14.10 — Constructor member initializer lists

learncpp.com/cpp-tutorial/constructor-member-initializer-lists/

This lesson continues our introduction of constructors from lesson 14.9 -- Introduction to constructors.

Member initialization via a member initialization list

To have a constructor initialize members, we do so using a **member initializer list** (often called a “member initialization list”). Do not confuse this with the similarly named “initializer list” that is used to initialize aggregates with a list of values.

Member initialization lists are something that is best learned by example. In the following example, our `Foo(int, int)` constructor has been updated to use a member initializer list to initialize `m_x`, and `m_y`:

```
#include <iostream>

class Foo
{
private:
    int m_x {};
    int m_y {};

public:
    Foo(int x, int y)
        : m_x { x }, m_y { y } // here's our member initialization list
    {
        std::cout << "Foo(" << x << ", " << y << ") constructed\n";
    }

    void print() const
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
    }
};

int main()
{
    Foo foo{ 6, 7 };
    foo.print();

    return 0;
}
```

The member initializer list is defined after the constructor parameters. It begins with a colon (:), and then lists each member to initialize along with the initialization value for that variable, separated by a comma. You must use a direct form of initialization here (preferably using braces, but parentheses works as well) -- using copy initialization (with an equals) does not work here. Also note that the member initializer list does not end in a semicolon.

This program produces the following output:

```
Foo(6, 7) constructed
Foo(6, 7)
```

When `foo` is instantiated, the members in the initialization list are initialized with the specified initialization values. In this case, the member initializer list initializes `m_x` to the value of `x` (which is `6`), and `m_y` to the value of `y` (which is `7`). Then the body of the constructor runs.

When the `print()` member function is called, you can see that `m_x` still has value `6` and `m_y` still has value `7`.

Member initializer list formatting

C++ provides a lot of freedom to format your member initializer lists as you prefer, as it doesn't care where you put your colon, commas, or whitespace.

The following styles are all valid (and you're likely to see all three in practice):

```
Foo(int x, int y) : m_x { x }, m_y { y }
{
}
```

```
Foo(int x, int y) :
    m_x { x },
    m_y { y }
{
}
```

```
Foo(int x, int y)
    : m_x { x }
    , m_y { y }
{
}
```

Our recommendation is to use the third style above:

- Put the colon on the line after the constructor name, as this cleanly separates the member initializer list from the function prototype.
- Indent your member initializer list, to make it easier to see the function names.

If the member initialization list is short/trivial, all initializers can go on one line:

```
Foo(int x, int y)
    : m_x { x }, m_y { y }
{
}
```

Otherwise (or if you prefer), each member and initializer pair can be placed on a separate line (starting with a comma to maintain alignment):

```
Foo(int x, int y)
    : m_x { x }
    , m_y { y }
{
}
```

Member initialization order

Because the C++ standard says so, the members in a member initializer list are always initialized in the order in which they are defined inside the class (not in the order they are defined in the member initializer list).

In the above example, because `m_x` is defined before `m_y` in the class definition, `m_x` will be initialized first (even if it is not listed first in the member initializer list).

Because we intuitively expect variables to be initialized left to right, this can cause subtle errors to occur. Consider the following example:

```
#include <algorithm> // for std::max
#include <iostream>

class Foo
{
private:
    int m_x{};
    int m_y{};

public:
    Foo(int x, int y)
        : m_y{ std::max(x, y) }, m_x{ m_y } // issue on this line
    {
    }

    void print() const
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
    }
};

int main()
{
    Foo foo{ 6, 7 };
    foo.print();

    return 0;
}
```

In the above example, our intent is to calculate the larger of the initialization values passed in (via `std::max(x, y)` and then use this value to initialize both `m_x` and `m_y`. However, on the author's machine, the following result is printed:

```
Foo(-858993460, 7)
```

What happened? Even though `m_y` is listed first in the member initialization list, because `m_x` is defined first in the class, `m_x` gets initialized first. And `m_x` gets initialized to the value of `m_y`, which hasn't been initialized yet. Finally, `m_y` gets initialized to the greater of the initialization values.

To help prevent such errors, members in the member initializer list should be listed in the order in which they are defined in the class. Some compilers will issue a warning if members are initialized out of order.

Best practice

Member variables in a member initializer list should be listed in order that they are defined in the class.

It’s also a good idea to avoid initializing members using the value of other members (if possible). That way, even if you do make a mistake in the initialization order, it shouldn’t matter because there are no dependencies between initialization values.

Member initializer list vs default member initializers

Members can be initialized in a few different ways:

- If a member is listed in the member initializer list, that initialization value is used
- Otherwise, if the member has a default member initializer, that initialization value is used
- Otherwise, the member is default initialized.

This means that if a member has both a default member initializer and is listed in the member initializer list for the constructor, the member initializer list value takes precedence.

Here’s an example showing all three initialization methods:

```
#include <iostream>

class Foo
{
private:
    int m_x{};    // default member initializer (will be ignored)
    int m_y{ 2 }; // default member initializer (will be used)
    int m_z;      // no initializer

public:
    Foo(int x)
        : m_x{ x } // member initializer list
    {
        std::cout << "Foo constructed\n";
    }

    void print() const
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ", " << m_z << ")\n";
    }
};

int main()
{
    Foo foo{ 6 };
    foo.print();

    return 0;
}
```

On the author’s machine, this output:

```
Foo constructed
Foo(6, 2, -858993460)
```

Here's what's happening. When `foo` is constructed, only `m_x` appears in the member initializer list, so `m_x` is first initialized to `6`. `m_y` is not in the member initialization list, but it does have a default member initializer, so it is initialized to `2`. `m_z` is neither in the member initialization list, nor does it have a default member initializer, so it is default initialized (which for fundamental types, means it is left uninitialized). Thus, when we print the value of `m_z`, we get undefined behavior.

Constructor function bodies

The bodies of constructors functions are most often left empty. This is because we primarily use constructor for initialization, which is done via the member initializer list. If that is all we need to do, then we don't need any statements in the body of the constructor.

However, because the statements in the body of the constructor execute after the member initializer list has executed, we can add statements to do any other setup tasks required. In the above examples, we print something to the console to show that the constructor executed, but we could do other things like open a file or database, allocate memory, etc…

New programmers sometimes use the body of the constructor to assign values to members:

```
#include <iostream>

class Foo
{
private:
    int m_x{};
    int m_y{};

public:
    Foo(int x, int y)
    {
        m_x = x; // incorrect: this is an assignment, not an initialization
        m_y = y; // incorrect: this is an assignment, not an initialization
    }

    void print() const
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
    }
};

int main()
{
    Foo foo{ 6, 7 };
    foo.print();

    return 0;
}
```

Although in this simple case this will produce the expected result, in case where members are required to be initialized (such as for data members that are const or references) assignment will not work.

Best practice

Prefer using the member initializer list to initialize your members over assigning values in the body of the constructor.

Quiz time

Question #1

Write a class named Ball. Ball should have two private member variables, one to hold a color, and one to hold a radius. Also write a function to print out the color and radius of the ball.

The following sample program should compile:

```
int main()
{
        Ball blue{ "blue", 10.0 };
        print(blue);

        Ball red{ "red", 12.0 };
        print(red);

        return 0;
}
```

and produce the result:

```
Ball(blue, 10)
Ball(red, 12)
```

Show Solution

Question #2

Why did we make `print()` a non-member function instead of a member function?

Show Solution

Question #3

Why did we make `m_color` a `std::string` instead of a `std::string_view`?

Show Solution