Special Members

OOP in C++

Special member: member functions that are, by default, implicitly defined as member functions (under specific circumstances). There are six:

1. **Default constructor**: `C::C();`
2. **Destructor**: `C::~C();`
3. **Copy constructor**: `C::C(const C&);`
4. **Copy assignment**: `C& operator=(const C&);`
5. **Move constructor**: `C::C(C&&);`
6. **Move assignment**: `C& operator=(C&&);`

1. Default Constructor

If a class is not explicitly defined with a constructor, the compiler assumes a default constructor. Therefore, the following class:

```
class Example{
public:
  int total;
  void accumulate(int x) {
    total += x;
  }
};
```

is assumed to have a default constructor. Therefore, we can initialize an object by declaring them with no parameters:

```
Example myObject;
```

However, once we explicitly define a constructor, the compiler will no longer provide the default constructor therefore:

```
class Example2 {
public:
  int total;
  Example2 (int initial) : total(initial) {}
  void accumulate(int x) {
    total += x;
  }
};
```

We must provide an argument when we instantiate the class.

```
Example2 myObj(100); //Okay: implicitly calls constructor
Example2 myObj = Example2(100); //Okay: explicitly call constructor
Example2 ex; //Not valid: no default constructor provided.
```

2. Destructor

Destructors do the opposite of constructors: they are called when an object is destroyed, or when it goes out of scope. Up until now, we have not had to allocate any memory when creating objects, therefore the need for a destructor was not necessary. However, imagine that the class dynamically allocates memory to store a string. In this case, it would be quite useful to have a function that is automatically called whenever the object goes out of scope, so that we can release its memory. We can create this by using a destructor. They are created in a similar fashion to constructors, where they have no return value, but they take no arguments and the class name is preceded with a tilde ( ).

```
#include <string>
#include <iostream>

class Example3 {
  std::string *ptr;
public:
  Example3(): ptr(new std::string) {} //default constructor
  Example3(const std::string& str) : ptr(new std::string(str)) {}

  //destructor:
  ~Example3() {delete ptr;}

  //access content(Read only)
  const std::string& content() const {return *ptr;}
};

int main() {
  {Example3 foo;}
  Example3 bar ("Example");
  std::cout << "bar: " << bar.content() << std::endl;
  return 0;
}
```

Output:

```
bar: Example
```

3. Copy Constructor

When an object is passed a **named** object of its own type as an argument, its copy constructor is invoked to copy the object. A copy constructor is a constructor whose first parameter is a const reference to the class itself (Why? A copy should never change the original). Thus a copy constructor must have the following signature:

```
MyClass::MyClass(const MyClass&)
```

If a class has no custom copy nor move constructors defined, an implicit copy constructor is provided. This copy performs a *shallow copy*. For example,

```
class MyClass {
public:
```

```
  int a, b;
  string c;
}
```

The implicit copy constructor would look something like:

```
MyClass::MyClass(const MyClass& x): a(x.a), b(x.b), c(x.c) {}
```

In most use cases, the default copy constructor is sufficient. However, shallow copies only copy the members of the class. Consider `Example3`. When we copy `Example3`, this would create a copy of only the pointer value, and not the content; thus, both pointers would point to the same value, and would share a single `string` object. At some point for destruction, both objects would attempt to delete the same value, and would (in most cases) cause a classic `segfault`. This can be solved by creating a custom copy constructor that performs a deep copy:

```
#include <iostream>
#include <string>

class Example4 {
  std::string *ptr;
public:
  //constructor
  Example4(const std::string& str) : ptr(new std::string(str)) {}
  //destructor
  ~Example4() {delete ptr;}
  //copy constructor:
  Example4(const Example4& other) : ptr(new std::string(other.content())) {}
  //access content:
  //NEEDS TO BE CONST METHOD!!!! a const method (the copy constructor) is
  ↪  invoking this method, thus it also needs to be const.
  const std::string& content() const {
    return *ptr;
  }
};

int main() {
  Example4 foo("Example");
  Example4 bar = foo;

  std::cout << "bar: " << bar.content() << std::endl;
  return 0;
}
```

Output:

```
bar: Example
```

4. Copy Assignment

Seen above, we are calling the copy constructor when declaring a new variable `bar`. However, copying can also be done through assignment.

```
MyClass foo;
MyClass bar(foo); //object declaration: copy constructor invoked
MyClass baz = foo; //object declaration: copy constructor invoked
foo = bar; //object assignment: copy assignment invoked
```

The copy assignment operator is an overload of `operator=` which takes a reference of the class as a parameter. The value is generally a reference to `this*`. For example, for `MyClass`, a copy assignment could have a signature that looks like:

```
MyClass& operator=(const MyClass&);
```

Implicitly, the copy assignment is defined as a shallow copy, much like the copy constructor. However, we can implement a deep copy in a similar fashion:

```
//inside Example4
Example4& operator=(const Example4& other) {
  *ptr = other.content(); //ptr is not const
  return *this;
}
```

5. Move Constructor and Assignment

Move semantics can be one of the most confusing parts of learning C++. After all, the definition for `std::move` on cppreference for many can cause even more confusion:

> `std::move` produces an xvalue expression that identifies its argument `t`. It is exactly equivalent to a `static_cast` to an rvalue reference type.
> Returns `static_cast<typename std::remove_reference<T>::type&&>(t)`

Doesn't really tell us that much. Simply put, moving an object involves "transferring" ownership of content from one object to another. The source loses it's content, which is taken over by the destination. This move can only occur iff the source is an unnamed object.

Unnamed object: Unnamed objects are objects that are temporary in nature, and thus don't have a name. Often times these are rvalues: values which don't have a spot in memory. Other examples may include return values of function or type-casts.

Using the value of a temporary object to initialize another object/assign it value does not require a copy; after all, there is no address to copy from. Instead, we move the value of the source into the destination. This invokes a different kind of method: move constructors and move assignments.

```
MyClass fn(); //function returning a MyClass object
MyClass foo; //default constructor
MyClass bar = foo; //copy constructor
MyClass baz = fn(); //move constructor
foo = bar; //copy assignment
baz = MyClass(); //move assignment; equal to baz = MyClass temp{};
```

The move constructor and assignment are members that take an rvalue reference of the class. Syntactically, this is done with a double ampersand.

```
MyClass (MyClass&&); //move constructor: takes a rvalue reference
```

```
MyClass& operator=(MyClass&&); move assignment
```

In classes that must dynamically allocate memory, move constructors and assignments are extremely important. We don't want to create a copy of the class if we are just passing a temporary instance of the class.

```
#include <iostream>
#include <string>

class Example5 {
  std::string *ptr;
public:
  Example5 (const std::string& str) : ptr(new std::string(str)) {}
  ~Example5 () {delete ptr;}

  //move constructor
  Example5 (Example5&& other) {
    this->ptr = other.ptr; //copies the pointer to this
    other.ptr = nullptr; //deassigns other pointer
  }

  //move assignment
  Example5& operator=(Example5&& other) {
    ptr = other.ptr;
    other.ptr = nullptr;
    return *this;
  }

  //access content
  const std::string& content() const {return *ptr;}
  //addition
  Example5 operator+(const Example5& rhs) {
    return Example5(content() + rhs.content());
  }
};

int main() {
  Example5 foo("Exam");
  Example5 bar = Example5("ple"); //move constructor
  foo = foo + bar; //move assignment
  std::cout << "foo: " << foo.content() << std::endl;
}
```

Output:

```
foo: Example
```

We can also invoke the move constructor using a pre-existing object, and transfer ownership using `std::move`.

```
Example5 foo("Example");
Example5 bar = std::move(foo); //move constructor invoked: foo is valid but
↪  unspecified, and bar contains "Example"
```

Maybe now the definition of `std::move` makes more sense. All it does is it simply takes

an object and casts it to an rvalue, and nothing else. Since we are then defining `bar` with an rvalue, it invokes the move constructor.