена позволяват изграждането на дървовидна структура от имена на идентификаторите в една C++ програма като по този начин намаляват риска от конфликт на имената.
- Пример: файлова система без директории.

# Пространство от имена

- Когато размерът на една програма започне да нараства — конфликтите на имена започват да стават често явление.
- За решаването на този проблем в C++ са въведени пространствата от имена.
- Пространствата от имена могат да бъдат вложени едно в друго и да образуват йерархични структури от имена, подобни на файловата система. Такава йерархична структура от имена може лесно да предпази кода на една програма от конфликти на имената.

# Дефиниране на пространство от имена

- За дефиниране на именувано пространство от имена се използва ключовата дума **namespace**. Например:

```
1 namespace elsys {  
2     class Student {  
3         ...  
4     };  
5};
```

- Към едно пространство от имена винаги може да се добавят нови имена.

```
1 namespace elsys {  
2     class Teacher {  
3         ...  
4     };  
5};
```

# Използване на пространства от имена

- Идентификаторът може да се квалифицира пълно.

```
elsys::Teacher teacher;
```

- Идентификаторът може да бъде включен в текущата област на видимост, като се използва **using**-дефиниция.

```
using elsys::Teacher;  
Teacher teacher;
```

- В текущата област на видимост могат да се включат всички идентификатори, дефинирани в рамките на дадено пространство от имена като се използва **using**-декларация.

```
using namespace elsys;  
Teacher teacher;  
Student student;
```



# Пространство от имена `std`

- Повечето от типовете, променливите и функциите от стандартната C++ библиотека са дефинирани в пространството от имена `std`.
- Често срещана практика е, да се използва **using**-декларация за включване на идентификаторите от стандартното пространство от имена в текущата област на видимост.

```
#include <cmath>  
#include <cstdlib>  
using namespace std;
```

# Входно/изходни операции

- Тъй като C++ и C са родствени езици, в една програма на C++ е напълно възможно да се използва стандартната C-библиотека за вход и изход.

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]) {
4     printf("Hello world!\n");
5     return 0;
6 }
```

- В C++ входно/изходните операции са организирани като операции с потоци. При разработването на входно/изходната библиотека на C++ специално внимание е обърнато на удобството и лекотата на използване на библиотеката.

# Стандартни потоци за вход и изход

- Стандартните потоци за вход и изход са декларирани в заглавния файл `<iostream>`.
- Потоците за вход и изход и операциите с тях са дефинирани в пространството от имена `std`.
- Стандартният поток за изход е `cout`.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(int argc, char* argv[]) {
5     cout << "Hello world!" << endl;
6     return 0;
7 }
```

# Стандартен поток за изход `cout`

- В един израз могат да се комбинират няколко оператора за изход.

```
cout << "Hello" << " " << "world!" << endl;
```

- С един оператор за изход могат да се извеждат различни типове данни. Например:

```
cout << "The answer is " << 42 << endl;
```

- За извеждане на край на реда се използва `endl`.

# Стандартен поток за вход cin

- Стандартният поток за вход е cin. Операторът за четене от потока е >>.
- Потокът за вход може да обработва последователност от различни по тип променливи.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(int argc, char* argv[]) {
5     int number1, number2;
6     cin >> number1 >> number2;
7     cout << "number1=" << number1 << endl
8         << "number2=" << number2 << endl;
9     return 0;
10 }
```

# Обработка на грешки

- По време на изпълнение на програмата дадена функция може да открие възникването на ненормална, грешна ситуация.
- Причината за възникването на такава ситуация може да бъде различна — неправилни входни данни, препълване на диска, изчерпване на наличната динамична памет, невъзможност да се отвори файл и т.н.
- По какъв начин функцията трябва да реагира на такава ситуация?

# Обработка на грешки

- С-подход: функцията, открила ненормална ситуация да върне резултат, който сигнализира за наличието на грешка.
- Голяма част от функциите в стандартната С библиотека са организирани точно по този начин.

```
FILE* fopen(const char* filename,  
            const char* mode);  
int fputc(int c, FILE* file);  
int fputs(const char* str, FILE* file);  
int fgetc(FILE* file);
```

# Обработка на грешки в класа Stack

Първоначална версия — липсва обработка на грешки.

```
1 class Stack {
2 ...
3 public:
4 ...
5 void push(int val) {
6     if(top_ < STACK_SIZE) {
7         data_[top_++] = val;
8     }
9 }
10 ...
11 };
```



# Обработка на грешки в класа Stack

```
1  int push(int val) {  
2      if(top_ < STACK_SIZE) {  
3          data_[top_++] = val;  
4          return 0;  
5      }  
6      return -1; // Грешка: стека е пълен  
7  }
```

# Обработка на грешки в класа Stack

Първоначална версия — липсва обработка на грешки.

```
1 class Stack {
2 ...
3 public:
4 ...
5     int pop(void) {
6         if(top_>0) {
7             return data_[--top_];
8         }
9         return 0;
10    }
11 ...
12 };
```

# Обработка на грешки в класа Stack

```
1  int pop(int& val) {  
2      if(top_>0) {  
3          val=data_[--top_];  
4          return 0;  
5      }  
6      return -1; // Грешка: стека е празен  
7  }
```

# Обработка на грешки

- Разгледаният подход за обработка на грешки е тежък и тромав.
- При всяко извикване на функция, резултатът от тази функция трябва да изследва за възможни настъпили грешки. Това прави кода на програмата труден за разбиране и поддържане.
- Друг недостатък на разглеждания подход е, че в него няма стандарти. Това прави трудно еднотипното обработване на грешки.

# Генериране и обработка на изключения

- Механизмът за обработката на изключения в C++ предоставя стандартни, вградени в езика средства за реагиране на ненормални, грешни ситуации по време на изпълнение програмата.
- Механизмът на изключенията предоставя еднообразен синтаксис и стил за обработка на грешки в програмата.
- Елиминира нуждата за изрични проверки за грешки и съсредоточава кода за обработка на грешки в отделни части на програмата.

# Генериране на изключение

- При възникване на ненормална ситуация в програмата, програмистът сигнализира за настъпването ѝ чрез генерирането на изключение.
- Когато се генерира изключение нормалното изпълнение на програмата се прекратява докато изключението не бъде обработено.
- В C++ за генериране на изключение се използва ключовата дума **throw**.

# Генериране на изключение

```
1 class StackError { ... };
2 class Stack {
3 ...
4 public:
5 ...
6     int pop(void) {
7         if (top_ <= 0)
8             throw StackError;
9         return data_[--top_];
10    }
11 ...
12 }
```

# Обработване на изключение

- Най-често изключенията в програмата се генерират и обработват от различни функции.
- След като изключението бъде обработено изпълнението на програмата продължава нормално. Възстановяването на изпълнението на програмата обаче става не от точката на генериране на изключението, а от точката, където изключението е било обработено.
- В C++ обработката на изключенията се изпълнява в **catch**-секции.

```
1 catch (StackError ex) {  
2     log_error(ex);  
3     exit(1);  
4 }
```



# Обработване на изключение

- Всяка една **catch**-секция трябва да се асоциира с **try**-блок. В един **try**-блок се групират един или повече оператори, които могат да генерират изключения с една или повече **catch**-секции.

```
1 try {  
2     // Използване на обекти от класа Stack  
3     ...  
4 } catch (StackError ex) {  
5     // Обработка на грешка при използването на стека  
6     ...  
7 } catch (...) {  
8     // Обработка на всички останали грешки  
9     ...  
10 }
```

# Пример за използване на изключения

```
1 class StackError {};  
2 const int STACK_SIZE=10;  
3 class Stack {  
4     int data_[STACK_SIZE];  
5     int top_;  
6 public:  
7     Stack() {  
8         top_=0;  
9     }  
10    void push(int val) {  
11        if(top_>=STACK_SIZE)  
12            throw StackError();  
13        data_[top_++]=val;  
14    }
```

# Пример за използване на изключения

```
1  int pop(void) {
2      if (top_ <= 0)
3          throw StackError();
4      return data_[--top_];
5  }
6  bool is_empty() {
7      return top_ == 0;
8  }
9  bool is_full() {
10     return top_ == STACK_SIZE;
11 }
12 };
```

# Пример за използване на изключения

```
1 #include <cstdlib>
2 #include <iostream>
3 using namespace std;
4 int main(int arch, char* argv[]) {
5     char* msg="Hello_Cruel_World!";
6     char buff[10];
```

```
1  try {
2      Stack st;
3      for(char* p=msg;*p!='\0';p++)
4          st.push(*p);
5      char* p=buff;
6      while(!st.is_empty())
7          *p+=st.pop();
8      *p='\0';
9  } catch(StackError ex) {
10     cerr<<"StackError caught..."<<endl;
11     exit(1);
12 } catch(...) {
13     cerr<<"Unknown error caught..."<<endl;
14     exit(1);
15 }
16 return 0;
17 }
```