Member Initializer Lists

Introduction

Normally, when you create a class, you'll initialize all of its instance variables inside the constructor. However, in some cases you'll need to initialize instance variables before the constructor begins running. Perhaps you'll have a `const` instance variable that you cannot assign a value, or maybe you have an object as an instance variable where you do not want to use the default constructor. For situations like these, C++ has a construct called the *member initializer list* that you can use to fine-tune the way your data members are set up. This handout discusses initializer list syntax, situations where initializer lists are appropriate, and some of the subtleties of initializer lists.

How C++ Constructs Objects

To fully understand why initializer lists exist in the first place, you'll need to understand the way that C++ creates and initializes new objects.

Let's suppose you have the following class:

```
class SimpleClass
{
    public:
        SimpleClass();
    private:
        int myInt;
        string myString;
        vector<int> myVector;
};
```

Let's define the `SimpleClass` constructor as follows:

```
SimpleClass::SimpleClass()
{
    myInt = 5;
    myString = "C++!";
    myVector.resize(10);
}
```

What happens when you create a new instance of the class `MyClass`? It turns out that the simple line of code `MyClass mc` actually causes a cascade of events that goes on behind the scenes. Let's take a look at what happens, step-by-step.

The first thing that C++ does to create an object is to simply allocate enough space for the object. This means that initially all of your object's variables are holding garbage values. In memory, this looks something like this:

New Instance of `SimpleClass`
- Space allocated for all objects

`int myInt`

??????? (garbage)

`string myString` *(object)*

Text: ??????? (garbage) Length: ??????? (garbage)

`vector<int> myVector` *(object)*

Values: ??????? (garbage) Length: ??????? (garbage)

As you can see, none of the instance variables have been initialized, so they all contain junk. At this point, C++ calls the default constructor of each instance variable. For primitive types, this leaves the variables unchanged. After this step, our object looks something like this:

New Instance of `SimpleClass`
- Constructors called on all instance variables

`int myInt`

??????? (garbage)

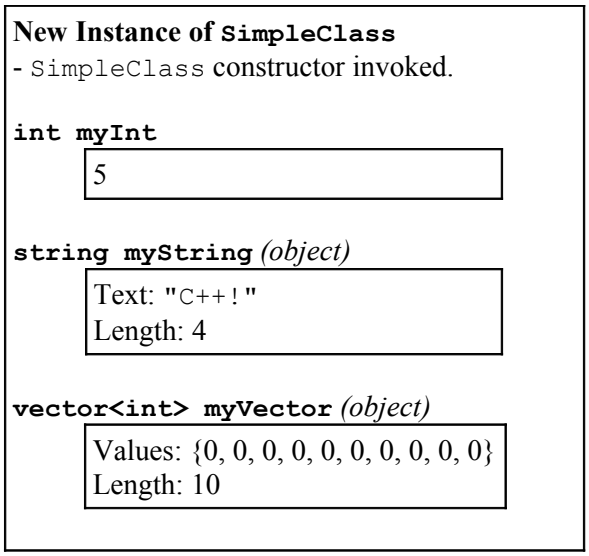
`string myString` *(object)*

Text: "" Length: 0

`vector<int> myVector` *(object)*

Values: { } Length: 0

Finally, C++ will invoke the object's constructor so you can perform any additional initialization code. Using the constructor defined above, the final version of the new object will look like this:



At this point, our object is fully-constructed and ready to use.

However, there's one thing to consider here. Before we reached the `SimpleClass` constructor, C++ called the default constructor on both `myString` and `myVector`. `myString` was therefore initialized to the empty string, and `myVector` was constructed to hold no elements. However, in the `SimpleClass` constructor, we immediately assigned `myString` to hold "C++!" and resized `myVector` to hold ten elements. This means that we effectively initialized `myString` and `myVector` *twice* – once with their default constructors and once in the `SimpleClass` constructor.*

To improve efficiency and resolve certain other problems which we'll explore later, C++ has a feature called an *initializer list*. An initializer list is simply a series of values that C++ will use instead of the default values to initialize instance variables. For example, in the above example, you can use an initializer list to specify that the variable `myString` should be set to "C++!" before the constructor even begins running.

To use an initializer list, you add a colon after the constructor and then list which values to initialize which variables with. For example, here's a modified version of the `SimpleClass` constructor that initializes all the instance variables in an initializer list instead of in the constructor:

```
SimpleClass::SimpleClass() : myInt(5), myString("C++!"), myVector(10)
{
    // Note: Empty constructor
}
```

Here, we're telling C++ to initialize the variables `myInt` and `myString` to 5 and "C++!," respectively, before the class constructor is even called. Also, by writing `myVector(10)`, we're telling C++ to invoke the parametrized constructor of `myVector` passing in the value 10, which creates a `vector` with ten elements. This time, when we create a new object of type `myVector`, the creation steps will look like this:

---

* Technically speaking, the objects are only initialized once, but the runtime efficiency is as though the objects were initialized multiple times. We'll talk about the differences between initialization and assignment later this week.

Space allocated for all objects

`int myInt`

??????? (garbage)

`string myString` *(object)*

Text: ??????? (garbage) Length: ??????? (garbage)

`vector<int> myVector` *(object)*

Values: ??????? (garbage) Length: ??????? (garbage)

Constructors called on all instance variables

`int myInt`

5

`string myString` *(object)*

Text: `"C++!"` Length: 4

`vector<int> myVector` *(object)*

Values: {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } Length: 10

`SimpleClass` constructor invoked. *(no change)*

`int myInt`

5

`string myString` *(object)*

Text: `"C++!"` Length: 4

`vector<int> myVector` *(object)*

Values: {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} Length: 10

As you can see, the values of the instance variables `myInt`, `myString`, and `myVector` are correctly set before the `SimpleClass` constructor is invoked. This is considerably more efficient than the previous version and will run much faster.

Note that while in this example we used initializer lists to initialize all of the object's instance variables, there is no requirement that you do so. However, in practice it's usually a good idea to set up all variables in an initializer list to make clear what values you want for each of your data members.

When Initializer Lists are Mandatory

As seen in this previous example, initializer lists can be quite useful in terms of efficiency. However, there are times where initializer lists are the only syntactically legal way to set up your instance variables.

Suppose we'd like to make an object called `Counter` that supports two functions, `increment` and `decrement`, that adjust an internal counter. However, we'd like to add the restriction that the `Counter` can't drop below 0 or exceed a user-defined limit. Thus we'll use a parametrized constructor that accepts an `int` representing the maximum value for the `Counter` and stores it as an instance variable. Since the value of the upper limit will never change, we'll mark it `const` so that we can't accidentally modify it in our code. The class definition for `Counter` thus looks something like this:

```
class Counter
{
    public:
        Counter(int maxValue);
        void increment();
        void decrement();
        int getValue() const;
    private:
        int value;
        const int maximum;
};
```

Then we'd *like* the constructor to look like this:

```
Counter::Counter(int maxValue)
{
    value = 0;
    maximum = maxValue; // ERROR!
}
```

Unfortunately, the above code isn't valid because in the second line we're assigning a value to a variable marked `const`. Even though we're in the constructor, we still cannot violate the sanctity of `const`ness. To fix this, we'll initialize the value of `maximum` in the initializer list, so that `maximum` will be *initialized* to the value of `maxValue`, rather than *assigned* the value `maxValue`. This is a subtle distinction, so make sure to think about it before proceeding.

The correct version of the constructor is thus

```
Counter::Counter(int maxValue) : value(0), maximum(maxValue)
{
    // Empty constructor
}
```

Note that we initialized `maximum` based on the constructor parameter `maxValue`. Interestingly, if we had forgotten to initialize `maximum` in the initializer list, the compiler would have reported an error. In C++, it is *mandatory* to initialize all `const` primitive-type instance variables in an initializer list. Otherwise, you'd have constants whose values were total garbage.

Another case where initializer lists are mandatory arises when a class contains objects with no legal or meaningful default constructor. Suppose, for example, that you have an object that stores a CS106 `Set` of a custom type `customT` with comparison callback `MyCallback`. Since the `Set` requires you to specify the callback function in the constructor, and since you're always going to use `MyCallback` as that parameter, you might think that the syntax looks like this:

```
class SetWrapperClass
{
    public:
        SetWrapperClass();
    private:
        Set<customT> mySet(MyCallback); // ERROR!
};
```

Unfortunately, this isn't legal C++ syntax. However, you can fix this by rewriting the class as

```
class SetWrapperClass
{
    public:
        SetWrapperClass();
    private:
        Set<customT> mySet; // Note: no parameters specified
};
```

And then initializing `mySet` in the initializer list as

```
SetWrapperClass::SetWrapperClass() : mySet(MyCallback)
{
    // Yet another empty constructor!
}
```

Now, when the object is created, `mySet` will have `MyCallback` passed to its constructor and everything will work out correctly.

Multiple Constructors

If you write a class with multiple constructors (which, after next week's discussion of copy constructors, will be most of your classes), you'll need to make initializer lists for each of your constructors. That is, an initializer list for one constructor won't invoke if a different constructor is called.