Learning to Program with Haiku

Lesson 16

Written by DarkWyrm

What with all of the emphasis on getting the hang of making Haiku applications the last couple of lessons, you might be getting the impression that we're done learning C++, but there are some more aspects of the language still waiting to be learned. We will address them as they are needed while learning to write Haiku programs.

Function and Operator Overloading

Normally you think of overloading as being something you typically don't want to do, like overloading an electrical circuit. In C++, it's something that we do all the time to extend the usefulness of an existing function or language operator. **Function overloading** defines several versions of a function which take different sets of parameters. Operators are just functions, so things like plus and minus signs can be extended to better integrate the classes we define with the rest of the API. This gives us the ability to make our objects incredibly flexible and convenient.

In last lesson's project we saw our first example of operator overloading with the BString class when we generated the title for the window. It looked like this:

```
BString labelString("Clicks: ");

// This adds the value of fCount to the end of labelString. More
// on this next lesson.
labelString << fCount;
```

Normally the << operator does a binary shift left, but it works differently on a BString. The << operator has been overloaded to work as a more flexible shortcut for its `Append()` method, converting `fCount` to a string and then tacking it on to the end of `labelString`. Any operator in C++ can be overloaded, including array brackets, parentheses used for function calls, and the arrow operator.

Like many other aspects of C++, having the ability to do something does not mean that you should do it, however. Making the + operator perform subtraction can only lead to weeping and gnashing of teeth, for example. Except in rare cases, overloaded operators should perform pretty much the same operation as they do on other objects. In doing so, you will prevent confusion and headaches.

Except for a select few, operator functions may be implemented either as methods, i.e. part of a class, or regular functions. There are benefits and drawbacks to each. For example, here are the two ways that a + operator may be overloaded:

Method:

```
MyClass operator+(const MyClass &from);
```

Regular function:

```
MyClass operator+(const MyClass &first, const MyClass &second);
```

All other binary operators follow this format. As a general rule, make binary operators regular functions so that you can use the associated objects without having to explicitly typecast them. The only operators that must be methods are assignment ('='), function call ('()'), subscript ('[]'), and member selection ('→'). Other assignment operators, such as += and /=, should be methods, but are not required to be.

Unary operators follow this format for implementation:

Method:
```
bool operator!(void) const;
```
Regular function:
```
bool operator!(const MyClass &target);
```

There are also a couple of special case operators which deserve some attention. First of all, what about the ++ and -- operators? Each of them can be used two different ways. This means that there are two different ways to overload them. When overloaded, both the prefix (`++i`) and postfix (`i++`) versions must be implemented.

```
class MyClass:
{
	// Preincrement operator
	MyClass operator++(void);

	// Postincrement operator. The integer is just a dummy argument to
	// differentiate the two
	MyClass operator++(int dummy);
};
```

The subscript operator ('`[]`') also is a special case because it can be on either side of an assignment. It is required to be a method, so there is only one way to implement it:

```
MyClass & operator[] (const int index);
```

As a convenience to keep all of this hard-to-remember information, here is an almost-exhaustive table which organizes it into one place.

Operator	Method	Regular	Operator	Method	Regular
+		Recommended	=	Required	
-		Recommended	+=	Recommended	
*		Recommended	-=	Recommended	
/		Recommended	*=	Recommended	
%		Recommended	/=	Recommended	
++	Recommended		%=	Recommended	
--	Recommended		<		Recommended
[]	Required		<=		Recommended
^	Recommended		>		Recommended
~	Recommended		>=		Recommended
!	Recommended		==		Recommended
&&		Recommended	!=		Recommended

Operator	Method	Regular	Operator	Method	Regular
||		Recommended	Bitwise &		Recommended
<<		Recommended	|		Recommended
>>		Recommended	<<=	Recommended	
()	Required		>>=	Recommended	
[]	Required		&=	Recommended	
Reference &	Recommended		|=	Recommended	
Dereference *	Recommended		^=	Recommended	
->	Required				

Copy Constructors

Having learned about constructors and destructors in the last lesson, we've learned about most of the basics, leaving out one related item: the copy constructor, but we're going to address it along the way as we deal with a problem with floating point numbers.

The `float` type isn't terribly accurate. Add together 50 or 100 floating point numbers and you won't necessarily have exactly the result you would have if you were to add them up with a calculator. This stems from how they are stored in memory. Normally, this isn't a problem, but if we were going to write a personal finance program, any rounding error would be a major issue. We're going to create a type which is accurate to merely two decimal places and no errors. First, let's start with a basic class definition and a quick `main()` function to test it out.

```
#include <SupportDefs.h>
#include <stdio.h>

class Fixed
{
public:
            Fixed(void);
            ~Fixed(void);
      float GetValue(void);
      void  SetValue(const int64 &value);

private:
      int64 *fValue;

};


Fixed::Fixed(void)
{
      fValue = new int64();
      *fValue = 0;
}


Fixed::~Fixed(void)
{
      delete fValue;
```

```
}


float
Fixed::GetValue(void)
{
	return float(*fValue) / 100.0;
}
void
Fixed::SetValue(const int64 &value)
{
	*fValue = value * 100;
}


int
main(void)
{
	Fixed f;
	f.SetValue(1234);

	printf("Value: %f\n",f.GetValue());
	return 0;
}
```

The code itself isn't terribly complicated. We have four public methods: the constructor, which allocates heap memory for `fValue` and initializes the value to zero, the destructor, which frees the heap memory for `fValue`, and methods to get and set the object's value.

The idea behind our fixed class is that we're going to use a regular integer to hold a floating point value to avoid any rounding errors. The lowest two digits are reserved for the fractional part, so we will have to multiply any outside numbers by 100 to be able to add them to our Fixed class and divide the value of our Fixed class by 100 to translate to proper values for the outside world. So far everything seems to work properly. Let's tweak our `main()` function a bit:

```
int
main(void)
{
	Fixed f1;

	if (f1.GetValue() == 0)
	{
		Fixed f2;
		f2.SetValue(1234);
		f1 = f2;
	}
	printf("Value: %f\n",f1.GetValue());
	return 0;
}
```

If compiled and run under Haiku, once again everything seems to be OK, but there are two devilishly hard-to-find little bugs in the works that will pop up only if our program gets more complicated: the address held by `fValue` in our variable `f1` is deleted twice and there is memory leaked. Double deletes are in some ways worse than memory leaks because they can potentially crash your program and the location of the crash rarely has much to do with the location of the problem. Worse yet, changing the

code can cause it to crash in a different place without any apparent reason.

The problem with this code is that the last line of the `if` block doesn't do exactly what we want it to do. What we want is to copy the *value*, but what is copied is the *pointer*. When `f2` goes out of scope at the end of the `if` block, its `fValue` is deleted. The problem is that `f1`'s `fValue` is pointing to the same address. Uh-oh. `f1.fValue` is pointing to a memory location that is now invalid. Anything can happen when `f1` is deleted when our program exits. The memory allocated is also floating out in the ether, never to be deleted until the operating system cleans up our mess for us. If a problem like this should appear in a program under Ubuntu Linux, error information is printed out, but for Haiku, nothing happens. This is actually worse because we're not told about a problem which exists.

Fixing the bug, fortunately, is pretty easy once we understand it. The problem is that a shallow copy was performed. Shallow object copies dump the exact values of the properties of one object into another. This is fine so long as the objects' properties are allocated on the stack. Heap-allocated properties require a deep copy. This can only be done by implementing our own copy constructor and, while we're at it, overloading the assignment operator.

The copy constructor is a function called whenever an object needs to be duplicated. The default copy constructor performs a shallow copy, which is sufficient in many cases. This isn't one of them, however. We will add these two entries to the class definition:

```
            Fixed(const Fixed &from);
Fixed &     operator=(const Fixed &from);
```

The implementation of these two functions will look like this:

```
Fixed::Fixed(const Fixed &from)
{
	fValue = new int64();
	*fValue = *from.fValue;
}


Fixed &
Fixed::operator=(const Fixed &from)
{
	*fValue = *from.fValue;
}
```

This copies the values in the addresses kept in `fValue` instead of copying addresses themselves, thus performing a deep copy. All is right with the universe again for a little while longer. Whew!

Project

Let's go one step further and really flesh out this class. It may come in quite handy at some point in the future. Here are the declarations for some more operator overloading that would be useful to integrate our new class into the rest of the programming environment:

```
Fixed operator+(const Fixed &first, const Fixed &second);
Fixed operator-(const Fixed &first, const Fixed &second);
bool operator<(const Fixed &first, const Fixed &second);
bool operator>(const Fixed &first, const Fixed &second);
```

```
bool operator<=(const Fixed &first, const Fixed &second);
bool operator>=(const Fixed &first, const Fixed &second);
bool operator!=(const Fixed &first, const Fixed &second);
bool operator==(const Fixed &first, const Fixed &second);
```

Get some overloading practice by implementing and testing these regular functions. You might also want to see what other operators might come in handy.