Copy конструктор и оператор=. Rule of Three.

RAII и живот на обекта. Value Semantics. Копиращ конструктор и оператор=. Rule Of Three.

д-р Тодор Цонков
todort@uni-sofia.bg

Живот 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 от живота

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 и д.р

Животът на ресурса = животът на обекта.

Без RAII:

```
{   char* name = new char[6];
   std::strcpy(name, "Todor");
   std::print("Name: {}\n", name);
   // Забравяме да извикаме delete[]
   // delete[] name; <-- липсва
}
```

с RAII

```
{
   std::string name = "Todor";
   std::print("Name: {}\n", name);
   // Няма нужда да мислим за delete
} // name излиза от scope →
```

деструкторът автоматично освобождава паметта

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[]

Затова в C++ не се ползва динамичен масив, а има готово решение.

Без RAII:

Пример за RAII

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

Какво е копиращ конструктор?

Декларира се като:
ClassName(const ClassName& other);
Използва се за създаване на нов обект от съществуващ в следните ситуации:

подаване по стойност, връщане по стойност, инициализация

Реализира се чрез два механизма:
1)копиращ конструктор
2)оператор за присвояване (=)

```
struct Point {
  int x, y;
  Point(const Point& p) : x(p.x), y(p.y) {}
};
```

Извиква се само при създаване на нов обект
const – задължително
приема референция, не стойност, защото в противен случай ще се ще се създаде копие на обекта, за да се предаде на конструктора.

Копиращ конструктор - продължение

Но за да се направи това копие, компилаторът трябва да използва копиращия конструктор… който все още се извиква.
Това води до рекурсивно извикване на копиращия конструктор → безкрайна рекурсия → компилаторска грешка или runtime crash.
Копиращият конструктор не трябва да променя оригиналния обект. Ако не е const, компилаторът няма да може да приеме const обекти или временно създадени обекти като аргумент.

```
Student s1("Ivan");
Student s2 = s1;   // copy constructor
Student s3(s1);    // copy constructor

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

Какво е Shallow copy: копира само указатели → двоен delete може да причини crash

Deep copy: копира самите данни → безопасно управление на ресурси

Shallow copy

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.

```
Stack:                Heap:
-------               ---------------
b1.data  ----\
              ---->  [ int int int int int ]
b2.data  ----/
```

```
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

Deep Copy - решение

Решение:

```
struct Buffer {
  int* data;
  size_t size;
  Buffer(size_t n) : size(n), data(new int[n])
{}
  Buffer(const Buffer& other) :
size(other.size), data(new int[other.size]) {
      std::copy(other.data, other.data + size,
data);
  }
  ~Buffer() { delete[] data; }
};
```

По този начин си гарантираме, че всяко копие на класа ще има собствена памет и няма да води до проблеми при използването и.
Ако не трябва да се копира:
File(const File&) = delete;

Примери - когато ползваме уникални ресурси (пример Singleton). Когато копирането може да причини double delete или dangling pointer.

Return Value Optimization

RVO (Return Value Optimization) е оптимизация на компилатора в C++, която намалява ненужното копиране на обекти при връщане от функции.

```
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!
```

```
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.

Value Semantics

Value Semantics означава, че обектите се държат като стойности, подобно на int, double или bool.
Тоест:
1)Копирането създава независим обект

2)Промяна в копието не влияе на оригинала
3)Всеки обект има собствено състояние

Ако класът претендира да бъде value type, shallow copy е грешка.

Защо Value Semantics е важно?

✅ Предвидимост
При копиране на обект е ясно какво се случва.

✅ Безопасност
Няма споделени ресурси без контрол.

✅ Лесно за използване
Може да се мисли, че обектите се държат като числа.

✅ По-малко бъгове, защото няма скрити странични ефекти.

Какво е assignment оператор?

Синтаксис: ClassName& operator=(const ClassName& other);

Използва се при вече съществуващи обекти

Трябва да:
1)освободи старите ресурси
2)копира новите
3)върне *this

в Java става:
MyObject b = new MyObject(a); // copy constructor pattern

Ако не дефинираме operator=, компилаторът генерира:

ClassName& operator=(const ClassName&) = default;

1)Копира всички членове поотделно
2)Извършва shallow copy
3)Не знае нищо за ownership

Ако класът управлява ресурс → трябва да се дефинира operator= и да се използват RAII типове когато е възможно

Пример

```
Buffer& operator=(const Buffer& other) {
  if (this != &other) {
      delete[] data;

      size = other.size;
      data = new int[size];

      std::copy(other.data, other.data + size,
data);
  }
  return *this;
}
```

Осигурява:

- Освобождаване на стар ресурс
- Алокация на нов
- Копиране на съдържанието
- Self-assignment защита

Без Self-assignment защитата
delete[] data;
std::copy(other.data, ...); //
other.data вече е изтрит се стига до
→ Undefined Behavior

Сравнение с copy constructor

1. Основна разлика

Copy Constructor

Създава нов обект , инициализацира нов обект и се извиква при създаване на обект. По-лесен за имплементация

Assignment Operator: Работи със съществуващ обект, използва се при присвояване и се извиква се след създаване на обекта. По-труден за имплементация, понеже обектът вече съществува.

```
Point a(1,2);
Point b = a;     // copy constructor
Point c(a);      // copy constructor
Point a(1,2);
Point b(3,4);
b = a;   // assignment operator
```

Използване на памет:

b2 създава нова памет

b1.data ----> [...] //copy constructor

b2.data ----> [...]

b2 вече има ресурс //assignment op

първо delete стария

после копира новия

Rule of Three

Ако класът има една от :
деструктор, копиращ конструктор,
оператор =
-> трябва да има и трите
задължително
Това е преди C++ 11!

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.

Използва се при:
1)имплементация на контейнери
2)ниско-ниво системен код
3)wrap-ване C API-та

Пример

```
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
```

По този начин е нарушена инвариантната на класа, че всеки ресурс трябва да има точно един собственик.

Следователно трябва да имплементираме и copy constructor и copy assignment operator, които да бъдат имплементирани коректно и по този начин да осигурим правилно поведение на класа.

Цялостен пример

```cpp
#include <algorithm> // за std::copy
#include <cstddef>   // за size_t
#include <print>     // за std::print

class Buffer {
private:
    char* data_;
    size_t size_;

public:
    // Конструктор
    explicit Buffer(size_t size) : size_(size), data_(new char[size]) {
        std::print("Constructor called, size = {}\n", size_);
    }

    // Деструктор
    ~Buffer() {
        std::print("Destructor called\n");
        delete[] data_;
    }

    // Копиращ конструктор (Rule of Three)
    Buffer(const Buffer& other) : size_(other.size_), data_(new char[other.size_]) {
        std::print("Copy constructor called\n");
        std::copy(other.data_, other.data_ + size_, data_);
    }
```

Цялостен пример - продължение

```
    // Оператор за присвояване (Rule of Three)
    Buffer& operator=(const Buffer& other) {
        std::print("Copy assignment operator called\n");
        if (this == &other) // самоприсвояване
            return *this;

        delete[] data_; // Премахваме старите данни

        size_ = other.size_;
        data_ = new char[size_];
        std::copy(other.data_, other.data_ + size_, data_);

        return *this;
    }

    int main() {
      Buffer buf1(10);      // Конструктор
      buf1.set(0, 'A');
      buf1.set(1, 'B');

      Buffer buf2 = buf1;   // Копиращ конструктор
      buf2.set(1, 'C');

      Buffer buf3(5);
      buf3 = buf1;          // Copy assignment operator

      std::print("buf1[1] = {}\n", buf1.get(1)); // B
      std::print("buf2[1] = {}\n", buf2.get(1)); // C
      std::print("buf3[1] = {}\n", buf3.get(1)); // B

    return 0; // Деструкторите се извикват автоматично
}
```

Въпроси?

Благодаря за вниманието!

Допълнителни материали: learncpp.com глава 21 + chapter 14