Learning to Program with Haiku

Lesson 13

Written by DarkWyrm

In the last lesson, we learned about a programming paradigm that focuses on the design, construction, and interaction of objects to perform one or more tasks. We also examined how the objects in C++ are called classes, what their parts are, and some other fundamentals, but to be a proficient Haiku developer, we must learn more.

Inheritance

Much of the code a Haiku developer writes relates to **inheritance**, which is creating a class that uses another as its base. For example, let's say we have a Rectangle class that has height, width, and provides a couple of functions for area and perimeter. It would be easy to create a RectangularSolid class which just added a third dimension and a function for volume.

If the RectangularSolid class' third dimension is its only difference from a regular Rectangle, it seems like a lot of needless work to write code to calculate area and perimeter for both classes.. It would be great if we could just write code for only the parts that are different. Inheritance makes this possible: the RectangularSolid class is called a child class or a subclass of the Rectangle class. Just as a child inherits genes from his parents, a subclass inherits methods and properties from its parent class, and our RectangularSolid class inherits all of the Rectangle's properties and methods.

	Rectangle	**RectangleSolid**
Properties	Height Width	Height (inherited) Width (inherited) Depth
Methods	Area Perimeter	Area (inherited) Perimeter (inherited) Volume

Inheritance allows us to reuse code that we've already written. Code reuse is foundational to C++ programming. Once again, work smarter, not harder.

To turn this example into code, we will write two class definitions, one for each class.

```
class Rectangle
{
public:
                  Rectangle(int width, int height);

      void        SetWidth(int width);
      int         Width(void);

      void        SetHeight(int height);
      int         Height(void);

      int         Area(void);
      int         Perimeter(void);

private:
      int         fHeight;
      int         fWidth;
};
```

```
// This line is what makes RectangularSolid a subclass of Rectangle
class RectangularSolid : public Rectangle
{
public:
                  RectangularSolid(int width, int height, int depth);

      // We don't list the methods inherited from Rectangle – only the ones
      // which are new to the child class
      void        SetDepth(int depth);
      int         Depth(void);

      int         Volume(void);

private:
      int         fDepth;
};
```

The `RectangularSolid` class only declares methods and properties that are new. Writing the code that goes with these classes requires more than what the class definitions would make you think. This is our first code working with classes, so study this code and the comments carefully.

```
// Rectangle's constructor. This will set the object's properties to what were
// passed to it. When writing the code for a class' methods, the class name plus
// two colons must precede the method's name.
Rectangle::Rectangle(int width, int height)
{
      fWidth = width;
      fHeight = height;
}


void
Rectangle::SetWidth(int width)
{
      fWidth = width;
}


int
Rectangle::Width(void)
{
      return fWidth;
}


void
Rectangle::SetHeight(int height)
{
      fHeight = height;
}
```

```
int
Rectangle::Height(void)
{
	return fHeight;
}


int
Rectangle::Area(void)
{
	return fWidth * fHeight;
}


int
Rectangle::Perimeter(void)
{
	return (2 * fWidth) + (2 * fHeight);
}


// This is the RectangularSolid constructor. In creating the RectangularSolid, it
first creates a Rectangle object using the Rectangle's constructor function. We
pass width and height to it and then use depth to initialize fDepth
RectangularSolid::RectangularSolid(int width, int height, int depth)
	:	Rectangle(width, height)
{
	fDepth = depth;
}


void
RectangularSolid::SetDepth(int depth)
{
	fDepth = depth;
}


int
RectangularSolid::Depth(void)
{
	return fDepth;
}


int
RectangularSolid::Volume(void)
{
	// We have to call Width() instead of using fWidth and Height() instead of
	// fHeight because even child classes can't access an object's private
	// methods and properties.
	return Width() * Height() * fDepth;
}
```

The real difference between this code and all of the code we've written up to this point is that thinking in terms of objects forces us to organize our code. Writing neat and organized code is imperative to making maintaining it easier.

The only part of this code that might seem strange is the need to call `Width()` and `Height()` in `Volume()`. It seems like it would make more sense to allow child classes to be able to access `fWidth` and `fHeight`. That is what `protected` access is for. It's not needed or used nearly as much as you might think, though.

One other point to note is the line which handles the inheritance – it reads `public Rectangle`. This is the type of inheritance. Almost all of the time you will use public inheritance. This means that the access classes of the parent's methods and properties do not change. Choosing one of the other two types limits the access to the base class. Choosing `protected` inheritance will make all public methods and properties of the base class `protected` instead of `public`, making them available to all child classes, but closed to the outside. Private inheritance makes all methods and properties of the base class `private`, making the parent class totally inaccessible to both the outside world and to any "grandchild" classes.

Virtual Functions

Not only is it possible for a child class to add new methods and properties, but it can also change the behavior of existing methods. This is possible only when the base class says that doing so is allowed. Adding the `virtual` keyword before the return type in a method declaration grants this permission.

```
// A child class can redefine the behavior of this method
virtual void MyChangeableMethod(int someInt);

// A child class is required to define this method
virtual void ThisMethodMustBeDefined(float someFloat) = 0;
```

Being able to change how a method works does **not** enable us to change the number or types of a method's parameters or its return type, however. The original version found in the parent class doesn't disappear completely either. It can be specified using the scope operator, as exemplified below:

```
void
ChildClass::DoSomething(void)
{
	printf("Child class did something\n");
	ParentClass::DoSomething();
}
```

Static Functions

You usually have to have an instance of an object to call one of its methods. This can be a hassle sometimes, but declaring a method as `static` binds the method to the class, removing this requirement. Static methods are called using the scope operator the same way that we did for calling the parent implementation of a virtual function above.

```
class MyClass
{
public:
		MyClass(void);
	int	DoSomething(void);
	static int	DoSomethingStatic(void);
};
```

```
int
main(void)
{
	MyClass myClassInstance;

	// The regular way of calling a method
	myClassInstance.DoSomething();

	// An instance not required for this one – just a slightly different way to
	// call the method
	MyClass::DoSomethingStatic();
	return 0;
}
```

Overloading: Functions with the Same Name

One limitation of programming in C that can be a real pain in the neck is that no two functions can have the same name even if they have different parameters. C++ removes this limitation so long as the compiler is able to figure out which version is being called based on the number and types of arguments given to it. This will work:

```
int MyFunction(int oneWay);
int MyFunction(char *anotherWay);
int MyFunction(float aThirdWay);
```

This, on the other hand, won't work:

```
int SomeMethod(const char *oneConstString);
int SomeMethod(const char *anotherString, const char *optionalString = NULL);
```

Why doesn't this work? If you leave out the `optionalString` parameter, the compiler can't figure out which one you mean.

Assignment

Read through the sections of the BeBook on BApplication, BWindow, and BView. You don't have to understand everything, but try to get a feel for each class – in the next lesson we will be writing our first *real* program for Haiku.