Learning to Program with Haiku

Lesson 12

Written by DarkWyrm

C++ is the language in which most Haiku programs are written. It is an extension of the C language written by Dennis Ritchie at Bell labs in the 1970s. The language fundamentals that we have been learning are common to both languages, but from here on out we will be learning the parts of C++ that C cannot do. It is these parts that make it powerful, but to really grasp them well, we must change our perspective of writing code.

Object-Oriented Programming

The programs that we have been writing have been part of a paradigm called procedural programming, which is development that centers around individual function calls. This is fine for many purposes, but programming for the Haiku **G**raphical **U**ser **I**nterface (GUI) goes beyond this to another paradigm called object-oriented programming.

The idea behind object-oriented programming is that an application consists of the interaction of objects to perform a task. Each object is a "black box" which has certain properties and ways of interacting with it while hiding how it actually works. All we know about the object is how to interact with it. A real life example would be a television: pushing buttons changes the channel, adjusts settings in the on-screen menu, adjusts the volume, and turns it on or off. We also know its size, color, and approximate weight, and unless you're a TV repairman, you probably don't know how it works or have access to its internal components.

A C++ object is very much like a television. It has functions, called **methods,** which provide a means to interact with it. It also has variables for holding data which are called **properties**. Not all of an object's methods and properties are necessarily accessible by outsiders.

Classes

Type definitions in C++ are called **classes**. Classes are defined with code very much like the data structures we learned about in the last lesson. Here is an example class definition for our television.

```
class Television
{
public:     // All of these functions are accessible by anyone. Without the
            // 'public' keyword, all of these would be strictly internal methods –
            // the default access mode is private.

                  // This is the constructor function. More on this in a moment.
                  Television(void);

                  // This is the destructor function. More on this in a moment,
                  // too.
                  ~Television(void);

      void        SetChannel(uint16 channel);
      uint16      GetChannel(void);

      void        SetVolume(uint8 volume);
      uint8       GetVolume(void);

      void        TogglePower(void);
      bool        IsTurnedOn(void);
```

```
private:    // Everything below this word can't be touched by the outside world

      int8        DetectActiveInput(void);
      void        ConnectToSignal(void);

      uint16      fChannel;
      uint8       fVolume;
      bool        fPowerState;
};
```

The code bears a striking resemblance to a `struct` definition. There are a lot more methods than properties and a couple of odd words – `public` and `private`. The bits closely related to the operation of the television, such as `ConnectToSignal()`, are in the `private` section of the class definition and can only be called by the `Television` class' other methods.

It might seem just a little strange that there are six methods devoted to settings and getting the value of the Television's properties. Wouldn't it be easier – and less work – to just make `fChannel`, `fVolume`, and `fPowerState` public? Yes, but it would make the code that works with it messier. Our objects should expose their internal workings only when absolutely necessary. This makes it possible to change how it operates internally without disturbing its interaction with the outside world. For example, if we decided to rename `fPowerState` to `fIsTurnedOn`, the name of the variable would need to be changed in a handful of places in the code. All of those places would be in the file which contains the code for the class' methods. However, if we had made `fPowerState` a public property, we would have to change every reference to it in the entire project. If your project is large, this is a *lot* more work. Hiding properties using methods is called **data abstraction**, a term commonly heard in C++.

Calling an object's methods requires you to have access to the object. The dot and arrow operators that we used to get to a structure's variables are used with classes to access its methods and properties.

```
int
main(void)
{
      Television tv;

      // Watching a TV that is turned off isn't much fun
      if (!tv.IsTurnedOn())
            tv.TogglePower();

      // Watch the Channel 8 news :)
      tv.SetChannel(8);

      return 0;
}
```

The other oddballs in the class definition are the functions `Television()` and `~Television()`. They don't have a return type – not even `void`! This is because they are special methods. `Television` is called the class' **constructor**, which is a method called whenever we create a `Television` object. `~Television()` is the class' **destructor**, which is called whenever we destroy a `Television` object. They are optional, but almost every class has a constructor and many have a destructor. They also play a key role when allocating and freeing objects. Speaking of which, let's look at the C++ way of using heap memory. We're going to need it very soon.

C++ Memory Allocation: new and delete

To create a new `Television` object, we don't call `malloc()`. In fact, you will hardly ever use it unless you are writing a program in C. Instead, we will be using `new` and `delete`, the C++ counterparts to `malloc()` and `free()`. It's best we look at how they are used in this code example.

```
int
main(void)
{
	// Create a pointer to hold our TV and an object to go with it. Using new
	// allocates enough memory to hold one Television object and then calls
	// its constructor.
	Television *tv = new Television();

	// Like before, it works better when it is turned on
	if (!tv->IsTurnedOn())
		tv->TogglePower();

	// The news on 8 is all bad. Maybe Phineas and Ferb is on?
	tv->SetChannel(172);

	// Free our TV. Right before its memory is freed, the destructor method for
	// the Television class is called.
	delete tv;

	return 0;
}
```

It's much simpler to use `new` and `delete` for allocating memory on the heap. In fact, for objects, they are required. Other methods will cause an object's constructor and/or destructor to never be executed and lead to all kinds of ugly messes. `new` and `delete` can also be used for arrays of objects, as you can see in this code example.

```
int
main(void)
{
	// Create an array of TVs. I guess we're going to open an appliance store,
	// starting with 100 sets. We're going to need a little retail space. ;-)
	Television *tvArray = new Television[100];

	// Like before, it works better when it is turned on, but we'll turn just
	// one on, seeing how they're all the same. ;-)
	if (!tvArray[0].IsTurnedOn())
		tvArray[0].TogglePower();

	// We don't care what channel is on so long as we can't hear it.
	tvArray[0].SetVolume(0);

	// Free our TV. The brackets are needed when freeing an array we received
	// from new. Leaving out the brackets will cause a memory leak because
	// then only one of the objects will be freed instead of all of them.
	delete [] tvArray;

	return 0;
}
```

Construction and Destruction

The two special functions that we have seen in the definition for our `Television` class are part of the C++ language itself. The main job of a constructor function is to do whatever is necessary to initialize the object. When we allocate a `struct`, its variables contain random data. It's the same way with objects, but the constructor does the initialization for us.

A class has a default constructor if one is not declared in the class definition. The default constructor for an object takes no parameters and does nothing. Our `Television` class defines the default constructor, but this is not a requirement for a class. Doing so, however, replaces the one provided and does something. The constructor for a class can also take parameters, such as either of these possibilities or something else:

```
Television(const char *name);
Television(bool isHD);
```

The destructor for a class always looks the same: a tilde (~), the name of the class, and taking no parameters. It is provided if a destructor is not declared in a class' definition. Like the default constructor, it doesn't do anything. The main job for a destructor is to clean up before the object is actually freed. Most of the time, this means freeing heap memory that was allocated somewhere in the object's methods.

Assignment

Practice thinking about object-oriented programming by writing down what a sample class definition for the following objects might look like, including both methods and properties:

- Alarm clock
- Car
- Stove
- Washing machine