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 & Move elision
- 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. Potential memory leak.
- 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

```
std::string string1 = "Hello";
std::string string2 = string1;
std::string string3 = std::move(string1);

std::cout << "string1: " << string1 << std::endl; //""
std::cout << "string2: " << string2 << std::endl; //"Hello"
std::cout << "string3: " << string3 << std::endl; //"Hello"
```

Figure 2: Gizka

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 3: 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 4: Class without constructor

Compiler generated functions

You can instruct compiler to create/prevent default implementation.

= **default** enforces compiler to create default implementation of particular function

```
class MyClass
{
    MyClass() = default; //better use this on i.e. move constructor
};
```

= **delete** prevents compiler to generate particular function

```
class MyClass
{
    MyClass() = delete; //better use this on i.e. move 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

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?

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 5: Move vs copy

Read more here.

Rule of Zero / Three / Five

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

Rule of Zero / Three / Five

	Rule of Zero	Rule of Three	Rule of Five
Destructor	no	YES	YES
Copy constructor	no	YES	YES
Copy assignment operator	no	YES	YES
Move constructor	no	no	YES
Move assignment operator	no	no	YES

Rule of Zero / Three / Five

Best practices

C.20: If you can avoid defining any default operations, do

This rule is also known as “the rule of zero“. That means, that you can avoid writing any custom constructor, copy/move constructors, assignment operators, or destructors by using types that support the appropriate copy/move semantics. This applies to the regular types such as the built-in types bool or double, but also the containers of the Standard Template Library (STL)such as std::vector or std::string.

Cpp best practices, C.20
Modernes C++

Rule of Zero / Three / Five

Best practices

C.20: If you can avoid defining any default operations, do

```
class NamedVector
{
public: // Consider this section to be private
    std::string name;
    std::vector<int> data;
};

NamedVector nv;                    //Default constructor
nv.name = "My vector";
nv.data = {1, 2, 3, 4, 5};
NamedVector anotherVector{nv};     //Copy constructor
NamedVector anotherVector2;
anotherVector2 = nv;               //Copy assignment operator
```

Rule of Zero / Three / Five

Best practices

C.20: If you can avoid defining any default operations, do

Demo time

Rule of Zero / Three / Five

Best practices

C.21: If you define or =delete any default operation, define or =delete them all

The big six are closely related. Due to this relation, you have to **define** or **=delete** all six. Consequently, this rule is called “the rule of six“. Sometimes, you hear “the rule of five“, because the default constructor is special, and, therefore, sometimes excluded.

Cpp best practices, C.21
Modernes C++

Rule of Zero / Three / Five

Best practices

C.21: If you define or =delete any default operation, define or =delete them all

Demo time

Copy elision & Move elision

Sometimes it is beneficial to delete copy and move operations.

```
class NMvNCp //Not moveable, not copyable
{
public:
    NMvNCp(const std::string name) : m_name(name) {}
    ~NMvNCp() = default;

    NMvNCp(const NMvNCp& other) = delete;                // copy construct
    NMvNCp& operator=(const NMvNCp& other) = delete;     // copy assignmen
    NMvNCp(NMvNCp&& other) noexcept = delete;            // move construct
    NMvNCp& operator=(NMvNCp&& other) noexcept = delete; // move assignmen

private:
    std::string m_name{};
};
```

Copy elision & Move elision

```
NMvNCp bob{"Bob"};

NMvNCp joe(bob);
    // Compilation error, copy constructor is deleted
NMvNCp joe = std::move(bob);
    // Compilation error, move assignment is deleted
```

Comparison with std::unique_ptr

std::unique_ptr contains following member functions:

- `(constructor)`
- `(destructor)`
- `operator=` assigns the unique_ptr
  - move assignment operator

```
std::unique_ptr<int> number1 = std::make_unique<int>(42);
std::unique_ptr<int> number2 = std::make_unique<int>(15);
number1 = std::move(number2); //ok, enforces move
number1 = number2; //error, enforces copy
```

TL;DR: `std::unique_ptr` is movable, but not copyable.

Cpp reference
operator=

Practical examples

Shallow vs deep copy

Common problem is when class contains pointer as member variable. This is evaluated as fundamental data type in C++ and therefore just **value can be copied**, no the content. This results in double free in destructor.

Shallow copy happens when user accidentally copies just value of pointer - BAD.
Deep copy happens when user intentionally allocates new memory and copies content - GOOD

Practical examples

Shallow vs deep copy

Demo time, `PointerWrapper`

Lessons learned

- **C.20** If you can avoid defining any default operations, do
- **C.21** If you define or =delete any default operation, define or =delete them all
- Be careful with classes that contains pointer as member variable

Q&A