LEARN C++
Skill up with our free tutorials

12.6 — Constructor member initializer lists

ALEX SEPTEMBER 9, 2021

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

```
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++:

```
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:

```
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:

```
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 possible 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 1.4 -- Variable assignment and initialization, you learned that you could initialize variables in three ways: copy, direct, and via uniform initialization.

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

Using an initialization list is almost identical to doing direct initialization or uniform initialization.

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

```
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';
    }
};
```

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

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

public:
    Something() : m_value1{ 1 }, m_value2{ 2.2 }, m_value3{ 'c' } // Initialize our member variables
    {
    // No need for assignment here
    }

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

int main()
{
    Something something{};
    something.print();
    return 0;
}
```

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 variables
    {
    // 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.

Initializing array members with member initializer lists

Consider a class with an array member:

```
class Something
{
private:
    const int m_array[5];

};
```

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

```
class Something
{
private:
    const int m_array[5];

public:
    Something(): m_array {} // zero the member array
    {
    }

};
```

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

```
class Something
{
private:
    const int m_array[5];

public:
    Something(): m_array { 1, 2, 3, 4, 5 } // use uniform initialization to initialize our member array
    {
    }

};
```

Initializing member variables that are classes

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

```
#include <iostream>

class A
{
public:
    A(int x = 0) { std::cout << "A " << x << '\n'; }
};

class B
{
private:
    A m_a {};
public:
    B(int y)
        : m_a{ y - 1 } // call A(int) constructor to initialize member
m_a
    {
        std::cout << "B " << y << '\n';
    }
};

int main()
{
    B b{ 5 };
    return 0;
}
```

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:

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

public:
    Something() : m_value1{ 1 }, m_value2{ 2.2 }, m_value3{ 'c' } // everything on one
line
    {
    }
};
```

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

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

public:
    Something(int value1, double value2, char value3='c') // this line already has a lot of stuff on it
        : m_value1{ value1 }, m_value2{ value2 }, m_value3{ value3 } // so we can put everything indented on next line
    {
    }

};
```

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:

```
class Something
{
private:
    int m_value1 {};
    double m_value2 {};
    char m_value3 {};
    float m_value4 {};

public:
    Something(int value1, double value2, char value3='c', float value4=34.6f) // this line already has a lot of stuff on it
        : m_value1{ value1 } // one per line
        , m_value2{ value2 }
        , m_value3{ value3 }
        , m_value4{ value4 }
    {
    }

};
```

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

Question #1

Write a class named RGBA that contains 4 member variables of type std::uint_fast8_t named m_red, m_green, m_blue, and m_alpha (#include cstdint to access type std::uint_fast8_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 4.6 -- Fixed-width integers and size_t.

Hint: If your print() function isn't working correctly, make sure you're casting uint_fast8_t to an int.

The following code should run:

```
int main()
{
 RGBA teal{ 0, 127, 127
};
 teal.print();

 return 0;
}
```

and produce the result:

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

Show Solution

Next lesson
12.7 Non-static member initialization

Back to table of contents

Previous lesson
12.5 Constructors

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ cod

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: 🔔 POST COMMENT

DP N N FOUT

Newest

X

X