Lesson 7: Object-Oriented Programming

3D Game Programming With C++

Digital Media Academy (Summer 2011)

Written by: Andrew Uzilov (andrew.uzilov@gmail.com)
Feel free to contact me with any questions.

Object-oriented programming is at the heart of modern software development. But what is it, and why do people think it is so useful? And what is an object, anyway?

An **object in C++** is basically a bundle of data and functions for operating on that data. It ties together a bunch of different things in one convenient place. Functions that operate on an object are called **methods** and have a special syntax. Just like with other data, **we can save objects into variables**.

Let's say we have a bunch of information about a player in your game, like its:

- health
- speed
- score
- bullets remaining

We could package all these up into an object (a box of data in memory), like this:

```
int health          int getHealth()         void setHealth (int)
double speed        double getSpeed()       void setSpeed (double)
int score           int getScore()          void setScore (int)
int numBullets      int getNumBullets()     void setNumBullets (int)
```

The left column is the variables (and their types). The middle and right columns are methods for getting or changing the values of these variables inside the object.

An object is created using a recipe that is written in its **class**. Each object is created using a specific class, and that class is the object's type. So, just like we can say:

> variable `x` is of type `int`

we can also say:

> object `y` is of class `Player`

To **create a new object** representing our player, we can use syntax like this:

```
Player myPlayer;       // declare a variable named "myPlayer"
myPlayer = Player();   // define it
```

or, we could combine the above to say the same thing, but like this:

```
Player myPlayer = Player();
```

This is a very mysterious expression! What does it mean exactly? Let's color-code its parts and explain each.

```
Player myPlayer = Player();
```

- We are **declaring** and **defining** a new variable with the name `myPlayer`. Remember than in C++, every variable has a type. The type of this variable is `Player`. Just as before with `int` and `double` types, you have to put the type right before the variable name when you declare it.

- The variable `myPlayer` will hold our object. For variables that hold objects, the type of the variable is the same as the class from which it was created. In this case, the class is `Player`.

- `Player()` is a special function called a **constructor**. This function is defined in the `Player` class. We haven't shown you the code for how to create a class, but you will come across it later. The constructor literally creates (constructs) a new object of the type (class) `Player` and returns it. Some constructors can take arguments, but this one doesn't. A constructor always returns a new object that is of the class where the constructor is defined.

- So, this expression calls a constructor function `Player()` to create a new `Player` object, then copies that object into the variable `myPlayer`.

Whew!

Now, how do we use the object? Specifically, how do we **call methods** on it? We use dot notation.

We write the object variable's name, then a dot, then the method name (with parentheses after it, because the method is a special kind of function), like this:

```
myPlayer.getHealth();  // this evaluates to health value of type "int"
```

We can use the returned value in any expression we want. For example, here is code that will print some of the player's stats:

```
cout << "The player has " << myPlayer.getHealth() << " remaining." << endl;
cout << "The player scored " << myPlayer.getScore() << " points." << endl;
cout << "The player is moving this fast: " << myPlayer.getSpeed() << endl;
```

We can do arithmetic on the returned values, for example:

```
int newHealth = myPlayer.getHealth() - damage;
int newPosition = currentPosition + myPlayer.getSpeed();
```

And we can update the values inside the `myPlayer` object like this:

```
// changes object state, but doesn't return anything
myPlayer.setHealth (newHealth);
```

STRINGS ARE OBJECTS

Remember that we can create variables of type `string`? Guess what – it turns out that `string` variables are actually **objects**, meaning they have methods on them that can tell us something about the string. Here is an example:

```
string myName = "Andrew";
int myNameSize = myName.size();
cout << "My name has " << myNameSize << " letters in it." << endl;
```

Not every variable is an object. Variables of type `int`, `float`, `double`, `bool`, and `char` are not objects – they are just simple pieces of data that don't contain anything else. They do not have methods.

POINTERS

C++ has a special kind of variable whose job is to simply point to other places in memory. This variable type is called a **pointer**. A pointer is a variable that literally stores a memory address inside it. Just like everything else, pointers have a type – but for pointers, the type describes **the type of data to which the pointer is pointing**.

Confused? Don't worry, you are not alone. Pointers are powerful but complicated beasts, and it is not obvious at first why they're useful. For now, just keep in mind that many functions in code libraries (like Panda3D) will require as input, or return as output, **pointers to objects**, instead of actual objects, and the syntax for calling methods on them is different.

For now, just know that there are 2 important things to remember about pointers:

(1) Pointers to objects are declared and defined differently:

```
/* Declaring. */
Player myPlayer;      // variable stores a Player-type object
Player *myPlayerPtr;  // variable stores a POINTER to a Player-type object

/* Defining. */
myPlayer = Player();         // defining using the usual way
myPlayerPtr = new Player();  // defining using a pointer

/* Declaring and defining can be combined
   for pointers, just like for the usual way.
 */
Player *myPlayerPtr = new Player();
```

(2) If you want to call the method of an object to which you have a pointer, use an arrow (→), not a dot, like this:

```
// If your variable has the object in it, do this:
int health = myPlayer.getHealth();
// If your variable has a POINTER to the object in it, do this:
int health = myPlayerPtr->getHealth();
```

Optional Side Note: Why are pointers useful?

This is optional, but useful for the really curious.

Why would we ever want to use pointers to objects, instead of variables that store the objects themselves? Here are some reasons:

1) They are useful for efficiently passing objects to functions and returning them from functions. When you pass variables to a function (or return them from a function), they have to be copied from one place in memory to another. If you are passing a large object, that copying could really take a while. But a pointer is a small variable – it just holds a memory address pointing at something else. So passing a pointer is less to copy, hence faster and more efficient.

2) Passing a pointer to a function to modify the object to which it is pointing. If we pass an object to a function, that object is copied, and if the function changes the object, it changes its own copy, but our own copy remains unchanged. But if we pass a pointer, the function that we called can modify the original object (for example, by calling methods on it) because there is just one object – and several pointers to it.

3) Objects created with the `new` keyword live in a different part of memory than your program (let's call it "dynamically allocated memory"). This memory is used for storing things whose size or quantity you can't predict when the program begins. For example, if you are writing a multiplayer online game, you can't predict how many players will log on. So, if you want to create an object for each player in the game, you have to use the `new` keyword. Since dynamically allocated memory is dynamic (meaning the compiler can't figure out at compile time what the memory address of each object will be), we have to use pointers. The actual memory addresses to which they point are actually determined with help from the memory management parts of the operating system.