Special member functions

Zdenko Pavlik

Scheidt&Bachmann Slovakia s.r.o.

Special member functions

... done with support of

Figure 1: scheidt&bachmann

Agenda

- Lessons learned from previous lesson (class introduction)
- Examples from previous lesson
- Special member function
- Compiler generated functions
- Copy semantics, Move semantics
- Rule of Zero / Three / Five
- Copy elision & RVO
- Practical examples

Lessons learned from previous lesson (class introduction)

Lessons learned from previous lesson (class introduction)

- Virtual destructor
  - Always declare destructor as `virtual`, otherwise it may not be called at all.
- Uninitialized variables
  - Always initialize your variables, at least with `{}`, otherwise those variables will obtain random values in release version.
- Encapsulation
  - Do not populate your variables as `public` to improve personal comfort. Other users can break functionality of your class.
- Use keywords like `const`, `override`, `final`
  - It will give a hint to other developers how to behave to your class.

Example of usage from previous lesson

Example of usage from previous lesson

- Inheritance (`Logger` example)
  - Reusing functionality
- Polymorphism (`std::vector<Animal> zoo` example)
- Polymorphism (`foo(Parent class)` example)

Example of usage from previous lesson

- Demo time

Special member function

Special member function

- Constructor (default or parametrized)
- Destructor
- **Copy** constructor
- **Copy** assignment operator
- **Move** constructor
- **Move** assignment operator

Compiler generated functions

Compiler will generate some functions for you, depending on your code. Even simple class will generate at least default constructor and destructor.

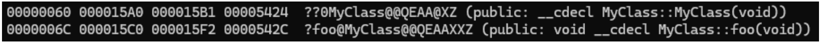
These functions (many times are correctly written) enables your class to be easily constructible, movable or copyable without developer's input.

But often (especially when class contains pointer as member variable) it needs to be handled by user!

Compiler generated functions

```
class MyClass
{
public:
    void foo()
    {
        std::cout << "foo" << std::endl;
    };

private:
    int i{};
};
```



```
00000060 000015A0 000015B1 00005424  ??0MyClass@@QEAA@XZ (public: __cdecl MyClass::MyClass(void))
0000006C 000015C0 000015F2 0000542C  ?foo@MyClass@@QEAAXXZ (public: void __cdecl MyClass::foo(void))
```

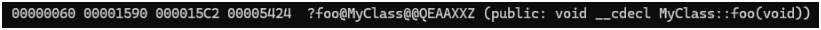
Figure 2: Compiler generated constructor

Compiler generated functions

[[DISCLAIMER]] This may not be true, even though I though different. :)

For very simple classes (without member variables) the code can be optimized that much, that even constructor is omitted.

```
class MyClass
{
public:
    void foo()
    {
        std::cout << "foo" << std::endl;
    };
};
```



```
00000060 00001590 000015C2 00005424  ?foo@MyClass@@QEAAXXZ (public: void __cdecl MyClass::foo(void))
```

Figure 3: Class without constructor

Copy semantics / Move semantics

Copy semantics

Copy semantics, also known as value semantics, is a principle in programming where an assignment or copy operation creates a new, independent, and equivalent object with its own copy of the original object's data, rather than sharing the underlying resource. This ensures that modifications to one object do not affect the other, preserving the independence of both objects. This is a common default in C++ for fundamental types but requires custom implementation for complex classes to avoid issues like shallow copying.

[TL;DR:] Ability to create **INDEPENDENT COPIES** of our instances.

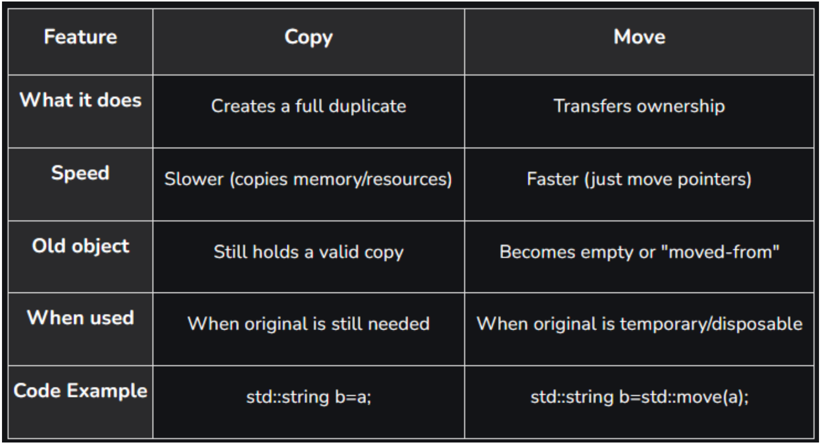
Copy semantics / Move semantics

Move semantics

Move semantics is a feature that allows our program to transfer ownership of resources (like memory, files, etc.) from one object to another instead of copying them. This results in faster performance, less memory usage, better efficiency, especially with big objects (like std::vector, std::string, or file streams).
[TL;DR:] Ability to create **REUSE RESOURCES** between instances.

Copy semantics / Move semantics

Comparison



Feature	Copy	Move
What it does	Creates a full duplicate	Transfers ownership
Speed	Slower (copies memory/resources)	Faster (just move pointers)
Old object	Still holds a valid copy	Becomes empty or "moved-from"
When used	When original is still needed	When original is temporary/disposable
Code Example	std::string b=a;	std::string b=std::move(a);

Figure 4: Move vs copy

Read more here.

Copy semantics / Move semantics

Interesting fact

Move semantics were introduced in C++11 standard. Since then we can use `std::move` function.

std::move

C++ move semantics from scratch

What is move semantics?

Rule of Zero / Three / Five

Constructor
Destructor
Copy constructor
Copy assignment operator
Move constructor
Move assignment operator

Copy elision & RVO

Practical examples - How to prevent class from being movable, copyable. Compare with std::unique_ptr