Today’s Outline

Topics
• Rule of Five Refresher
• Copy and Swap

Learning Objectives
• Learn how the C++ copy and swap idiom can improve your code

Assessments
• Quiz 09

Materials
• https://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom
Quiz Part 1
Rule of Five

• We are going to start by discussing the Rule of Five again
• It’s not that it is the most important topic in C++
• It is just a great way to teach a lot of C++ style and idiom
**Rule of Five Refresher**

If your class owns dynamic memory (it calls *new*) you must implement all of the following:

1. Destructor  
   (deallocated owned memory)
2. Copy Constructor  
   (copy from other)
3. Copy Assignment Operator  
   (deallocate own, copy from other)
4. Move Constructor  
   (steal from other)
5. Move Assignment Operator  
   (deallocate own, steal from other)

If your class does not own memory, then don’t implement any of them (known as the Rule of Zero)
class rule_of_five {
    char *cstring;

public:
    rule_of_five(const char *s = "") : cstring(nullptr) {
        if (s) {
            std::size_t n = std::strlen(s) + 1;
            cstring = new char[n];
            std::memcpy(cstring, s, n);
        }
    }
}
Destructor
~rule_of_five() { delete[] cstring; }

Copy Constructor
rule_of_five(const rule_of_five &other) : rule_of_five(other.cstring) {}

Copy Assignment Operator
rule_of_five &operator=(const rule_of_five &other) {
    return *this = rule_of_five(other);
}

Move Constructor
rule_of_five(rule_of_five &&other) noexcept :
    cstring(std::exchange(other.cstring, nullptr)) {}

Move Assignment Operator
rule_of_five &operator=(rule_of_five &&other) noexcept {
    std::swap(cstring, other.cstring);
    return *this;
}
};
Destructor

• You only need a destructor if you own memory

• Owning memory means that you call `new` somewhere in your class (mostly likely in one of your constructors) and your class is responsible for calling `delete` (or `delete[]`).

• If you don’t own memory then you don’t need a destructor
  • Unless the class is a base class, in which case you need to declare a virtual destructor
Destructor

```cpp
rule_of_five(const char *s = "") :
  cstring(nullptr) {
  if (s) {
    std::size_t n = std::strlen(s) + 1;
    cstring = new char[n];
    std::memcpy(cstring, s, n);
  }
}

~rule_of_five() { delete[] cstring; }

rule_of_five *rof1 = new rule_of_five("One");
```

Heap

Stack

nullptr

cstring

rule_of_five { cstring }

rof1
Destructor

```cpp
rule_of_five(const char *s = "") :
    cstring(nullptr) {
    if (s) {
        std::size_t n = std::strlen(s) + 1;
        cstring = new char[n];
        std::memcpy(cstring, s, n);
    }
}

~rule_of_five() { delete[] cstring; }
```

```
rule_of_five *rof1 = new rule_of_five("One");
```

Heap

```cpp
cstring
```

Stack

```cpp
rof1
```
Destructor

```cpp
rule_of_five (const char *s = ":") :
    cstring(nullptr) {
        if (s) {
            std::size_t n = std::strlen(s) + 1;
            cstring = new char[n];
            std::memcpy(cstring, s, n);
        }
    }

~rule_of_five() { delete[] cstring; }
```

```cpp
destructor rule_of_five *rof1 = new rule_of_five("One");
delete rof1;
```

Heap

```
one \0
```

Stack

```
rule_of_five { cstring }
```

rof1
Destructor

```cpp
rule_of_five(const char *s = "") :
cstring(nullptr) {
  if (s) {
    std::size_t n = std::strlen(s) + 1;
    cstring = new char[n];
    std::memcpy(cstring, s, n);
  }
}
~rule_of_five() { delete[] cstring; }
```

```cpp
rule_of_five rof1("One");
```
Incorrect
Copy Constructor

```cpp
correct

rule_of_five(const rule_of_five &other) {
    this->cstring = other.cstring;
}

~rule_of_five() { delete[] cstring; }
```

We are going to double delete!
AND
All changes to one will change both!
Correct Copy Constructor

```cpp
rule_of_five(const rule_of_five &other) : rule_of_five(other.cstring)
{
    // Constructor delegation
}
~rule_of_five() { delete[] cstring; }
```

```cpp
rule_of_five rof1("One");
rule_of_five rof2(rof1);
```
Incorrect
Copy Assignment

```
rule_of_five &operator=(
    const rule_of_five &other)
{
    this->cstring = other.cstring;
    return *this;
}

~rule_of_five() { delete[] cstring; }
```

We have a memory leak!
AND
We are going to double delete!
AND
All changes to one will change both!

```cpp
rule_of_five rof1("One");
rule_of_five rof2("Two");
rof1 = rof2;
```
Correct
Copy Assignment

```cpp
rule_of_five &operator=(
    const rule_of_five &other)
{
    return *this = rule_of_five(other);
}
~rule_of_five() { delete[] cstring; }
```

We’ll come back to it.
Move Constructor

```cpp
rule_of_five rof1("One");
rule_of_five rof2(std::move(rof1));

rule_of_five(rof1, other) noexcept:
  : cstring(std::exchange(other.cstring, nullptr))
{}

rof2 rule_of_five { cstring }
rof1 rule_of_five { cstring }
```
Move Assignment

```cpp
rule_of_five &operator=(rule_of_five &other) noexcept
{
    std::swap(cstring, other.cstring);
    return *this;
}
```

Temporary goes out of scope immediately after the move.

Destructor is called.

```cpp
rule_of_five rof1("One");
rof1 = rule_of_five("Two");
```
Move Assignment

```cpp
rule_of_five &operator=(rule_of_five &&other) noexcept
{
    std::swap(cstring, other.cstring);
    return *this;
}
```

Temporary goes out of scope immediately after the move.

Destructor is called.

```
rule_of_five rof1("One");
rof1 = rule_of_five("Two");
```
Correct
Copy Assignment

```
rule_of_five &operator=(
    const rule_of_five &other)
{
    return *this = rule_of_five(other);
}

~rule_of_five() { delete[] cstring; }
```

We’ll come back to it.
Correct
Copy Assignment

```cpp
rule_of_five &operator=(
    const rule_of_five &other)
{
    return *this = rule_of_five(other);
}

~rule_of_five() { delete[] cstring; }

rule_of_five rof1("One");
rule_of_five rof2("Two");
rof1 = rof2;
```

Temporary goes out of scope.
Destructor is called.
Correct Copy Assignment

```cpp
rule_of_five &operator=(
    const rule_of_five &other)
{
    return *this = rule_of_five(other);
}

~rule_of_five() { delete[] cstring; }
```

Temporary goes out of scope.

Destructor is called.

```cpp
rule_of_five rof1("One");
rule_of_five rof2("Two");
rof1 = rof2;
```
Copy and Swap Idiom

- Rule of Four and ½
  1. Destructor
  2. Copy Constructor
  3. Move Constructor
  4. Move/Copy Operator
  5. Friend Swap Function (helper)

- Write less code (eliminate code duplication)
- Let the compiler do more for you
- Strong exception guarantee
- Read this excellent post:
Copy and Swap Idiom

• Destructors stay the same
• Copy constructors stay the same
Start with a **swap** function

```cpp
friend void swap(rule_of_five &one, rule_of_five &two) {
    using std::swap;
    swap(one.cstring, two.cstring);
}
```

• Although we only have one member to swap in this example, it is still good practice to create this friend function.

• **It will be defined inside the class as a** friend
Updated Move Constructor

```cpp
rule_of_five(rule_of_five &&other) : rule_of_five()
{
    swap(*this, other);
}

rule_of_five(rule_of_five &&other) noexcept :
cstring(std::exchange(other.cstring, nullptr)) {}
```
Move/Copy Assignment Operator

`rule_of_five &operator=(rule_of_five other)`

```cpp
{ 
    swap(*this, other);
    return *this;
}
```

- The argument is passed by value (a copy is created)!
- We’re letting the compiler create the copy

You should try this out on previous labs and see when the different constructors and operators are called.
Quiz Part 2