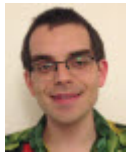
Understanding Initialization Lists in C++



By Alex Allain

Understanding the Start of an Object's Lifetime

In C++, whenever an object of a class is created, its constructor is called. But that's not all--its parent class constructor is called, as are the constructors for all objects that belong to the class. By default, the constructors invoked are the default ("no-argument") constructors. Moreover, all of these constructors are called before the class's own constructor is called.

For instance, take the following code:

```
#include <iostream>
class Foo
{
        public:
        Foo() { std::cout << "Foo's constructor" << std::endl; }
};
class Bar : public Foo
{
        public:
        Bar() { std::cout << "Bar's constructor" << std::endl; }
};

int main()
{
        // a lovely elephant ;)
        Bar bar;
}
```

The object bar is constructed in two stages: first, the Foo constructor is invoked and then the Bar constructor is invoked. The output of the above program will be to indicate that Foo's constructor is called first, followed by Bar's constructor.

Why do this? There are a few reasons. First, each class should need to initialize things that belong to it, not things that belong to other classes. So a child class should hand off the work of constructing the portion of it that belongs to the parent class. Second, the child class may depend on these fields when initializing its own fields; therefore, the constructor needs to be called before the child class's constructor runs. In addition, all of the objects that belong to the class should be initialized so that the constructor can use *them* if it needs to.

But what if you have a parent class that needs to take arguments to its constructor? This is where initialization lists come into play. An initialization list immediately follows the constructor's signature, separated by a colon:

```
class Foo : public parent_class
{
        Foo() : parent_class( "arg" ) // sample initialization list
```

```
        {
                // you must include a body, even if it's merely empty
        }
};
```

Note that to call a particular parent class constructor, you just need to use the name of the class (it's as though you're making a function call to the constructor).

For instance, in our above example, if Foo's constructor took an integer as an argument, we could do this:

```
#include <iostream>
class Foo
{
        public:
        Foo( int x )
        {
                std::cout << "Foo's constructor "
                          << "called with "
                          << x
                          << std::endl;
        }
};

class Bar : public Foo
{
        public:
        Bar() : Foo( 10 )  // construct the Foo part of Bar
        {
                std::cout << "Bar's constructor" << std::endl;
        }
};

int main()
{
        Bar stool;
}
```

Using Initialization Lists to Initialize Fields

In addition to letting you pick which constructor of the parent class gets called, the initialization list also lets you specify which constructor gets called for the objects that are fields of the class. For instance, if you have a string inside your class:

```
class Qux
{
        public:
                Qux() : _foo( "initialize foo to this!" ) { }
                // This is nearly equivalent to
                // Qux() { _foo = "initialize foo to this!"; }
                // but without the extra call to construct an empty string

        private:
```

```
        std::string _foo;
};
```

Here, the constructor is invoked by giving the name of the object to be constructed rather than the name of the class (as in the case of using initialization lists to call the parent class's constructor).

If you have multiple fields of a class, then the names of the objects being initialized should appear in the order they are declared in the class (and after any parent class constructor call):

```
class Baz
{
        public:
                Baz() : _foo( "initialize foo first" ), _bar( "then bar" ) { }

        private:
        std::string _foo;
        std::string _bar;
};
```

Initialization Lists and Scope Issues

If you have a field of your class that is the same name as the argument to your constructor, then the initialization list "does the right thing." For instance,

```
class Baz
{
        public:
                Baz( std::string foo ) : foo( foo ) { }
        private:
            std::string foo;
};
```

is roughly equivalent to

```
class Baz
{
        public:
                Baz( std::string foo )
                {
                    this->foo = foo;
                }
        private:
            std::string foo;
};
```

That is, the compiler knows which foo belongs to the object, and which foo belongs to the function.

Initialization Lists and Primitive Types

It turns out that initialization lists work to initialize both user-defined types (objects of classes) and primitive types (e.g., int). When the field is a primitive type, giving it an argument is equivalent to assignment. For instance,

```
class Quux
{
        public:
                Quux() : _my_int( 5 )  // sets _my_int to 5
                { }

        private:
                int _my_int;
};
```

This behavior allows you to specify templates where the templated type can be either a class or a primitive type (otherwise, you would have to have different ways of handling initializing fields of the templated type for the case of classes and objects).

```
template <class T>
class my_template
{
        public:
                // works as long as T has a copy constructor
                my_template( T bar ) : _bar( bar ) { }

        private:
                T _bar;
};
```

Initialization Lists and Const Fields

Using initialization lists to initialize fields is not always necessary (although it is probably more convenient than other approaches). But it is necessary for const fields. If you have a const field, then it can be initialized only once, so it must be initialized in the initialization list.

```
class const_field
{
        public:
                const_field() : _constant( 1 ) { }
                // this is an error: const_field() { _constant = 1; }

        private:
                const int _constant;
};
```

When Else do you Need Initialization Lists?

No Default Constructor

If you have a field that has no default constructor (or a parent class with no default constructor), you must specify which constructor you wish to use.

References

If you have a field that is a reference, you also must initialize it in the initialization list; since references are immutable they can be initialized only once.

Initialization Lists and Exceptions

Since constructors can throw exceptions, it's possible that you might want to be able to handle exceptions that are thrown by constructors invoked as part of the initialization list.

First, you should know that even if you catch the exception, it will get rethrown because it cannot be guaranteed that your object is in a valid state because one of its fields (or parts of its parent class) couldn't be initialized. That said, one reason you'd want to catch an exception here is that there's some kind of translation of error messages that needs to be done.

The syntax for catching an exception in an initialization list is somewhat awkward: the 'try' goes right before the colon, and the catch goes after the body of the function:

```
class Foo
{
        Foo() try : _str( "text of string" )
        {
        }
        catch ( ... )
        {
                std::cerr << "Couldn't create _str";
                // now, the exception is rethrown as if we'd written
                // "throw;" here
        }
};
```

Initialization Lists: Summary

Before the body of the constructor is run, all of the constructors for its parent class and then for its fields are invoked. By default, the no-argument constructors are invoked. Initialization lists allow you to choose which constructor is called and what arguments that constructor receives.

If you have a reference or a const field, or if one of the classes used does not have a default constructor, you must use an initialization list.

Previous: C++ Class Design
Next: Templates in C++
Back to C++ Tutorial Index

Related articles

Inheritance in C++
Constructors and Destructors in C++

C++ Class Design Tips