Special members

[NOTE: This chapter requires proper understanding of *dynamically allocated memory*]

Special member functions are member functions that are implicitly defined as member of classes under certain circumstances. There are six:

Member function	typical form for class C:
Default constructor	`C::C();`
Destructor	`C::~C();`
Copy constructor	`C::C (const C&);`
Copy assignment	`C& operator= (const C&);`
Move constructor	`C::C (C&&);`
Move assignment	`C& operator= (C&&);`

Let's examine each of these:

Default constructor

The *default constructor* is the constructor called when objects of a class are declared, but are not initialized with any arguments.

If a class definition has no constructors, the compiler assumes the class to have an implicitly defined *default constructor*. Therefore, after declaring a class like this:

```
1 class Example {
2   public:
3     int total;
4     void accumulate (int x) { total += x; }
5 };
```

The compiler assumes that `Example` has a *default constructor*. Therefore, objects of this class can be constructed by simply declaring them without any arguments:

```
Example ex;
```

But as soon as a class has some constructor taking any number of parameters explicitly declared, the compiler no longer provides an implicit default constructor, and no longer allows the declaration of new objects of that class without arguments. For example, the following class:

```
1 class Example2 {
2   public:
3     int total;
4     Example2 (int initial_value) : total(initial_value) { };
5     void accumulate (int x) { total += x; };
6 };
```

Here, we have declared a constructor with a parameter of type `int`. Therefore the following object declaration would be correct:

```
Example2 ex (100);   // ok: calls constructor
```

But the following:

```
Example2 ex;         // not valid: no default constructor
```

Would not be valid, since the class has been declared with an explicit constructor taking one argument and that replaces the implicit *default constructor* taking none.

Therefore, if objects of this class need to be constructed without arguments, the proper *default constructor* shall also be declared in the class. For example:

```
1  // classes and default constructors
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  class Example3 {
7      string data;
8    public:
9      Example3 (const string& str) : data(str) {}
10     Example3() {}
11     const string& content() const {return data;}
12 };
13
14 int main () {
15   Example3 foo;
16   Example3 bar ("Example");
17
```

```
bar's content: Example
```

```
  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

Here, `Example3` has a *default constructor* (i.e., a constructor without parameters) defined as an empty block:

```
Example3() {}
```

This allows objects of class `Example3` to be constructed without arguments (like `foo` was declared in this example). Normally, a default constructor like this is implicitly defined for all classes that have no other constructors and thus no explicit definition is required. But in this case, `Example3` has another constructor:

```
Example3 (const string& str);
```

And when any constructor is explicitly declared in a class, no implicit *default constructors* is automatically provided.

Destructor

Destructors fulfill the opposite functionality of *constructors*: They are responsible for the necessary cleanup needed by a class when its lifetime ends. The classes we have defined in previous chapters did not allocate any resource and thus did not really require any clean up.

But now, let's imagine that the class in the last example allocates dynamic memory to store the string it had as data member; in this case, it would be very useful to have a function called automatically at the end of the object's life in charge of releasing this memory. To do this, we use a *destructor*. A destructor is a member function very similar to a *default constructor*: it takes no arguments and returns nothing, not even `void`. It also uses the class name as its own name, but preceded with a tilde sign (~):

```
// destructors
#include <iostream>
#include <string>
using namespace std;

class Example4 {
    string* ptr;
  public:
    // constructors:
    Example4() : ptr(new string) {}
    Example4 (const string& str) : ptr(new string(str)) {}
    // destructor:
    ~Example4 () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};

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

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

```
bar's content: Example
```

On construction, `Example4` allocates storage for a `string`. Storage that is later released by the destructor.

The destructor for an object is called at the end of its lifetime; in the case of `foo` and `bar` this happens at the end of function `main`.

Copy constructor

When an object is passed a named object of its own type as argument, its *copy constructor* is invoked in order to construct a copy.

A *copy constructor* is a constructor whose first parameter is of type *reference to the class* itself (possibly `const` qualified) and which can be invoked with a single argument of this type. For example, for a class `MyClass`, the *copy constructor* may have the following signature:

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

If a class has no custom *copy* nor *move* constructors (or assignments) defined, an implicit *copy constructor* is provided. This copy constructor simply performs a copy of its own members. For example, for a class such as:

```
class MyClass {
  public:
    int a, b; string c;
};
```

An implicit *copy constructor* is automatically defined. The definition assumed for this function performs a *shallow copy*, roughly equivalent to:

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

This default *copy constructor* may suit the needs of many classes. But *shallow copies* only copy the members of the class themselves, and this is probably not what we expect for classes like class `Example4` we defined above, because it contains pointers of which it handles its storage. For that class, performing a *shallow copy* means that the pointer value is copied, but not the content itself; This means that both objects (the copy and the original) would be sharing a single `string` object (they would both be pointing to the same object), and at some point (on destruction) both objects would try to delete the same block of memory, probably causing the program to crash on runtime. This can be solved by defining the

following custom *copy constructor* that performs a *deep copy*:

```
// copy constructor: deep copy
#include <iostream>
#include <string>
using namespace std;

class Example5 {
    string* ptr;
  public:
    Example5 (const string& str) : ptr(new string(str)) {}
    ~Example5 () {delete ptr;}
    // copy constructor:
    Example5 (const Example5& x) : ptr(new string(x.content())) {}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example5 foo ("Example");
  Example5 bar = foo;

  cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

```
bar's content: Example
```

The *deep copy* performed by this *copy constructor* allocates storage for a new string, which is initialized to contain a copy of the original object. In this way, both objects (copy and original) have distinct copies of the content stored in different locations.

Copy assignment

Objects are not only copied on construction, when they are initialized: They can also be copied on any assignment operation. See the difference:

```
MyClass foo;
MyClass bar (foo);       // object initialization: copy constructor called
MyClass baz = foo;       // object initialization: copy constructor called
foo = bar;               // object already initialized: copy assignment called
```

Note that `baz` is initialized on construction using an *equal sign*, but this is not an assignment operation! (although it may look like one): The declaration of an object is not an assignment operation, it is just another of the syntaxes to call single-argument constructors.

The assignment on `foo` is an assignment operation. No object is being declared here, but an operation is being performed on an existing object; `foo`.

The *copy assignment operator* is an overload of `operator=` which takes a *value* or *reference* of the class itself as parameter. The return value is generally a reference to `*this` (although this is not required). For example, for a class `MyClass`, the *copy assignment* may have the following signature:

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

The *copy assignment operator* is also a *special function* and is also defined implicitly if a class has no custom *copy* nor *move* assignments (nor move constructor) defined.

But again, the *implicit* version performs a *shallow copy* which is suitable for many classes, but not for classes with pointers to objects they handle its storage, as is the case in `Example5`. In this case, not only the class incurs the risk of deleting the pointed object twice, but the assignment creates memory leaks by not deleting the object pointed by the object before the assignment. These issues could be solved with a *copy assignment* that deletes the previous object and performs a *deep copy*:

```
Example5& operator= (const Example5& x) {
  delete ptr;                     // delete currently pointed string
  ptr = new string (x.content());  // allocate space for new string, and copy
  return *this;
}

```

Or even better, since its `string` member is not constant, it could re-utilize the same `string` object:

```
Example5& operator= (const Example5& x) {
  *ptr = x.content();
  return *this;
}
```

Move constructor and assignment

Similar to copying, moving also uses the value of an object to set the value to another object. But, unlike copying, the content is actually transferred from one object (the source) to the other (the destination): the source loses that content, which is taken over by the destination. This moving only happens when the source of the value is an *unnamed object*.

Unnamed objects are objects that are temporary in nature, and thus haven't even been given a name. Typical examples of *unnamed objects* are return values of functions or type-casts.

Using the value of a temporary object such as these to initialize another object or to assign its value, does not really require a copy: the object is never going to be used for anything else, and thus, its value can be *moved into* the destination object. These cases trigger the *move constructor* and *move assignments*:

The *move constructor* is called when an object is initialized on construction using an unnamed temporary. Likewise, the

move assignment is called when an object is assigned the value of an unnamed temporary:

```
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
```

Both the value returned by `fn` and the value constructed with `MyClass` are unnamed temporaries. In these cases, there is no need to make a copy, because the unnamed object is very short-lived and can be acquired by the other object when this is a more efficient operation.

The move constructor and move assignment are members that take a parameter of type *rvalue reference to the class* itself:

```
MyClass (MyClass&&);            // move-constructor
MyClass& operator= (MyClass&&);  // move-assignment
```

An *rvalue reference* is specified by following the type with two ampersands (`&&`). As a parameter, an *rvalue reference* matches arguments of temporaries of this type.

The concept of moving is most useful for objects that manage the storage they use, such as objects that allocate storage with new and delete. In such objects, copying and moving are really different operations:
- Copying from A to B means that new memory is allocated to B and then the entire content of A is copied to this new memory allocated for B.
- Moving from A to B means that the memory already allocated to A is transferred to B without allocating any new storage. It involves simply copying the pointer.

For example:

```
// move constructor/assignment
#include <iostream>
#include <string>
using namespace std;

class Example6 {
    string* ptr;
  public:
    Example6 (const string& str) : ptr(new string(str)) {}
    ~Example6 () {delete ptr;}
    // move constructor
    Example6 (Example6&& x) : ptr(x.ptr) {x.ptr=nullptr;}
    // move assignment
    Example6& operator= (Example6&& x) {
      delete ptr;
      ptr = x.ptr;
      x.ptr=nullptr;
      return *this;
    }
    // access content:
    const string& content() const {return *ptr;}
    // addition:
    Example6 operator+(const Example6& rhs) {
      return Example6(content()+rhs.content());
    }
};


int main () {
  Example6 foo ("Exam");
  Example6 bar = Example6("ple");   // move-construction

  foo = foo + bar;                  // move-assignment

  cout << "foo's content: " << foo.content() << '\n';
  return 0;
}
```

```
foo's content: Example
```

Compilers already optimize many cases that formally require a move-construction call in what is known as *Return Value Optimization*. Most notably, when the value returned by a function is used to initialize an object. In these cases, the *move constructor* may actually never get called.

Note that even though *rvalue references* can be used for the type of any function parameter, it is seldom useful for uses other than the *move constructor*. Rvalue references are tricky, and unnecessary uses may be the source of errors quite difficult to track.

Implicit members

The six *special members functions* described above are members implicitly declared on classes under certain circumstances:

Member function	implicitly defined:	default definition:
Default constructor	if no other constructors	does nothing
Destructor	if no destructor	does nothing
Copy constructor	if no move constructor and no move assignment	copies all members
Copy assignment	if no move constructor and no move assignment	copies all members
Move constructor	if no destructor, no copy constructor and no copy nor move assignment	moves all members
Move assignment	if no destructor, no copy constructor and no copy nor move assignment	moves all members

Notice how not all *special member functions* are implicitly defined in the same cases. This is mostly due to backwards compatibility with C structures and earlier C++ versions, and in fact some include deprecated cases. Fortunately, each class can select explicitly which of these members exist with their default definition or which are deleted by using the keywords `default` and `delete`, respectively. The syntax is either one of:

```
function_declaration = default;
function_declaration = delete;
```

For example:

```
// default and delete implicit members
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle (int x, int y) : width(x), height(y) {}
    Rectangle() = default;
    Rectangle (const Rectangle& other) = delete;
    int area() {return width*height;}
};

int main () {
  Rectangle foo;
  Rectangle bar (10,20);

  cout << "bar's area: " << bar.area() << '\n';
  return 0;
}
```

```
bar's area: 200
```

Here, `Rectangle` can be constructed either with two `int` arguments or be *default-constructed* (with no arguments). It cannot however be *copy-constructed* from another `Rectangle` object, because this function has been deleted. Therefore, assuming the objects of the last example, the following statement would not be valid:

```
Rectangle baz (foo);
```

It could, however, be made explicitly valid by defining its copy constructor as:

```
Rectangle::Rectangle (const Rectangle& other) = default;
```

Which would be essentially equivalent to:

```
Rectangle::Rectangle (const Rectangle& other) : width(other.width), height(other.height) {}
```

Note that, the keyword `default` does not define a member function equal to the *default constructor* (i.e., where *default constructor* means constructor with no parameters), but equal to the constructor that would be implicitly defined if not deleted.

In general, and for future compatibility, classes that explicitly define one copy/move constructor or one copy/move assignment but not both, are encouraged to specify either `delete` or `default` on the other special member functions they don't explicitly define.

Previous: **Classes (II)** Index Next: **Friendship and inheritance**