Move семантики. Rule of Five.

Move семантики (C++ 11) - ползи, lvalue, rvalue, , std::move, move конструктор/оператор=. Rule of Five - примери.

д-р Тодор Цонков
todort@uni-sofia.bg

Move семантики

Move позволява прехвърляне на ресурс (pointer) от един обект в друг — почти мигновено.

Копирането на големи буфери е скъпа операция от гледна точка време и памет.

По този начин подобряваме производителността при контейнери и фабрични функции.

Намалява алокациите и освобождавания на памет.

```
SimpleString a("Very very long string ...");

SimpleString b = a; // copy ctor → заделя нов буфер и копира всички символи

SimpleString c = std::move(a); // move ctor → прехвърля само указателя
```

При копиране: нов new[] + memcpy.
При move: само присвояване на указател и зануляване на стария.

Move семантики

Move позволява прехвърляне на ресурс (pointer) от един обект в друг — почти мигновено.

Копирането на големи буфери е скъпа операция от гледна точка време и памет.

По този начин подобряваме производителността при контейнери и фабрични функции.

Намалява алокациите и освобождавания на памет.

```
SimpleString a("Very very long string ...");

SimpleString b = a; // copy ctor → заделя нов буфер и копира всички символи

SimpleString c = std::move(a); // move ctor → прехвърля само указателя
```

При копиране: нов new[] + memcpy.

При move: само присвояване на указател и зануляване на

lvalues and rvalues

lvalue: стойност с име/адрес — може да има длъжност да бъде променяна. Пример: x, obj.member.

rvalue: временна стойност, няма дългосрочен адрес. Пример: 42, x + y, std::string("tmp").

съществуват още xvalues, prvalues, glvalues

```
int x = 10; // x е lvalue, 10 - rvalue
int y = x; // x се използва като lvalue
int& ref = x; // lvalue reference
```

```
SimpleString s1("Hello"); // s1 е lvalue
SimpleString s2 = s1; // копиране (lvalue → copy ctor)

SimpleString s3 = SimpleString("Temp"); // временен обект → rvalue

int x = foo(); x - lvalue, foo() - rvalue

std::string s2 = s + " world";
s - lvalue, s + " world" - rvalue
```

Какво е rvalue reference?

rvalue референция със синтаксис T&& може да се свърже само с rvalue (временни обекти). Това е основният механизъм зад move семантиката и позволява да се вземат ресурсите на обектите.

```
int&& r = 5;     // OK
int x = 5;
int&& r2 = x;    // ❌ грешка
```

rvalue reference не означава временен живот — r има име и е lvalue вътре във функцията.

Опростено:
T& → стабилен обект
T&& → временен обект, на който може да вземем ресурсите.

```
void process(const std::string& s) {
   std::print("lvalue: {}\n", s);
}
void process(std::string&& s) {
   std::print("rvalue: {}\n", s);
}
std::string str = "hello";
process(str);   // lvalue
process(std::string("x")); // rvalue
process("test"s);   // rvalue
```

Какво прави std::move?

В C++ имаме value semantics – обектите по подразбиране се копират. Копирането на тежки ресурси (динамична памет, файлови дескриптори, сокети, mutex-и) е:
скъпо и понякога невъзможно.

От C++11 насам езикът въвежда move semantics – възможност за преместване на ресурса вместо копиране. Опростено прави променлива от тип T в тип T&&

```
std::string s = "hello";
std::string t = std::move(s);
```

std::move не мести нищо, а превръща lvalue в rvalue (по-точно xvalue), за да позволи извикване на move constructor или move assignment оператор.

Обектът остава във валидно, но неопределено състояние след move и е третиран като временен обект.

```
void foo(Buffer&& b) {
  Buffer x = std::move(b); // move
}
```

Използване на std::move

```
1)std::string createString() {
    std::string result = "expensive data";
    return std::move(result);// ГРЕШКА
} //RVO е задължително!
```

```
2) Опит за move от константен обект
    const std::vector<int> data = getData();
    consume(std::move(data));//copy, not
move
```

```
3)std::string name = "Alice";
std::string mName = std::move(name);
std::cout << mName << std::endl;  //
Undefined state!
```

```
void process(std::string s) {
   std::print("{}\n", s);
}
int main() {
   std::string str = "hello";
   process(std::move(str));
}
```

Сравнение по скорост на 10000 операции:

Deep Copy	7.82 ms
Move (Correct)	1.08 ms
Move from const	7.50 ms

Move конструктор

Move конструкторът е специален конструктор в C++ (въведен в C++11), който позволява прехвърляне на ресурси от временен обект (rvalue) към нов обект, вместо копиране.
ClassName(ClassName&& other);

При copy constructor: ClassName(const ClassName& other);
→ правим дълбоко копие на ресурса.
При move constructor:
ClassName(ClassName&& other);
→ отнемаме ресурса от other и го присвояваме на новия обект

Без заделяне на нова памет.
Без копиране. Само прехвърляне.

```
Buffer(Buffer&& other)
: size_(other.size_),
data_(other.data_) {

std::print("Move constructor\n");
// оставяме other в валидно,
но празно състояние
other.size_ = 0;
other.data_ = nullptr;
}
```

Move конструктор

Как работи?

Buffer a(10);
Buffer b = std::move(a);
std::move(a) превръща a в rvalue
Извиква се Buffer(Buffer&&)
b получава указателя
a остава с nullptr

Moved-from обектът трябва да е:
валиден, destructible, безопасен за присвояване
Но стойността му е неопределена логически.

Кога компилаторът генерира move constructor автоматично?
Ако няма user-defined copy constructor, user-defined destructor, user-defined copy assignment.
Тогава компилаторът може да генерира:

ClassName(ClassName&&) = default;

Но, ако имаме raw pointer почти сигурно се налага да имплементираме сами!

Move Assignment operator

Move assignment operator е специална член-функция на клас, която дефинира как се прехвърлят ресурсите на временен обект (rvalue) към вече съществуващ обект.

T& operator=(T&& other);

T&& е rvalue reference

other обикновено се модифицира (оставя се във валидно, но неопределено състояние)

функцията връща T&, за да поддържа chaining (a = b = c)

Move assignment е част от move semantics (C++11), въведени за:

1)елиминиране на излишни копия
2)прехвърляне на ownership
3)оптимизация на performance
4)ефективна работа с ресурси (heap, file handles, sockets и др.)

Без move assignment, всеки a = std::move(b) би извикал копиращ оператор, което често е скъпо.

Move Assignment operator

1)Използва се при присвояване от rvalue.

2)Освобождава текущите ресурси и ги взема от other.

3)Оставя other в валидно, но празно състояние.

4)noexcept е важно да се слага заради STL контейнери!

Пример:

```
Buffer& operator=(Buffer&& other) noexcept {
    if (this != &other) {
        // 1. Освобождаваме текущия ресурс
        delete[] data_;
        // 2. Прехвърляме ownership
        size_ = other.size_;
        data_ = other.data_;
        // 3. Нулираме source обекта
        other.size_ = 0;
        other.data_ = nullptr;
    }
    return *this;
}
```

Rule of Five

Ако даден клас дефинира или изисква потребителска дефиниция на един от петте специални член-функции, свързани с управление на ресурси и семантика на копиране/преместване, то най-често е необходимо изрично да се дефинират (или поне обмислят) и останалите четири.

Те са:
деструктор, copy/move конструктор, copy/move assignment operator

В C++ обектите могат да притежават ресурси:
динамично заделена памет (new/delete), файлови дескриптори
сокети, mutex-и, други OS или библиотечни ресурси

Компилаторно-генерираните версии на горните функции обикновено извършват повърхностно копиране (shallow copy), което води до:
двойно освобождаване на ресурс, memory leaks, dangling pointers, нарушени инварианти.

Rule of Five

Rule of Five не е синтактично изискване, а дизайнерски принцип, произтичащ от:

1)RAII (Resource Acquisition Is Initialization)

2)value semantics

3)ownership модели

Rule of Three (C++98): destructor + copy ctor + copy assignment

След въвеждането на rvalue references в C++11 се добавя move семантика, което разширява класическото Rule of Three до Rule of Five.

Управлението на ресурси в C++ изисква пълна и консистентна дефиниция на семантиката за унищожаване, копиране и преместване на обектите, когато класът притежава ресурс с динамичен или ексклузивен жизнен цикъл.

Rule of Zero: ако класът не управлява ресурс директно, не дефинира нито една от тези функции.

Цялостен пример

```cpp
#include <iostream>
#include <cstring>
#include <utility> // за std::move
#include <format>

struct MyString {
private:
    char* data_ = nullptr;
    std::size_t size_ = 0;

public:
    // 1. Конструктор
    MyString(const char* s = "") {
        if (s) {
            size_ = std::strlen(s);
            data_ = new char[size_ + 1];
            std::strcpy(data_, s);
        }
    }

      // 2. Деструктор
    ~MyString() {
        delete[] data_;
    }
```

Цялостен пример

```
    MyString(const MyString& other) {
        size_ = other.size_;
        data_ = new char[size_ + 1];
        std::strcpy(data_, other.data_);
        std::print("Copy ctor called\n");
    }

    // 4. Копиращ оператор за присвояване
    MyString& operator=(const MyString& other) {
        if (this == &other) return *this;

        char* new_data = new char[other.size_ + 1];
        std::strcpy(new_data, other.data_);
        delete[] data_;

        data_ = new_data;
        size_ = other.size_;
        std::print("Copy assignment called\n");
        return *this;
    }

    // Помощна функция за достъп до съдържанието
    const char* c_str() const { return data_; }
    std::size_t size() const { return size_; }
```

Цялостен пример

```
    // 5. Преместващ конструктор
    MyString(MyString&& other) noexcept
        : data_(other.data_), size_(other.size_)
    {
        other.data_ = nullptr;
        other.size_ = 0;
        std::print("Move ctor called\n");
    }

    // 6. Преместващ оператор за присвояване
    MyString& operator=(MyString&& other) noexcept {
        if (this == &other) return *this;

        delete[] data_;

        data_ = other.data_;
        size_ = other.size_;

        other.data_ = nullptr;
        other.size_ = 0;
        std::print("Move assignment called\n");
        return *this;
    }
```

Цялостен пример

```
int main() {
    MyString a("Hello");
    MyString b = a;            // копиращ конструктор
    MyString c;
    c = a;                     // копиращо присвояване

    MyString d = MyString("World"); // преместващ конструктор
    MyString e;
    e = MyString("C++");           // преместващо присвояване

    std::print("a: {}\n", a.c_str());
    std::print("b: {}\n", b.c_str());
    std::print("c: {}\n", c.c_str());
    std::print("d: {}\n", d.c_str());
    std::print("e: {}\n", e.c_str());
}
```

Въпроси?

Благодаря за вниманието!

Допълнителни материали: learncpp.com глава 22