8.5a — Constructor member initializer lists

In the previous lesson, for simplicity, we initialized our class member data in the constructor using the assignment operator. For example:

```cpp
class Something
{
private:
    int m_value1;
    double m_value2;
    char m_value3;

public:
    Something()
    {
        // These are all assignments, not initializations
        m_value1 = 1;
        m_value2 = 2.2;
        m_value3 = 'c';
    }
};
```

When the class's constructor is executed, m_value1, m_value2, and m_value3 are created. Then the body of the constructor is run, where the member data variables are assigned values. This is similar to the flow of the following code in non-object-oriented C++:

```cpp
int m_value1;
double m_value2;
char m_value3;

m_value1 = 1;
m_value2 = 2.2;
m_value3 = 'c';
```

While this is valid within the syntax of the C++ language, it does not exhibit good style (and may be less efficient than initialization).

However, as you have learned in previous lessons, some types of data (e.g. const and reference variables) must be initialized on the line they are declared. Consider the following example:

```cpp
class Something
{
private:
    const int m_value;

public:
    Something()
    {
        m_value = 1; // error: const vars can not be assigned to
    }
};
```

This produces code similar to the following:

```cpp
const int m_value; // error: const vars must be initialized with a value
m_value = 5; //  error: const vars can not be assigned to
```

Assigning values to const or reference member variables in the body of the constructor is clearly not sufficient in some cases.

Member initializer lists

To solve this problem, C++ provides a method for initializing class member variables (rather than assigning values to them after they are created) via a **member initializer list** (often called a "member initialization list"). Do not confuse these with the similarly named initializer list that we can use to assign values to arrays.

In lesson **2.1 -- Fundamental variable definition, initialization, and assignment**, you learned that you could initialize variables in three ways: copy, direct, and via uniform initialization (C++11 only).

```
1    int value1 = 1; // copy initialization
2    double value2(2.2); // direct initialization
3    char value3 {'c'} // uniform initialization
```

Using an initialization list is almost identical to doing direct initialization (or uniform initialization in C++11).

This is something that is best learned by example. Revisiting our code that does assignments in the constructor body:

```
1    class Something
2    {
3    private:
4        int m_value1;
5        double m_value2;
6        char m_value3;
7
8    public:
9        Something()
10       {
11           // These are all assignments, not initializations
12           m_value1 = 1;
13           m_value2 = 2.2;
14           m_value3 = 'c';
15       }
16   };
```

Now let's write the same code using an initialization list:

```
1    class Something
2    {
3    private:
4        int m_value1;
5        double m_value2;
6        char m_value3;
7
8    public:
9        Something() : m_value1(1), m_value2(2.2), m_value3('c') // directly initialize our member variables
10       {
11       // No need for assignment here
12       }
13
14       void print()
15       {
16           std::cout << "Something(" << m_value1 << ", " << m_value2 << ", " << m_value3 << ")\n";
17       }
18   };
19
20   int main()
21   {
22       Something something;
23       something.print();
24       return 0;
25   }
```

This prints:

```
Something(1, 2.2, c)
```

The member initializer list is inserted after the constructor parameters. It begins with a colon (:), and then lists each variable to initialize along with the value for that variable separated by a comma.

Note that we no longer need to do the assignments in the constructor body, since the initializer list replaces that functionality. Also note that the initializer list does not end in a semicolon.

Of course, constructors are more useful when we allow the caller to pass in the initialization values:

```cpp
#include <iostream>

class Something
{
private:
    int m_value1;
    double m_value2;
    char m_value3;

public:
    Something(int value1, double value2, char value3='c')
        : m_value1(value1), m_value2(value2), m_value3(value3) // directly initialize our member variab
les
    {
    // No need for assignment here
    }

    void print()
    {
        std::cout << "Something(" << m_value1 << ", " << m_value2 << ", " << m_value3 << ")\n";
    }

};

int main()
{
    Something something(1, 2.2); // value1 = 1, value2=2.2, value3 gets default value 'c'
    something.print();
    return 0;
}
```

This prints:

```
Something(1, 2.2, c)
```

Note that you can use default parameters to provide a default value in case the user didn't pass one in.

Here's an example of a class that has a const member variable:

```cpp
class Something
{
private:
    const int m_value;

public:
    Something(): m_value(5) // directly initialize our const member variable
    {
    }
};
```

This works because we're allowed to initialize const variables (but not assign to them!).

Rule: Use member initializer lists to initialize your class member variables instead of assignment.

Uniform initialization in C++11

In C++11, instead of direct initialization, uniform initialization can be used:

```cpp
class Something
{
private:
    const int m_value;

public:
```

```
 7        Something(): m_value { 5 } // Uniformly initialize our member variables
 8        {
 9        }
10    };
```

We strongly encourage you to begin using this new syntax (even if you aren't using const or reference member variables) as initialization lists are required when doing composition and inheritance (subjects we will be covering shortly).

Rule: Favor uniform initialization over direct initialization if your compiler is C++11 compatible

Initializing array members with member initializer lists

Consider a class with an array member:

```
1    class Something
2    {
3    private:
4        const int m_array[5];
5
6    };
```

Prior to C++11, you can only zero an array member via a member initialization list:

```
 1    class Something
 2    {
 3    private:
 4        const int m_array[5];
 5
 6    public:
 7        Something(): m_array {} // zero the member array
 8        {
 9            // If we want the array to have values, we'll have to use assignment here
10        }
11
12    };
```

However, in C++11, you can fully initialize a member array using uniform initialization:

```
 1    class Something
 2    {
 3    private:
 4        const int m_array[5];
 5
 6    public:
 7        Something(): m_array { 1, 2, 3, 4, 5 } // use uniform initialization to initialize our member array
 8        {
 9        }
10
11    };
```

Initializing member variables that are classes

A member initialization list can also be used to initialize members that are classes.

```
 1    #include <iostream>
 2
 3    class A
 4    {
 5    public:
 6        A(int x) { std::cout << "A " << x << "\n"; }
 7    };
 8
 9    class B
10    {
11    private:
12        A m_a;
13    public:
14        B(int y)
```

```
15              : m_a(y-1) // call A(int) constructor to initialize member m_a
16          {
17              std::cout << "B " << y << "\n";
18          }
19      };
20
21      int main()
22      {
23          B b(5);
24          return 0;
25      }
```

This prints:

```
A 4
B 5
```

When variable b is constructed, the B(int) constructor is called with value 5. Before the body of the constructor executes, m_a is initialized, calling the A(int) constructor with value 4. This prints "A 4". Then control returns back to the B constructor, and the body of the B constructor executes, printing "B 5".

Formatting your initializer lists

C++ gives you a lot of flexibility in how to format your initializer lists, and it's really up to you how you'd like to proceed. But here are some recommendations:

If the initializer list fits on the same line as the function name, then it's fine to put everything on one line:

```
1   class Something
2   {
3   private:
4       int m_value1;
5       double m_value2;
6       char m_value3;
7
8   public:
9       Something() : m_value1(1), m_value2(2.2), m_value3('c') // everything on one line
10      {
11      }
12  };
```

If the initializer list doesn't fit on the same line as the function name, then it should go indented on the next line.

```
1   class Something
2   {
3   private:
4       int m_value1;
5       double m_value2;
6       char m_value3;
7
8   public:
9       Something(int value1, double value2, char value3='c') // this line already has a lot of stuff on it
10          : m_value1(value1), m_value2(value2), m_value3(value3) // so we can put everything indented on
11      next line
12      {
13      }
14
    };
```

If all of the initializers don't fit on a single line (or the initializers are non-trivial), then you can space them out, one per line:

```
1   class Something
2   {
3   private:
4       int m_value1;
5       double m_value2;
```

```
6        char m_value3;
7        float m_value4;
8
9    public:
10       Something(int value1, double value2, char value3='c', float value4=34.6) // this line already has a
11    lot of stuff on it
12           : m_value1(value1), // one per line, commas at end of each line
13       m_value2(value2),
14       m_value3(value3),
15       m_value4(value4)
16    {
17    }
18
     };
```

Initializer list order

Perhaps surprisingly, variables in the initializer list are not initialized in the order that they are specified in the initializer list. Instead, they are initialized in the order in which they are declared in the class.

For best results, the following recommendations should be observed:
1) Don't initialize member variables in such a way that they are dependent upon other member variables being initialized first (in other words, ensure your member variables will properly initialize even if the initialization ordering is different).
2) Initialize variables in the initializer list in the same order in which they are declared in your class. This isn't strictly required so long as the prior recommendation has been followed, but your compiler may give you a warning if you don't do so and you have all warnings turned on.

Summary

Member initializer lists allow us to initialize our members rather than assign values to them. This is the only way to initialize members that require values upon initialization, such as const or reference members, and it can be more performant than assigning values in the body of the constructor. Member initializer lists work both with fundamental types and members that are classes themselves.

Quiz Time

1) Write a class named RGBA that contains 4 member variables of type std::uint8_t named m_red, m_green, m_blue, and m_alpha (#include cstdint to access type std::uint8_t). Assign default values of 0 to m_red, m_green, and m_blue, and 255 to m_alpha. Create a constructor that uses a member initializer list that allows the user to initialize values for m_red, m_blue, m_green, and m_alpha. Include a print() function that outputs the value of the member variables.

If you need a reminder about how to use the fixed width integers, please review lesson **2.4a -- Fixed-width integers and the unsigned controversy**.

Hint: If your print() function isn't working correctly, make sure you're casting uint8_t to an int.

The following code should run:

```
1    int main()
2    {
3        RGBA teal(0, 127, 127);
4        teal.print();
5
6        return 0;
7    }
```

and produce the result:

```
r=0 g=127 b=127 a=255
```

Hide Solution

```
1    #include <iostream>
2    #include <cstdint> // for std::uint8
3
4    class RGBA
```

```cpp
 5    {
 6    private:
 7        std::uint8_t m_red;
 8        std::uint8_t m_green;
 9        std::uint8_t m_blue;
10        std::uint8_t m_alpha;
11
12    public:
13        RGBA(std::uint8_t red=0, std::uint8_t green=0, std::uint8_t blue=0, std::uint8_t alpha=255) :
14            m_red(red), m_green(green), m_blue(blue), m_alpha(alpha)
15        {
16        }
17
18        void print()
19        {
20            std::cout << "r=" << static_cast<int>(m_red) <<
21                " g=" << static_cast<int>(m_green) <<
22                " b=" << static_cast<int>(m_blue) <<
23                " a=" << static_cast<int>(m_alpha) << '\n';
24        }
25    };
26
27    int main()
28    {
29        RGBA teal(0, 127, 127);
30        teal.print();
31
32        return 0;
33    }
```

 8.5b -- Non-static member initialization

 Index

 8.5 -- Constructors

Share this:

134 comments to 8.5a — Constructor member initializer lists

« Older Comments 1 2

Peter Baum
May 16, 2018 at 3:53 pm · Reply

Regarding the non-word "performant" see
https://english.stackexchange.com/questions/38945/what-is-wrong-with-the-word-performant

Alex