Конструктори. Деструктор.

Извикване на конструктори и деструктори. Конвертиращи конструктори. Указател this. Извикване на конструктори и деструктори при създаване масиви (статични и динамични).

д-р Тодор Цонков
todort@uni-sofia.bg

Какво е конструктор?

1) Специална член-функция на класа

2) Извиква се автоматично при създаване на обект

3) Има същото име като класа

4) Няма тип за връщане

5) Има за цел да инициализира състоянието му (данните му) и при нужда да задели ресурси.

```
class Student {
public:
    Student() { // конструктор
    }
};
```

1)Инициализира данните на обекта

2)Гарантира валидно начално състояние

3)Изпълнява логика при създаване

4) Конструктор без параметри се нарича дефолтен конструктор

Лоши примери за инициализация

Забравена инициализация:

```
class Point {
        public:
            int x, y;
            Point() {
                // нищо не правим
            }
};
```

Двустъпкова инициализация:

```
class Person {
    std::string name;
public:
    Person(std::string n) {
        name = n; // първо празен string, после
присвояване
    }
};
```

Неизползване на конструктор:

```
class Student {
        std::string name;
        double grade;
public:
    Student() { }
    void setGrade(double value){ grade =
value;}
};
int main(){
       Student s;
        s.setGrade(4.50)
        return 0;
}
```

1) name не е инициализирана
2) Създава се празен обект и после пълни с данни

Конструктор с параметри

Възможен, но неправилен вариант:

```
class Student {
   std::string name_;
   double grade_;
public:
   Student(const std::string& name, double grade) {
      name_ = name;
      grade_ = grade;
   }
};
```

//Две операции - първо се създават празен name и grade и после се инициализират!

Правилен вариант

Инициализиращ списък:

```
Student(const std::string& name, double
grade)
   : name_(name), grade_(grade) {}
```

Предимства:

1) По-бърз
2) Задължителен за const и референции
3) По-четим код
4) Спестява се невидимо копие

Конструктор - Инициализиращ списък

Задължително се използва при:

1) Константна променлива

```
class Student {
   const int id_;
public:
   Student(int id) : id_(id) {} // ЗАДЪЛЖИТЕЛНО
};
// НЕ може да бъде инициализирана в конструктора
```

2) Член данни от тип референция

```
class Wrapper {
   int& ref_;
public:
   Wrapper(int& x) : ref_(x) {} // ЗАДЪЛЖИТЕЛНО
};
```

3) Класове без дефолтен конструктор

```
class FacultyNumber {
   int value_;
public:
   FacultyNumber(int v) : value_(v) {}
};

class Student {
   FacultyNumber fn_;
public:
   Student(int fn) : fn_(fn) {} // ЗАДЪЛЖИТЕЛНО
};
```

Конструктор - допълнение

1) Ред на инициализацията

```
class Example {
   int x_;
   int y_;
public:
   Example() : y_(2), x_(1) {} // x_ се
инициализира първи!
};
```

Редът на инициализацията винаги отговаря на реда, в който са декларирани член данните на класа, а не реда, в който се извикват в конструктора. Това може да доведе до компилационни или логически грешки при неправилно използване

2) Конструктор по подразбиране

```
class Student {
public:
   Student() : name_(""),   grade_(2.0) {}
};

Student s;
```

Конструкторът по подразбиране е конструктор, който няма аргументи и е отговорен за първоначалната инициализация на обекта.

Забележка: При дефиниция на друг конструктор, default не се генерира автоматично и може да доведе!

Деструктор

1)Функция, която се извиква автоматично при унищожаване на обекта

2)Освобождава ресурси

3)Име: ~ClassName()

```
class Student {
public:
   ~Student() {
      // cleanup
   }
};
```

Вика се при следните ситуации:

1) Край на scope

```
{
   Student s("Ivan", 5.50);
} // ← тук се вика деструктор
```

2)delete или delete[] - по преценка на програмиста обектът може да бъде унищожен

3) Унищожаване на временни обекти

Деструктор - продължение

```
class Array {
   int* data_;
public:
   Array(int n) {
      data_ = new int[n];
   }

   ~Array() {
      delete[] data_;
   }
};
```

Важно е да е delete[] иначе - memory leak!

Лоши практики:

Деструкторът НЕ трябва да:

1)хвърля exceptions (това са специален тип грешки, за които ще учим по-нататък).

2)съдържа сложна логика - тъй като е възможно да не се изпълни цялата по различни причини

3)разчита на други вече унищожени обекти

4)да се извиква в кода - почти никога не трябва да се вика той, а при нужда delete/delete[]

Ред на извикване на деструкторите

```
class A { ~A(){ std::println("A"); } };
class B { ~B(){ std::println( "B"); } };

class C {
    A a_;
    B b_;
public:
    ~C(){  std::println("C"); }
};

//Изход: CBA
//Редът на създаване на обектите е: ABC!

//Редът на деструкторите е обратен на
реда на конструкторите
```

Обяснение:

Деструкторите се извикват в обратен ред на създаване, за да се гарантира, че зависимостите между обектите се разрушават безопасно.

Това следва принципа LIFO (Last In, First Out) — същия като при стек.
Нека първо се създава a, осле се създава b. Ако b използва a, тогава a трябва да остане жив, докато b съществува.

Много рядко има случаи на обекти, които да не могат да бъдат унищожавани.

Защо редът е такъв?

Обектите често зависят един от друг.

Първо се създават по-базовите части (членове, базови класове)

После по-специфичните

При унищожаване:

Първо се унищожава най-специфичната част

Накрая базовите части

Така гарантираме, че докато деструкторът на обекта се изпълнява, неговите членове все още съществуват.

Пример:

```
void foo() {
    A a;
    B b;
}
```

//Създава се a, b, унищожава се b, a.

Обратният ред гарантира:

✔ безопасност

✔ липса на достъп до вече унищожени обекти

✔ правилно освобождаване на ресурси

Undefined Behaviour

Undefined Behavior (неопределено поведение) означава:
Стандартът на C++ не дефинира какво трябва да се случи.

Компилаторът няма задължение:
1)да даде грешка
2)да даде предупреждение
3)да се държи предвидимо

Програмата може:
1)да работи „нормално“
2)да крашне
3)да дава грешни резултати

Пример:

```
class A {
public:
  ~A() { std::prinltn("Destroyed"); }
};


void foo() {
  A* a = new A();
  delete a;
  a->~A();   // ❌ UB
}
int* p = new int(5);
delete p;
std::print(*p);  // ❌ UB
```

Указател this

this е указател към текущия обект
Примерен Тип: Student*
*this е самият обект (Student&)

Защо ни трябва?

* Разграничаване на членове и параметри
this->member = member, иначе се получава member = member, което няма смисъл

* Връщане на текущия обект

* Method chaining - навръзване на методи
Примера вдясно: s.setName("Petar").print(); // chaining

Пример:

```
class Student {
private:
   std::string name;
public:
   Student(const std::string& n) : name(n) {}
   // Метод, който връща референция към текущия обект
   Student& setName(const std::string& n) {
      name = n;
      return *this;   // връщаме текущия обект
   }

   void print() const {
      std::prinltn(name);
   }
};
```

Грешки при използване на указател this

Връщане на временен обект:

```
Student& setGrade(double g) {
   Student tmp;
   tmp.grade_ = g;
   return tmp; // dangling reference
}

//tmp не е жив след излизането от scope
```

Връщане на this вместо на *this:

```
Student& setGrade(double g) {
   return this; // типова грешка
}
```

Пример:

```
class Student {
   double grade_;
public:
   Student& setGrade(double g) const {
      grade_ = g;   // не може – методът е const
      return *this;
   }
};
```

Проблем: const методът обещава, че няма да се променя обектът, а имаме достъп до него

Конвертиращи конструктори

Конструктор с един параметър, който позволява преобразуване от друг тип към типа на класа.

```
class Temperature {
   double celsius_;
public:
   Temperature(double c) : celsius_(c) {}
};

Temperature t = 36.6;
```

Пример за добра практика:

```
class Meters {
   double value_;
public:
   Meters(double m) : value_(m) {}
};

Meters distance = 5.0; // естествено
```

✔ По-кратък код
✔ По-изразителен интерфейс

Конвертиращи конструктори - лоши примери

```
class Student {
public:
    Student(int fn) {}
};

Student s = 12345; // неочаквано

void printStudent(Student s); // по копие не по референция

printStudent(12345); //12345 → създава се обект от класа Student

void process(Student s);
process(5); // Student(5) ❓
```

Когато имаме повече от 1 конструктор и не е ясно кой ще избере

```
class Number {
    double value_;
public:
    Number(int v) : value_(v) {}
    Number(double v) : value_(v) {}
};

void print(Number n) {}

int main() {
    print(5);    // ❌ ambiguous? може да избере int или double
}
```

Какво са шаблони за дизайн?

Дефиниция:
Дизайн патерн е повторяемо решение на често срещан проблем в софтуерния дизайн, не е готов код, а по-скоро рецепта или шаблон за това как да се организират класове и обекти, така че системата да е гъвкава, разширяема и поддържана.
Не е готов клас или библиотека, а концепция.

Builder Pattern
Проблем, който решава:
Когато имаме клас с много опционални параметри или сложна логика за създаване, конструкторът може да стане неудобен и объркващ.
Builder Pattern разделя конструкцията на обекта от неговото представяне, позволявайки стъпково и ясно конфигуриране.

Builder - пример

```
Student s = Student::Builder("Todor",
12345)
        .gpa(5.75)
        .major("CS")
        .build();

s.print();

Builder(std::string name, int fn)
: name_((name)), faculty_number_(fn)
{}
```

```
Builder& gpa(double gpa) {
    gpa_ = gpa;
    return *this;
}

Builder& major(const std::string& major)
{
    major_ = major;
    return *this;
}
Student build() {
    return Student(std::move(name_),
faculty_number_, gpa_, major_);
}
```

Експлицитни конструктори

Експлицитен конструктор е конструктор, който е маркиран с ключовата дума explicit. Той забранява неявни преобразувания.

```
class Temperature {
   double celsius_;
public:
   explicit Temperature(double c) : celsius_(c) {}
};

Temperature t1(36.6);   // OK
Temperature t2 = 36.6; // НЕ се компилира!
```

Добри практики

1) Да се използва explicit по подразбиране

2) Да се използва implicit само ако преобразуването е естествено и няма риск от объркване

3) Библиотеки като STL ползват винаги explicit

Масиви от обекти

Масив от обекти = поредица от инстанции на клас

За всеки елемент се извиква:
1)конструктор при създаване
2)деструктор при унищожаване

Ключови принципи

1)Конструкторите се извикват в реда на индексите
2)Деструкторите се извикват в обратен ред
3)Това важи както за статични, така и за динамични масиви

Пример

```
class A {
public:
    A()  { std::println( "A()"); }
    ~A() { std::println("~A()"); }
};

int main() {
    A arr[3];
}
```

Изход на изпълнението на програмата:

```
A()
A()
A()
~A()
~A()
~A()
```

Динамични масиви от обекти

Масив от обекти = поредица от инстанции на клас

Това е масив, чийто размер се определя по време на изпълнение, а паметта се заделя в heap-а (динамичната памет), не в стека.

Пример:

```
class C {
public:
    C()  { std::cout << "C()\n"; }
    ~C() { std::cout << "~C()\n"; }
};

int main() {
    C* arr = new C[3];
    delete[] arr;
}

//Отново: C(), C(), C(), ~C(), ~C(), ~C()
```

Чести грешки при използването.

1)C* arr = new C[3];
// memory leak + неизвикани деструктори заради забравен delete[]

2)delete arr; //Undefined behaviour - може да се изчисти правилно, но стандарта не гарантира нищо.

```
class D {
public:
    D(int);
};
```

3) D arr[5]; // компилационна грешка заради липса на дефолтен конструктор, който се създава експлицитно само, ако няма друг.

Пример с класа Student

```cpp
#include <iostream>
#include <string>

class Student {
    std::string name_;
    int age_;

public:
    // 1️⃣ Конструктор по подразбиране
    Student() : name_("Unknown"), age_(0) {
        std::cout << "Default constructor called\n";
    }

    // 2️⃣ Параметризиран конструктор
    Student(std::string name, int age)
        : name_(name), age_(age) {
        std::cout << "Parameterized constructor called\n";
    }

    // 3️⃣ Конвертиращ конструктор (string -> Student)
    Student(std::string name)
        : name_(name), age_(18) {
        std::cout << "Converting constructor called\n";
    }
```

Пример с класа Student

```
    // Метод, който използва this
    void setAge(int age) {
        this->age_ = age;  // разграничава член-променлива от параметъра
    }

    void print() const {
        std::cout << "Name: " << name_
                  << ", Age: " << age_ << "\n";
    }

    // Деструктор
    ~Student() {
        std::cout << "Destructor called for "
                  << name_ << "\n";
    }
};

int main() {

    std::cout << "---- s1 ----\n";
    Student s1;  // default constructor

    std::cout << "---- s2 ----\n";
    Student s2("Ivan", 21);  // parameterized constructor

    std::cout << "---- s3 ----\n";
    Student s3 = "Maria";  // конвертиращ конструктор
    std::cout << "---- end of main ----\n";
}
```

Въпроси?

Благодаря за вниманието!

Допълнителни материали: learncpp.com глава 14