PASSING OBJECTS AS ARGUMENTS IN CPP

Passing Objects to Functions

Objects may be passed to functions in just the same way that any other type of variable can. Objects are passed to functions through the use of the standard call-byvalue mechanism. Although the passing of objects is straightforward, some rather unexpected events occur that relate to constructors and destructors.

To understand why, consider this short program.

```
// Passing an object to a function.
#include <iostream>
using namespace std;
class myclass {
int i; public:
myclass(int n);
~myclass();
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass::myclass(int n)
{
i = n;
cout << "Constructing " << i << "\n";
}
myclass::~myclass()
{
cout << "Destroying " << i << "\n";
}
```

```
void f(myclass ob);
int main()
{
myclass o(1);
f(o);
cout << "This is i in main: ";
cout << o.get_i() << "\n";
return 0;
}
void f(myclass ob)
{
ob.set_i(2)
cout << "This is local i: " << ob.get_i();
cout << "\n";
}
```

This program produces this output:

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

As the output shows, there is one call to the constructor, which occurs when **o** is created in **main( )**, but there are *two* calls to the destructor. Let's see why this is the case. When an object is passed to a function, a copy of that object is made (and this copy becomes the parameter in the function). This means that a new object comes into existence. When the function terminates, the copy of the argument (i.e., the parameter) is destroyed. This raises two fundamental questions: First, is the object's constructor called when the copy is made? Second, is the object's destructor called when the copy is destroyed? The answers may, at first, surprise you. When a copy of an argument is made during a function call, the normal constructor is *not* called. Instead, the object's *copy constructor* is called. A copy constructor defines how a copy of an object is made. As explained in Chapter 14, you can explicitly define a copy constructor

for a class that you create . However, if a class does not explicitly define a copy constructor, as is the case here, then C++ provides one by default.

The default copy constructor creates a bitwise (that is, identical) copy of the object. The reason a bitwise copy is made is easy to understand if you think about it. Since a normal constructor is used to initialize some aspect of an object, it must not be called to make a copy of an already existing object. Such a call would alter the contents of the object. When passing an object to a function, you want to use the current state of the object, not its initial state. However, when the function terminates and the copy of the object used as an argument is destroyed, the destructor *is* called. This is necessary because the object has gone out of scope. This is why the preceding program had two calls to the destructor. The first was when the parameter to **f( )** went out-of-scope. The second is when **o** inside **main( )** was destroyed when the program ended.

To summarize: When a copy of an object is created to be used as an argument to a function, the normal constructor is not called. Instead, the default copy constructor makes a bit-by-bit identical copy. However, when the copy is destroyed (usually by going out of scope when the function returns), the destructor is called. Because the default copy constructor creates an exact duplicate of the original, it can, at times, be a source of trouble. Even though objects are passed to functions by means of the normal call-by-value parameter passing mechanism which, in theory, protects and insulates the calling argument, it is still possible for a side effect to occur that may affect, or even damage, the object used as an argument. For example, if an object used as an argument allocates memory and frees that memory when it is destroyed, then its local copy inside the function will free the same memory when its destructor is called. This will leave the original object damaged and effectively useless. To prevent this type of problem you will need to define the copy operation by creating a copy constructor for the class, as explained

Source : http://elearningatria.files.wordpress.com/2013/10/cse-iii-object-oriented-programming-with-c-10cs36-notes.pdf