References Available Upon Request
Jesse Liberty

In my previous column,[1] I talked at length about pointers; this month the focus is on references. A reference is an alias to an object. While it is often *implemented* by the compiler using pointers, that fact is only a distraction, and you should not allow it to confuse you; references and pointers are quite different.

A pointer is a variable that holds the address of an object. A reference is an alias for an object. Pointers may be null references may *not* be null—you can't alias nothing.

References *must* be initialized. You create a reference by writing the type of the target object, followed by the reference operator (&), followed by the name of the reference. Thus, if you already have an integer variable named `someInt,` you can make a reference to that variable by writing:

```
int &someRef = someInt;
```

This is read as "`someRef` is a reference to integer and `someRef` is initialized to refer to `someInt,`" or more commonly, "`someRef` is a reference to `someInt.`"

If you ask a reference for its address, it returns the address of its target. That is the nature of references. They are aliases for the target. This can get you in trouble. Even experienced C++ programmers, who know the rule that references cannot be reassigned and are always aliases for their target, can be confused by what happens when you try to reassign a reference. What appears to be a successful reassignment turns out to be the assignment of a new value to the target. Listing 1 illustrates.

In Listing 1 a variable, `intOne,` is created and a reference, `someRef,` is initialized to `intOne.` The value 5 is assigned to `intOne,` and thus indirectly to `someRef.` These are printed. `intTwo` is created and initialized with the value 8, and then comes the important line:

```
someRef = intTwo;
```

It is tempting to believe that the result of this assignment will be that `intOne` will remain 5, `intTwo` will remain 8, and `someRef` will have the value 8. Unfortunately, that is not what will happen. `someRef` is and remains a reference to `intOne,` so writing

```
someRef = intTwo;
```

is *exactly* like writing

```
intOne = intTwo;
```

The result is that all three values are 8. On my computer, the results look like this:

```
intOne:  5
someRef: 5
&intOne: 0x0012FF7C
&someRef: 0x0012FF7C



intOne:  8
intTwo:  8
someRef: 8
```

```
   &intOne: 0x0012FF7C
   &intTwo: 0x0012FF74
   &someRef: 0x0012FF7C
```
What is critical here is to note that the addresses returned by `&intOne` and `&someRef` are the same—remember, `someRef` is an alias for `intOne` and when you ask it for its address you get the address of the object it refers to. It is as if `someRef` said, "Oh, I'm unimportant, sir. The one you want is the most esteemed `intOne`."

NULL REFERENCES When pointers are not initialized, or when they are deleted, they ought to be assigned to `null (0)`. This is not true for references. In fact, a reference *cannot* be null, and a program with a reference to a null object is considered an invalid program.

Of course, writing

```
someRef = 0;
```
does not create a null reference; it assigns the value 0 to whatever `someRef` refers to. You can force the creation of a null reference like this:

```
int * pointer;
pointer = 0; // create a null pointer
int & reference = *pointer; // Danger, Will Robinson!
```
Hey! Presto! A null reference, and thus, an invalid C++ program. When a program is invalid, just about anything can happen.

Most compilers will support a null object without much complaint, crashing only if you try to use the object in some way. Don't let this make you complacent when you move your program to another machine or compiler, mysterious bugs may develop, and your program is not right.

WHY BOTHER WITH REFERENCES? The value of references is that they simplify working with parameters to functions. When you want to pass by reference, pointers can be a bit cumbersome and references can be easier to work with.

In my previous column on pointers I illustrated the classic swap function. You'll remember the goal of this function was to swap two values, and I showed two attempts, first passing by value (Listing 2) and then passing by reference using pointers (Listing 3).

Listing 2 fails, as you may remember, because when you pass by value a copy is made and the original variables are unchanged. Passing in pointers fixes this problem, but at the cost of creating code that is cumbersome and unwieldy. Listing 4 illustrates passing by reference using references, and as you can see, this is much easier to read and to maintain.

Output from Listing 2:

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10
```

Output from Listing 3:

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, *px: 5 *py: 10
Swap. After swap, *px: 10 *py: 5
Main. After swap, x: 10 y: 5
```

Output from Listing 4:

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, rx: 5 ry: 10
Swap. After swap, rx: 10 ry: 5
Main. After swap, x: 10 y: 5
```

Passing By Reference for Efficiency Each time you pass an object into a function by value, a copy of the object is made; this takes time and uses memory. For small objects, such as the built-in integer values, this is a trivial cost; but for larger, user-defined objects these copies add up quickly.

The size of a user-created object on the stack is the sum of the sizes of each of its member variables. These, in turn, can each be user-created objects, and passing such a massive structure by copying it onto the stack can be expensive in performance and memory consumption.

There is another cost as well. When you pass an object by value the compiler must make a temporary copy of the object—it is the copy that is passed to the function (that is why changes in that function do not affect the original object). The creation of each temporary object is accomplished by a call to the appropriate copy constructor. When these temporary objects are no longer needed (when the function ends) they must be destroyed, which is accomplished by a call to the destructor. These constructor and destructor calls take time. Listing 5 illustrates both the problem and its solution.

Here is the output from Listing 5:

```
Making a cat...
Cat Constructor...
Frisky is 2 years old


Calling FunctionOne...
Cat Copy Constructor...
Function One...
Cat Destructor...
Frisky is 2 years old


Calling FunctionTwo...
Function Two...
Frisky is 8 years old


Calling FunctionThree...
Function Three...
Frisky is 16 years old


Exiting...
```

```
Cat Destructor...
```
As you can see, the call to `FunctionOne` (which takes a parameter by value) caused a temporary copy to be created and later destroyed. The copy was set to four-years old, but that had no effect on the original object. The calls to `FunctionTwo` and `FunctionThree` were more efficient. Because they pass by reference, no copy was made. Note that you can pass by reference using either a pointer or a reference.

Never Return a Reference to an Object with Local Scope No one sets out to create a reference to a null object, but sooner or later we all do. The classic case is a result of passing a reference to an object which then goes out of scope. Listing 6 illustrates how this might happen in one of the most common places to make this mistake: operator overloading.

You will remember that when overloading the increment operator (`++`) you must provide two forms: prefix and postfix. The semantics of the prefix operator (`++myObject`) are "increment, then fetch," while the semantics of the post fix operator (`myObject++`) are "fetch, then increment." That is, with the latter, you want back the unincremented version.

When implementing the prefix-increment operator, you can tell the object to increment itself and then return its *this* pointer—what you want is the incremented object itself. Of course, you'll want to return this by reference (for efficiency), but you'll return a constant version so that it can't be incremented again (that is, you want to prevent `++++myObject`).

The post fix operator is trickier. Here you need to return the object with its previous value, and so a temporary must be created. It is tempting (but wrong!) to return this temporary by reference, as illustrated in Listing 6. Here we create a simple `Counter` class and overload the increment operator.

Take a look at the implementation of the postfix operator and note that the local variable *temp* is created on the stack. Were this to be returned by reference, it would be out of scope and thus deleted by the time it was used in main. Despite the apparent inefficiency, this variable *must* be returned by value.

When to Use References and When to Use Pointers C++ programmers strongly prefer references to pointers. References are cleaner and easier to use, as you saw in the swap example. Consider the example using pointers. From `main()` you must pass in the address of the variables you want to swap, and then within `swap()` itself, you must deference the variables each time you use them. Now look at the example using references. The calling method just passes in the variables and the right thing happens. Further, within the `swap()` method there is no need to deference; because you are working with references you can work with the variables directly.

Remember that references cannot be reassigned, however. If you need to point first to one object and then another, you must use a pointer. And to say it just one more time, references can never be null, so if there is any chance that the object in question may be null, you must not use a reference. You must use a pointer.

CONCLUSION References are a powerful tool for simplifying the passing of parameters by reference. They complement pointers but they bring their own complications and risk. When you know the object can't be null and won't be reassigned, consider using

references, rather than pointers, to simplify the code and make it easier to read and to work with. References play a powerful role in operator overloading, and an upcoming column will explore this topic in detail.

You'll notice that we have renamed this column "Object-Oriented C++ from Scratch," and starting next month I'll begin to introduce object-oriented analysis and design into this discussion of C++. Along the way we'll explore the Unified Modeling Language (UML) and a number of related topics.

Reference

1. Liberty, J. "A Few Pointers," *C++ Report,* 10(10):57-62, Nov./Dec. 1998.

This article originally appeared in the
February 1999 issue of *Journal of C++ Report*.