Special members

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

注意:本章要求正确理解动态分配内存]

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

特殊成员函数是在特定情况下隐式定义为类成员的成员函数。有六个:

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.

默认构造函数是在声明类的对象时调用的构造函数，但不使用任何参数初始化。

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:

如果类定义没有构造函数，编译器就假定该类具有隐式定义的默认构造函数。因此，在像这样声明一个类之后:

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

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

编译器假设Example有一个默认构造函数。因此，这个类的对象可以通过简单地声明而不带任何参数来构造:

```
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:

但是，一旦类的某个构造函数显式声明了任意数量的形参，编译器就不再提供隐式默认构造函数，也不再允许声明没有实参的类的新对象。例如，下面的类:

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

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

```
Example2 ex (100);
```

But the following:

```
Example2 ex;
```

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.

将无效，因为类是用带有一个参数的显式构造函数声明的，该构造函数将替换不带参数的隐式默认构造函数。

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:

因此，如果需要在不带参数的情况下构造该类的对象，也应该在类中声明适当的默认构造函数。例如:

```
// classes and default constructors
#include <iostream>
```

```
#include <string>
using namespace std;

class Example3 {
    string data;
  public:
    Example3 (const string& str) : data(str) {}
    Example3() {}
    const string& content() const {return data;}
};

int main () {
  Example3 foo;
  Example3 bar ("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:

在这里，' Example3 '有一个定义为空块的*默认构造函数*(即无形参的构造函数):

```
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:

这允许在不带参数的情况下构造类Example3的对象(就像本例中声明的foo一样)。通常，像这样的默认构造函数是为所有没有其他构造函数的类隐式定义的，因此不需要显式定义。但在本例中，Example3有另一个构造函数:

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

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

当类中显式声明任何构造函数时，不会自动提供隐式默认构造函数。

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.

析构函数实现构造函数相反的功能:它们负责类生命周期结束时所需的必要清理。我们在前几章中定义的类没有分配任何资源，因此实际上不需要进行任何清理。

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 (`~`):

但是现在，让我们假设上一个例子中的类分配了动态内存来存储它作为数据成员的字符串;在这种情况下，让一个在对象生命周期结束时自动调用的函数来负责释放这些内存将非常有用。为此，我们使用析构函数。析构函数是一个成员函数，非常类似于默认构造函数:它不接受参数，也不返回任何东西，甚至不返回void。它还使用类名作为自己的名称，但在前面加一个波浪号(~):

```
// 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;
}
```

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

在构造时，Example4为字符串分配存储空间。稍后由析构函数释放的存储。

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`.

对象的析构函数在其生命周期结束时调用;在foo和bar的情况下，这发生在函数main的末尾。

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.

当一个对象被传递一个自己类型的命名对象作为参数时，它的*复制构造函数*被调用以构造一个副本。

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:

复制构造函数*是这样一种构造函数，它的第一个形参的类型是引用类*本身(可能是' const '限定的)，并且可以用这种类型的单个实参来调用它。例如，对于类' MyClass '，*复制构造函数*可能具有以下签名:

```
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:

如果类没有定义自定义复制或移动构造函数(或赋值)，则提供隐式复制构造函数。这个复制构造函数只是执行它自己成员的副本。例如，对于以下类:

```
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:

自动定义隐式复制构造函数。这个函数的定义执行了一个浅拷贝，大致相当于:

```
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*:

这个默认复制构造函数可以满足许多类的需要。但是浅复制只复制类本身的成员，这可能不是我们对上面定义的类Example4所期望的，因为它包含了它处理其存储的指针。对于这个类，执行浅复制意味着复制指针值，而不是内容本身;这意味着两个对象(复印件及原件)将共享单个字符串对象(他们都是指向同一个对象),并在某种程度上(毁灭)对象都将尝试删除同一块内存,可能会导致在运行时程序崩溃。这可以通过定义以下自定义复制构造函数来解决:

```
// 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;
}
```

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.

此复制构造函数执行的深层复制为新字符串分配存储空间，该字符串被初始化为包含原始对象的副本。这样，两个对象(复制和原始)都有存储在不同位置的内容的不同副本。

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:

对象不仅在构造时被复制，还可以在初始化时被复制:它们也可以在任何赋值操作时被复制。看到的区别:

```
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.

注意，baz在构造时是用等号初始化的，但这不是赋值操作!对象的声明不是赋值操作，它只是调用单参数构造函数的另一种语法。

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`.

对foo的赋值是一个赋值操作。这里没有声明对象，但是在一个现有对象上执行了一个操作;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:

复制赋值操作符是operator=的重载，它接受类本身的值或引用作为参数。返回值通常是*this的引用(尽管这不是必需的)。例如，对于MyClass类，复制赋值可能有以下签名:

```
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.

复制赋值操作符也是一个特殊函数，如果类没有定义自定义的复制或move赋值(也没有定义move构造函数)，它也会隐式定义。

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*:

但同样，隐式版本执行浅拷贝，这适用于许多类，但不适用于具有指针的类，这些指针指向处理其存储的对象，如Example5中的情况。在这种情况下，不仅类会招致两次删除所指向对象的风险，而且赋值操作由于没有在赋值之前删除对象所指向的对象而造成内存泄漏。这些问题可以通过复制赋值来解决，删除前一个对象并执行深度复制:

```
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:

或者更好，因为它的string成员不是常量，它可以重新使用相同的string对象:

```
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*.

与复制类似，移动也使用一个对象的值来设置另一个对象的值。但是，与复制不同的是，内容实际上是从一个对象(源)传输到另一个对象(目标):源丢失了内容，而内容被目标接管。只有当值的源是一个未命名的对象时才会发生这种移动。

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.

未命名对象本质上是临时的对象，因此甚至没有命名。未命名对象的典型例子是函数的返回值或类型转换。

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*:

使用这样的临时对象的值来初始化另一个对象或给它赋值，实际上并不需要复制:对象永远不会被用于其他任何事情，因此，它的值可以移动到目标对象中。这些情况会触发move构造函数和move赋值:

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:

当使用未命名的临时对象在构造时初始化对象时，将调用move构造函数。同样，当一个对象被赋给一个未命名的临时对象的值时，调用move赋值:

```
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.

fn返回的值和用MyClass构造的值都是未命名的临时值。在这些情况下，不需要进行复制，因为未命名对象的寿命非常短，而且当这是一种更有效的操作时，其他对象可以获取该对象。

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

move构造函数和move赋值函数都是接受引用类本身的右值型形参的成员:

```
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.

右值引用是通过在类型后面加上两个&&来指定的。作为形参，右值引用与这种类型的临时变量的实参相匹配。

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.

移动的概念对于管理它们使用的存储的对象最有用，比如使用new和delete分配存储的对象。在这样的对象中，复制和移动是完全不同的操作:
—从A复制到B是指将新的内存分配给B，然后将A的全部内容复制到分配给B的新内存中。
—从A移动到B意味着已经分配给A的内存被转移到B，而不分配任何新的存储。它只涉及复制指针。

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;
}
```

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.

在所谓的返回值优化中，编译器已经优化了许多正式需要move-construction调用的情况。最明显的是，当函数返回的值用于初始化对象时。在这些情况下，move构造函数实际上可能永远不会被调用。

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.

注意，尽管右值引用可以用于任何函数形参的类型，但它很少用于除move构造函数以外的其他用途。右值引用很棘手，不必要的使用可能是很难跟踪的错误来源。

Implicit members

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

上述六个特殊成员函数是在特定情况下在类中隐式声明的成员:

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:

注意，并非所有特殊成员函数都是在相同的情况下隐式定义的。这主要是由于与C结构和早期c++版本的向后兼容，事实上，有些版本还包括已弃用的情况。幸运的是，每个类都可以显式地选择哪些成员具有默认定义，或者分别使用关键字default和delete删除哪些成员。语法是:

```
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;
}
```

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:

在这里，' Rectangle '可以用两个' int '参数构造，也可以用*default-construct _(不带参数)。然而，它不能从另一个' Rectangle '对象复制构造_，因为这个函数已被删除。因此，假设最后一个例子中的对象，下面的语句将无效:

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

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

但是，可以通过将其复制构造函数定义为:

```
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.

注意，关键字' default '并没有定义一个等于*default构造函数*(即，*default构造函数*表示没有参数的构造函数)的成员函数，而是等于如果不删除将隐式定义的构造函数。

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.

一般来说，为了将来的兼容性，那些显式定义了一个复制/移动构造函数或一个复制/移动赋值函数但不是同时定义两个赋值函数的类，鼓励在它们没有显式定义的其他特殊成员函数上指定' delete '或' default '。