Конструктори. Деструктор.

титуляр на курса: д-р Тодор Цонков (ttsonkov@gmail.com)

От какво ще се състои настоящата лекция?

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

Какво е конструктор в езика C++?

- Специална член-функция на класа
- Извиква се автоматично при създаване на обект
- Има същото име като класа
- Няма тип за връщане

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

Роля на конструктора

Какво прави?

1)Инициализира данните на обекта

2)Гарантира валидно начално състояние

3)Изпълнява логика при създаване

Лош пример

```
class Student {
public:
    std::string name;
    double grade;
};


Student s;
s.setGrade(5.50);
//name не е инициализирано
//Двустъпковото инициализиране също не е правилно!
```

По-добър пример

```
Student s(5.50, “Ivan”);

Student(const std::string& name, double grade) {
    if (grade < 2.0 || grade > 6.0)
        throw std::invalid_argument("Invalid grade");
    name_ = name;
    grade_ = grade;
}
```

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

Кога инициализиращия списък е задължителен?

Константна променлива

```
class Student {
    const int id_;
public:
    Student(int id) : id_(id) {} //
ЗАДЪЛЖИТЕЛНО
};
```

Класове без default constructor

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

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

Член-данни от тип reference

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

Ред на инициализациите

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

Конструктор по подразбиране

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

Student s;
```

Забележка: При дефиниция на друг конструктор, default не се генерира автоматично!

Деструктор

1)Извиква се автоматично при унищожаване на обекта

2)Освобождава ресурси

3)Име: ~ClassName()

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

Кога се извиква деструктора?

1) Край на scope

2) delete или delete[]

3) Унищожаване на временни обекти

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

Деструктори и динамична памет

```
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:

- хвърля exceptions
- съдържа сложна логика
- разчита на други вече унищожени обекти
- да се извиква в кода

Ред на унищожаване на обектите

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

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

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

```
class A {
public:
    ~A() = default; //стандартен
деструктор!
};


class B {
public:
    ~B() = delete;
};

//Обектът НЕ може да бъде унищожен -
редки случаи
```

Указател this

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

Защо ни трябва?

* Разграничаване на членове и параметри

* Връщане на текущия обект

* Method chaining - навръзване на методи

```
class Student {
    std::string name_;
    double grade_;
public:
    Student& setName(const std::string& name)
{       name_ = name;
        return *this;
    }

    Student& setGrade(double grade) {
        grade_ = grade;
        return *this;
    }
};

Можем да направим:
s.setName("Ivan")
 .setGrade(5.75)
```

Грешки с използването 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
}
```

```
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;
```

Кога са полезни?

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

Meters distance = 5.0; // естествено
```

✔ По-кратък код
✔ По-изразителен интерфейс

Лоши примери за конвертиращи конструктори

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

Student s = 12345; // неочаквано


void printStudent(Student s);

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

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

Кога да ги използваме?

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

Meters distance = 5.0; // естествено
```

✔ По-кратък код
✔ По-изразителен интерфейс

Explicit конструктори

```
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:
конструктор при създаване
деструктор при унищожаване

Ключови принципи

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

Пример

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

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

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

Динамични масиви от обекти

```
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

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

2)delete arr; //Undefined behaviour!


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

3) D arr[5];  // компилационна грешка
```

Обобщение на наученото пример с класа Student

```
class Student {
    int id;
    char name[32];

public:
    // 1) Нормален конструктор
    Student(int id = -1, const char* n = "unknown") : id(id) {
        strncpy(name, n, 31);
        name[31] = '\0';
        cout << "Student ctor  id=" << id << " name=" << name << endl;
    }

    // 2) Конвертиращ конструктор: int -> Student
    Student(int id_only) : id(id_only) {
        strncpy(name, "no-name", 31);
        name[31] = '\0';
        cout << "Converting ctor  id=" << id << endl;
    }

    // 3) Копиращ конструктор
    Student(const Student& other)
        : id(other.id)
    {
        strncpy(name, other.name, 31);
        name[31] = '\0';
        cout << "Copy ctor  id=" << id << " name=" << name << endl;
    }

    // 4) Деструктор
    ~Student() {
        cout << "Student dtor  id=" << id << " name=" << name << endl;
    }

    // 5) this – връщане на *текущия* обект (chain setters)
    Student& setName(const char* n) {
        strncpy(name, n, 31);
        name[31] = '\0';
        cout << "setName called for id=" << id << " (this=" << this << ")" << endl;
        return *this; // <-- използване на this
    }

    Student& setId(int newId) {
        this->id = newId; // <-- this е скрит параметър
        return *this;
    }

    void print() const {
        cout << "Student[id=" << id << ", name=" << name << "]" << endl;
    }
};
```

Въпроси?

Благодаря за вниманието!

Допълнителни материали: learncpp.com глава 14