Getting C++ Special Members Right

Or:

How to get your compiler to work for you without shooting yourself in the foot

Introduction

Things we need to get out of the way first

Key Terms

- *Function declaration* – tells the compiler what the function signature is
- *Function definition* – the actual code that will run when the function is called. A function definition is also a declaration!
- *Shallow copy* – copies pointers and references themselves
- *Deep copy* – copies pointed-to and referenced objects
- *Rule* – every rule mentioned here is not a rule of the language but rather a rule of thumb. Some are taken from the C++ Core Guidelines http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

History (pre c++ 11)

To prevent a function being called it could be:

- Declared but never defined
  - This would leave the error catching to the linker. We prefer catching errors early
  - If another translation unit defined such a function, no error would even be generated
  - Less of a problem for member functions
- Not declared
  - Error catching will happen at compiler (good!)
  - Not explicit
  - Might be accidentally added
  - Does not prevent implicit conversion

C++11 – Deleted Functions

A function can be prevented from being used by defining it as `= delete;`
This Can only be done at the first declaration.

- Error catching will happen at compile time
- Explicit
- Deleted functions participate in overload resolution

This does not only apply to special members!

```
void foo(bool);
void foo(int) = delete;
```

```
struct Base {
  int foo();
};

struct Derived : public Base {
  int foo() = delete;
};
```

What are special member functions?

Member functions that the compiler will automatically generate a declaration for if no user declaration exists and certain criteria are met.

- Default Constructor
- Copy Constructor
- Move Constructor
- Copy assignment
- Move Assignment
- Destructor

Default Special Member Functions

All Special Member Functions

- Will only be declared if no user declaration of the same function exists
- Will only be default defined if declared and if possible, otherwise will be deleted
- Can be forced to be defined using `= default;`
  This is user-declared and default-defined
- Can be forced to be deleted using `= delete;`

All Default Special Member Functions

- Will perform their operation on all base classes and members of the class, in initialization order
- Are public
- Are inline
- Are not virtual
- Are `noexcept` as appropriate
- Are `constexpr` if appropriate
- Are trivial if appropriate
- Does The Right Thing™ (usually)

```
struct Dog : public Animal {
  bool trained;
  int age;
};
```

Default Copy Functions

- Will perform a shallow copy
- Are implicit
- Will receive their parameter by `const&` if possible, otherwise by non-const reference.

User \ Default	Default constructor	Copy constructor	Copy operator=	Move constructor	Move operator=	Destructor
Nothing	YES	YES	YES	YES	YES	YES
Any constructor	NO	YES	YES	YES	YES	YES
Default constructor	NO	YES	YES	YES	YES	YES
Copy constructor	NO	NO	YES	NO	NO	YES
Copy operator=	YES	YES	NO	NO	NO	YES
Move constructor	NO	DELETED	DELETED	NO	NO	YES
Move operator=	YES	DELETED	DELETED	NO	NO	YES
Destructor	YES	YES	YES	NO	NO	NO

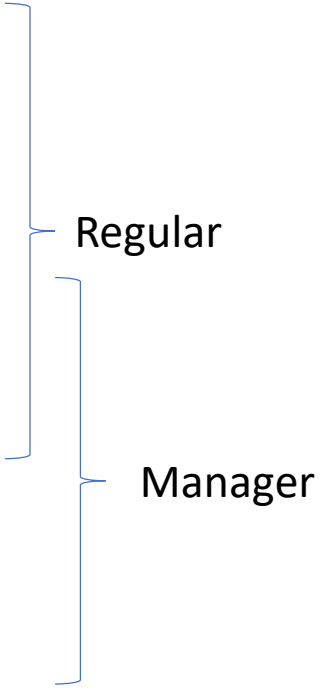
Source: https://mariusbancila.ro/blog/2018/07/26/cpp-special-member-function-rules/

User Defined Special Member Functions

Types of Types

Types can be very broadly categorized into 4 kinds:

- Non owning – ex. `std::span, std::complex`. Will probably need none of the special members to be user-declared.
- Owning – ex. `std::vector`. Will probably need some or all of the special members to be user-defined.
- Move-only – ex. `std::unique_ptr`. Will probably need all of the special members to be user-defined or deleted.
- Hierarchy – ex. `std::ios_base`. Special case (sort of)



```
struct Base {
  Base() = default;
  virtual ~Base() {}
  Base(Base const&) = default;
  Base& operator=(Base const&) = default;
};
```

```
struct Base {
  Base() = default;
  virtual ~Base() {}
  Base(Base const&) = delete;
  Base& operator=(Base const&) = delete;
};
```

Default Constructor

- Any constructor that can take zero parameters
- Should be `noexcept`
- Should be simple and cheap
- Should always be defined. `= default` is also a definition (and often a very good one)
- Should avoid initializing using static values. Use in-class initialization instead

(Old) Rule of Three

If a class needs to declare a destructor, copy constructor or copy assignment, it needs to declare all three.

```
class CyclicBuffer {
  std::size_t size = 0;
  std::size_t head = 0;
  std::size_t tail = 0;
  char* data = nullptr;
public:
  CyclicBuffer() = default;
  CyclicBuffer(int size)
    : size(size)
    , data(new char[size])
  {}
  ~CyclicBuffer() {
    delete[] data;
  }
```

```
  CyclicBuffer(
          CyclicBuffer const& other)
    : size(other.size)
    , head(other.head)
    , tail(other.tail)
    , data(new char[size])
  {
    std::copy(other.data,
              other.data + size,
              data);
  }
  .
  .
  .
};
```

```
CyclicBuffer& operator=(CyclicBuffer const& other) {
  if (this != &other) {
    size = other.size;
    head = other.head;
    tail = other.tail;
    delete[] data;
    data = nullptr;
    data = new char[size];
    std::copy(other.data, other.data + size, data);
  }
}
```

All done, right?

1. What will happen in the case of self assignment, e.g. `buffer1 = buffer1;` ?
2. What about exception safety?

```
CyclicBuffer& operator=(CyclicBuffer const& other) {
  if (this != &other) {
    size = other.size;
    head = other.head;
    tail = other.tail;
    delete[] data;
    data = nullptr;
    data = new char[size];
    std::copy(other.data, other.data + size, data);
  }
}
```

Now we are all done, right?

- We are still paying the price on every use for the rear case
- This is still only basic exception safety
- Most of the copying code is duplicated with the copy constructor

Copy and Swap (a.k.a Rule of 3.5)

Let’s add a non throwing swap method (this is recommended for most classes)

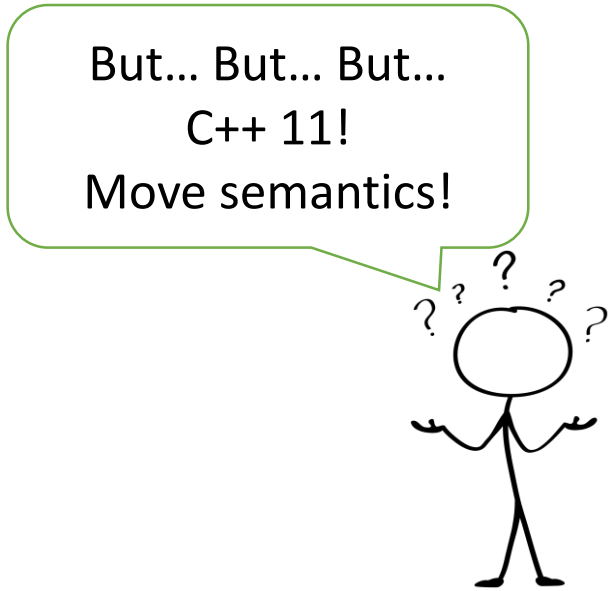
```
friend void swap(CyclicBuffer& a, CyclicBuffer& b) noexcept {
  using std::swap;
  swap(a.size, b.size);
  swap(a.head, b.head);
  swap(a.tail, b.tail);
  swap(a.data, b.data);
}
```

Copy and Swap (a.k.a Rule of 3.5)

Now we will implement the copy assignment using the following boilerplate:

```
CyclicBuffer& operator=(CyclicBuffer other) noexcept {
  swap(*this, other);
  return *this;
}
```

- Self assignment protection is free
- Strong exception safety
- No code duplication



Adding Move

All we need to do is add a boilerplate move constructor

```
CyclicBuffer(CyclicBuffer&& other) noexcept : CyclicBuffer() {
  swap(*this, other);
}
```

The assignment operator we wrote before will now be also a “fake” move assignment. Since it accepts it’s parameter by value, it will move-construct it if it can!

Move operations should always be `noexcept` !

```
class CyclicBuffer {
  std::size_t size = 0;
  std::size_t head = 0;
  std::size_t tail = 0;
  char* data = nullptr;
public:
  CyclicBuffer() = default;
  CyclicBuffer(int size)
    : size(size)
    , data(new char[size])
  {}
  ~CyclicBuffer() {
    delete[] data;
  }
  CyclicBuffer(CyclicBuffer const& other)
    : size(other.size)
    , head(other.head)
    , tail(other.tail)
    , data(new char[size])
  {
    std::copy(other.data, other.data + size, data);
  }
  CyclicBuffer(CyclicBuffer&& other) noexcept
    : CyclicBuffer()
  {
    swap(*this, other);
  }
  CyclicBuffer& operator=(CyclicBuffer other) noexcept {
    swap(*this, other);
    return *this;
  }
  friend void swap(CyclicBuffer& a, CyclicBuffer& b)
    noexcept
  {
    using std::swap;
    swap(a.size, b.size);
    swap(a.head, b.head);
    swap(a.tail, b.tail);
    swap(a.data, b.data);
  }
};
```

(New) Rule of 0/3/4 (+ 0.5)

	Destructor	Copy constructor	Copy assignment	Move constructor	Move assignment
No resource management (RTTI)	NO	NO	NO	NO	NO
No advantage to moving	YES	YES	YES	NO	NO
Non-copyable	YES	DELETED	DELETED	YES	YES
Copyable and movable	YES	YES	YES	YES	REDUNDANT

- Very easy to implement
- Very easy to get right
- Good-enough for 99% of cases – near optimal performance

Thank you