Constructors and Destructors

When we discussed the procedural programming extensions in C++, we talked about how we could initialize a variable at the point it was declared. This is easily done for built-in types, but up until now, we've had to declare objects and initialize them separately. This can cause hard-to-find errors due to uninitialized variables, and makes it difficult to initialize objects which are declared in static memory. Because C++ strives to support user-defined types as well as it supports built-in types, it is no surprise that C++ has a feature to enable the initialization of objects at the point of declaration.

Another problem exists with multiple initialization of objects. When we first discussed objects and classes, we saw the example of the `Person` class. We wanted a person's name to stay the same after the object was initialized. However, we needed to provide an initialization method in order to initialize the name. Unfortunately, the initialization method could be called at any time, thus allowing the name to be changed at some later point. To solve these problems, C++ provides mechanizsm for defining **constructors**—methods that are automatically called on behalf of the client whenever a new instance, be it statically or dynamically allocated, comes into scope. We'll aso talk a bit about what are called destructors. Not surprisingly, these are methods called on behalf of the client whenever a particular object goes out of scope.

Readings from Eckel:

Continute with the last reading assignment, but now focus on Chapter 6.

Readings from Deitel[2]:

Continute with the Chapter 6 and 7 assignment from last time, but go back to focus on Sections 6.10 through 6.14

Constructors

Simply put, constructors assure the initialization of an object. Suppose we had a class which represents a geometric rectangle. The class definition for this class appears below:

```
class Rectangle
{
   public:
      void init(double w, double h, const char* c);
      const char* getColor(void) const;
      void setColor(const char* c);
      double getWidth(void) const;
      void setWidth(double w);
      double getHeight(void) const;
      void setHeight(double h);
```

```
      double getArea(void) const;
      void draw(void) const;

   private:
      double width;
      double height;
      const char* color;
};
```

This class requires separate declaration and initialization. We increase our chance of having uninitialized variable errors by doing this. For example, consider this code:

```
Rectangle box;
cout << "The color of our box is " << box.getColor() << ".\n";
```

This would likely have disastrous effects, because the color accessor would return an uninitialized pointer from which we'd try to print. We can avoid this problem by providing a constructor. A constructor is declared within the class, just like any other member function. However, you don't specify any return type for a constructor [1], and it's name is always same name as the class. A constructor can take any number of parameters, and you can supply default arguments for them. You can use overloading to provide multiple constructors for a single class. In the Rectangle class, we'd want to remove the initialization method and replace it with a constructor as shown below:

```
class Rectangle
{
   public:
      Rectangle(double width, double height, const char* color);
      ...
};
```

We also need to provide the implementation for the constructor function. Again, we leave out the return type, and the name of the constructor is the same as the name of the class:

```
Rectangle::Rectangle(double w, double h, const char* c)
{
   width = w;
   height = h;
   char *buffer = new char[strlen(c) + 1];
   strcpy(buffer, c);
   color = buffer;
}
```

After providing a constructor which requires at least one argument, we can no longer simply declare a variable of type Rectangle on the stack like this:

```
Rectangle box; // With our new constructor, this is an error
```

Instead, we supply the constructor arguments after the declaration of the variable in parentheses:

```
Rectangle greenBox(2.0, 1.0, "Green");
```

If we wanted to dynamically allocate a Rectangle object, we specify the constructor arguments after
the class name within parentheses. When we do this, new will call the constructor with the arguments we provide:

```
Rectangle* blueBox = new Rectangle(10.0, 5.0, "Blue");
```

Note that even though we've provided a constructor for the Rectangle class, the "blueBox" variable is unitialized before we call new. This would seem to break the promise that "constructors assure initialization". However, declaring a pointer to an object is not the same as declaring an object directly. When declaring a pointer to an object, you **must** call new to allocate memory for the object and initialize it.

Default Constructors

You may recall that when we discussed new and delete, we saw how it was possible to combine the initialization of variables with memory allocation. For instance, when we had a pointer to a double which we wanted to initialize to the value of pi, we could do this:

```
double* pi = new double(3.14159);
```

Sure enough, this is the same way that we initialize user-defined types. You may also recall that we could **not** perform this initialization when we used the array form of new. This is also true when using user-defined types. That means that when we include a constructor which takes at least one parameter, we can't allocate an array of structures. This is true for any type of array declaration, be it static or dynamic. For instance, with the constructor we defined above, the following implementation would issue several compile-time errors:

```
Rectangle boxes[10];                          // Error
Rectangle* moreBoxes = new Rectangle[10];     // Error
```

In order to get around these restrictions, we need to have what's called a **default constructor**. A default constructor is a constructor which takes no arguments, or one that requires no arguments by providing a default parameter for each argument. We could define a default constructor for the Rectangle class by providing default parameters for the constructor:

```
class Rectangle
{
   public:
      Rectangle(double w = 0, double h = 0, const char* c = “Black”);
      ...
};
```

Now we can declare arrays of `Rectangle`s, and dynamically allocate them as well. We can also declare an object without providing any constructor arguments and it will be initialized using the default values. With our default constructor, if we wrote the following code:

```
Rectangle box;
cout << “The color of our box is ” << box.getColor() << “.\n”;
```

we would see this as the output:

```
The color of our box is Black.
```

Even if we didn’t intend for this box to be black, this output is still better than what we'd get had we not supplied a constructor.

You may be wondering what happens when we supply no constructor for a class. When we do this, the compiler provides a default constructor for us. This automatically generated default constructor simply calls the default constructor for each data member of the class. If each data member of the class does not have a default constructor, the compiler will issue an error.

Initialization Lists

Some data members need to be initialized at the time the object is created. For example, if a class contains a data member which is a reference, a constant, or an object without a default constructor, the compiler would flag an error when trying to create an instance of the object. For example, suppose we had a class to represent a `Point` which had no default constructor, and a class to represent a `Color`. If we defined a class called `Circle` we would have the following data members.

```
class Circle
{
   ...
   private:
      Color& color; // Note: the color is a reference
      Point center; // Note: Point has no default constructor
      const double radius; // Note: the radius is a constant
};
```

We would think our constructor for a `Circle` could look something like this:

```
Circle::Circle(Color& c, const Point& cent, const double r)
{
   color = c;        // Error: too late to initialize a reference
   center = cent;    // Error: no default constructor for a Point
   radius = r;       // Error: too late to initialize a constant
}
```

Unfortunately, even within a constructor, it is too late to initialize these three types of members. They must be initialized **before** the constructor gets called. C++ allows us to do this with **initialization lists**. We can provide an initialization list for the `Circle` class by altering the constructor like this:

```
Circle::Circle(Color& c, const Point& cent, const double r)
                                        : color(c), center(cent), radius(r)
{
   // empty body, since everything is taken care of
   // by the initialization list.  You can have stuff here, though..
}
```