Copy конструктор и оператор=. Rule of Three.

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

От какво ще се състои настоящата лекция?

RAII. Копиращ конструктор и оператор=. Rule Of Three.

Какво е животът на обекта?

Живот на обекта = времето от: създаването (инициализация, конструиране)
до унищожаването (деструкция, освобождаване на ресурси).

Включва:
1)време на съществуване в паметта
2)валидност на състоянието му
3)достъпност чрез променлива или указател

ВИНАГИ в езика C++ се гарантира извикването на конструктора и деструктора на обект!

Етапи

1)Алокация на памет -статична, автоматична (стек), динамична (heap)

2)Конструиране - извиква се конструктор

3)Използване - извикване на методи, промяна на състояние

4)Унищожаване - извиква се деструктор, освобождават се ресурси

RAII принцип

Фундамент в модерният C++ означаващ Resource Acquisition Is Initialization.

Какво представлява? Ресурсът се придобива в конструктора и се освобождава в деструктора.

Какви ресурси?

Динамична памет (new, malloc)
Файлове (FILE*, fstream)
Mutex-и, lock-ове
Сокети, дескриптори

Животът на ресурса = животът на обекта.

Лош пример без RAII

```
void f() {
    int* p = new int[10];
    if (error())
        return;   // memory leak!
    delete[] p;
}
```

Добър пример

```
void f() {
    std::vector<int> v(10);
    if (error())
        return;   // безопасно
}
```

RAII принцип

Защо RAII работи?

Деструкторът се извиква ВИНАГИ без значение дали е край на scope, exception или return

Пример за динамичен масив:

```
A* arr = new A[5];
// ...
delete[] arr;
```

Лесно се забравя delete[]

Няма exception safety

Трудно се поддържа - заради това в модерния C++ се използва std::vector

Пример за RAII

```
class File {
    FILE* f;
public:
    File(const char* name) {
        f = fopen(name, "r");
    }

    ~File() {
        if (f)
            fclose(f);
    }
};
```

Пример за RAII

```cpp
// ================== RAII клас ====================
// Управлява динамичен масив от Student
class StudentArrayRAII {
    Student* data;
    size_t size;

public:
    StudentArrayRAII(size_t n) : size(n) {
        cout << "RAII wrapper ctor\n";
        data = new Student[n]; // извикват се n дефолтни конструктора
    }

    Student& operator[](size_t i) { return data[i]; }

    ~StudentArrayRAII() {
        cout << "RAII wrapper dtor -> delete[] students\n";
        delete[] data; // извикват се n деструктора
    }
};

int main() {
    cout << "=== Единични обекти ===\n";
    Student a(1, "Ivan");
    Student b = 10;               // конвертиращ конструктор
    Student c = a;                // копиращ конструктор

    a.setName("Petar").setId(111); // демонстрация на this
    a.print();

    cout << "\n=== Статичен масив от обекти ===\n";
    Student group1[3] = {
        Student(2, "Maria"),
        Student(3, "Georgi"),
        Student(4, "Ana")
    }; // -> извикват се 3 конструктора

    cout << "\n=== Динамичен масив от обекти ===\n";
    Student* group2 = new Student[2]; // 2 дефолтни конструктора
    group2[0].setName("Todor").setId(7);
    group2[1].setName("Nikolai").setId(8);

    delete[] group2; // извикват се деструктората на 2-та студента

    cout << "\n=== RAII принцип ===\n";
    {
        StudentArrayRAII arr(2);  // ресурс се заделя
        arr[0].setName("RAII-1").setId(21);
        arr[1].setName("RAII-2").setId(22);
    } // <-- излизаме от блока: автоматично delete[] → без течове

    cout << "\n=== Край на main ===\n";
    return 0;
}
```

Какво е копиращия конструктор?

Използва се при:
подаване по стойност, връщане по стойност, инициализация

Два механизма:

1)копиращ конструктор
2)оператор за присвояване (=)

Правилен вариант

```
class Student {
public:
    Student(const Student& other);
};
```

Извиква се само при създаване на нов обект
const – задължително
приема референция, не стойност

Кога се извиква копиращия конструктор?

```
Student s1("Ivan");
Student s2 = s1;   // copy constructor


Student s3(s1);    // copy constructor


void print(Student s) {}
print(s1); // copy constructor
```

Return Value Optimization

```
Student create() {
    Student s("Maria");
    return s; // copy ctor (или move /
RVO)
}

// Очаква се да върне копие, но RVO ще
оптимизира
// Задължително от C++ 17!

Student arr[2] = { s1, s1 };

//отново при инициализация се създава!
```

RVO - особености

```
T make(bool flag) {
    if (flag) {
        T a;
        return a;   // ❌
    } else {
        T b;
        return b;   // ❌
    }
}
```

в този случай нямаме RVO понеже компилаторът не знае кое от двете условия ще е вярно и компилаторът ще създаде копие.

Проблем с shallow copy

Какво е плитко копие?

Плитко копие (shallow copy) е копиране на обект, при което:
се копират стойностите на членовете,
но ако обектът съдържа указатели (raw pointers), се копират само адресите, а не самите данни, към които сочат.

Лош пример

```
class Student {
   char* name;
public:
   Student(const char* n) {
      name = new char[strlen(n)+1];
      strcpy(name, n);
   }
Student(const Student& other) {
   name = other.name; // pointer
copy
}
};
```

Компилаторът генерира плитко копие

Два обекта → един и същ name

❌ double delete

Правилен пример

```
Student(const Student& other) {
   name = new
char[strlen(other.name)+1];
   strcpy(name, other.name);
}

//Всеки студент е със собствен
name
```

Какво е assignment operator=?

Student& operator=(const Student& other);
Използва се при вече съществуващи обекти

Трябва да:

1)освободи старите ресурси

2)копира новите

3)върне *this

Пример за operator=

```
Student& operator=(const Student& other)
{   if (this == &other) return *this;

    delete[] name;

    name = new char
      [strlen(other.name)+1];

    strcpy(name, other.name);

    return *this;
}
```

Rule of Three

Ако класът има:
деструктор
копиращ конструктор
оператор =

-> трябва да има и трите задължително

Това е преди C++ 11!

Допълнение

Rule of Three важи за класове, които притежават ресурс:
динамична памет (new/delete), файл, mutex, сокет

Този ресурс трябва да се:

1) освобождава

2) копира коректно

3) присвоява коректно

Трите операции покриват трите различни момента от живота на обекта.

Пример на проблемен код

```
class Bad {
    int* p;
public:
    Bad() { p = new int(5); }
    ~Bad() { delete p; }
};
```

Без да знаем компилатора генерира:

```
Bad(const Bad&); // shallow copy
Bad& operator=(const Bad&); // shallow
copy
```

Допълнение

Трябва да се мисли за Rule of Three

имплементация на контейнери

ниско-ниво системен код

wrap-ване C API-та

В модерния C++ почти винаги се заменя от:

std::vector

std::string

std::unique_ptr

Пример на проблемен код

```
class A {
public:
    A(const A& other) {
        data = new int[*other.size];
    }
    ~A() {
        delete[] data;
    }
};
```

a = b; все още е опасно
Имаме memory leak / double free

Допълнение

```
A& operator=(const A& other) {
    delete[] data;
    data = new int[other.size];
    return *this;
}
```

Ако a = a; → delete + use-after-free
Решение: if (this == &other) return *this;

```
A& operator=(const A& other) {
    data = new int[other.size]; // старото
data?
    return *this;
}
```

Старият ресурс е изгубен → memory leak

Пример за Rule of Three

```cpp
#include <cstring>
#include <iostream>

class Buffer {
    size_t size_;
    char* data_;

public:
    // 1️⃣ Constructor
    Buffer(size_t size = 0)
        : size_(size), data_(size ? new char[size] : nullptr)
    {
        std::cout << "Constructor\n";
    }

    // 2️⃣ Destructor
    ~Buffer() {
        delete[] data_;
        std::cout << "Destructor\n";
    }

    // 3️⃣ Copy constructor
    Buffer(const Buffer& other)
        : size_(other.size_), data_(other.size_ ? new char[other.size_] :
nullptr)
        std::memcpy(data_, other.data_, size_);
        std::cout << "Copy constructor\n";
    }

    // 4️⃣ Copy assignment operator
    Buffer& operator=(const Buffer& other) {
        std::cout << "Copy assignment\n";

        if (this == &other)
            return *this;

        delete[] data_;

        size_ = other.size_;
        data_ = other.size_ ? new char[other.size_] : nullptr;
        std::memcpy(data_, other.data_, size_);

        return *this;
    }
};
```

Въпроси?

Благодаря за вниманието!

Допълнителни материали: learncpp.com - глава 14, глава 21