10.7 — std::initializer_list

BY ALEX ON MARCH 9TH, 2017 | LAST MODIFIED BY ALEX ON JUNE 6TH, 2018

Consider a fixed array of integers in C++:

```
int array[5];
```

If we want to initialize this array with values, we can do so directly via the initializer list syntax:

```
int main()
{
    int array[5] { 5, 4, 3, 2, 1 }; // initializer list
    for (int count=0; count < 5; ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

This prints:

```
5 4 3 2 1
```

This also works for dynamically allocated arrays:

```
int main()
{
    int *array = new int[5] { 5, 4, 3, 2, 1 }; // initializer list
    for (int count = 0; count < 5; ++count)
        std::cout << array[count] << ' ';
    delete[] array;

    return 0;
}
```

In the previous lesson, we introduced the concept of container classes, and showed an example of an IntArray class that holds an array of integers:

```
#include <cassert> // for assert()

class IntArray
{
private:
    int m_length;
    int *m_data;

public:
    IntArray():
        m_length(0), m_data(nullptr)
    {
    }

    IntArray(int length):
        m_length(length)
    {
        m_data = new int[length];
    }

    ~IntArray()
    {
        delete[] m_data;
        // we don't need to set m_data to null or m_length to 0 here, since the object will be destroye
d immediately after this function anyway
    }
```

```
    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() { return m_length; }
};
```

What happens if we try to use an initializer list with this container class?

```
int main()
{
    IntArray array { 5, 4, 3, 2, 1 }; // this line doesn't compile
    for (int count=0; count < 5; ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

This code won't compile, because the IntArray class doesn't have a constructor that knows what to do with an initializer list. As a result, we're left initializing our array elements individually:

```
int main()
{
    IntArray array(5);
    array[0] = 5;
    array[1] = 4;
    array[2] = 3;
    array[3] = 2;
    array[4] = 1;

    for (int count=0; count < 5; ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

That's not so great.

Prior to C++11, list initialization could only be used with static or dynamic arrays. However, as of C++11, we now have a solution to this problem.

Class initialization using std::initializer_list

When a C++11 compiler sees an initializer list, it automatically convert it into an object of type std::initializer_list. Therefore, if we create a constructor that takes a std::initializer_list parameter, we can create objects using the initializer list as an input.

std::initializer_list lives in the <initializer_list> header.

There are a few things to know about std::initializer_list. Much like std::array or std::vector, you have to tell std::initializer_list what type of data the list holds using angled brackets. Therefore, you'll never see a plain std::initializer_list. Instead, you'll see something like std::initializer_list<int> or std::initializer_list<std::string>.

Second, std::initializer_list has a (misnamed) size() function which returns the number of elements in the list. This is useful when we need to know the length of the list passed in.

Let's take a look at updating our IntArray class with a constructor that takes a std::initializer_list.

```
#include <cassert> // for assert()
#include <initializer_list> // for std::initializer_list
#include <iostream>

class IntArray
{
private:
```

```
    int m_length;
    int *m_data;

public:
    IntArray() :
        m_length(0), m_data(nullptr)
    {
    }

    IntArray(int length) :
        m_length(length)
    {
        m_data = new int[length];
    }

    IntArray(const std::initializer_list<int> &list): // allow IntArray to be initialized via list init
ialization
        IntArray(list.size()) // use delegating constructor to set up initial array
    {
        // Now initialize our array from the list
        int count = 0;
        for (auto &element : list)
        {
            m_data[count] = element;
            ++count;
        }
    }

    ~IntArray()
    {
        delete[] m_data;
        // we don't need to set m_data to null or m_length to 0 here, since the object will be destroye
d immediately after this function anyway
    }

    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() { return m_length; }
};

int main()
{
    IntArray array { 5, 4, 3, 2, 1 }; // initializer list
    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

This produces the expected result:

```
5 4 3 2 1
```

It works! Now, let's explore this in more detail.

Here's our IntArray constructor that takes a std::initializer_list<int>.

```
    IntArray(const std::initializer_list<int> &list): // allow IntArray to be initialized via list init
ialization
        IntArray(list.size()) // use delegating constructor to set up initial array
    {
        // Now initialize our array from the list
```

```
        int count = 0;
        for (auto &element : list)
        {
            m_data[count] = element;
            ++count;
        }
    }
```

On line 1: As noted above, we have to use angled brackets to denote what type of element we expect inside the list. In this case, because this is an IntArray, we'd expect the list to be filled with int. Note that we also pass the list by const reference, so we don't make an unnecessary copy of the std::initializer_list when it's passed to our constructor.

On line 2: We delegate allocating memory for the IntArray to the other constructor via a delegating constructor (to reduce redundant code). This other constructor needs to know the length of the array, so we pass it list.size(), which contains the number of elements in the list.

The body of the constructor is reserved for copying the elements from the list into our IntArray class. For some inexplicable reason, std::initializer_list does not provide access to the elements of the list via subscripting (operator[]). The omission has been noted many times and never addressed.

However, there are easy ways to work around the lack of subscripts. The easiest way is to use a for-each loop here. The for-each loops steps through each element of the initialization list, and we can manually copy the elements into our internal array.

Class assignment using std::initializer_list

You can also use std::initializer_list to assign new values to a class by overriding the assignment operator to take a std::initializer_list parameter. This work analogously to the above. We'll show an example of how to do this in the quiz solution below.

Note that if you implement a constructor that takes a std::initializer_list, you should ensure you do at least one of the following:

1. Provide an overloaded list assignment operator
2. Provide a proper deep-copying copy assignment operator
3. Make the constructor explicit, so it can't be used for implicit conversions

Here's why: consider the above class (which doesn't have an overloaded list assignment or a copy assignment), along with following statement:

```
    array = { 1, 3, 5, 7, 9, 11 }; // overwrite the elements of array with the elements from the list
```

First, the compiler will note that an assignment function taking a std::initializer_list doesn't exist. Next it will look for other assignment functions it could use, and discover the implicitly provided copy assignment operator. However, this function can only be used if it can convert the initializer list into an IntArray. Because the constructor that takes a std::initializer_list isn't marked as explicit, the compiler will use the list constructor to convert the initializer list into a temporary IntArray. Then it will call the implicit assignment operator, which will shallow copy the temporary IntArray into our array object.

At this point, both the temporary IntArray's m_data and array->m_data point to the same address (due to the shallow copy). You can already see where this is going.

At the end of the assignment statement, the temporary IntArray is destroyed. That calls the destructor, which deletes the temporary IntArray's m_data. This leaves our array variable with a hanging m_data pointer. When you try to use array->m_data for any purpose (including when array goes out of scope and the destructor goes to delete m_data), you'll get undefined results (and probably a crash).

Rule: If you provide list construction, it's a good idea to provide list assignment as well.

Summary

Implementing a constructor that takes a std::initializer_list parameter (by reference to prevent copying) allows us to use list initialization with our custom classes. We can also use std::initializer_list to implement other functions that need to use an initializer list, such as an assignment operator.

Quiz time

1) Using the IntArray class above, implement an overloaded assignment operator that takes an initializer list.

The following code should run:

```
int main()
{
    IntArray array { 5, 4, 3, 2, 1 }; // initializer list
    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    std::cout << '\n';

    array = { 1, 3, 5, 7, 9, 11 };

    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

This should print:

```
5 4 3 2 1
1 3 5 7 9 11
```

Hide Solution

```
#include <cassert> // for assert()
#include <initializer_list> // for std::initializer_list
#include <iostream>

class IntArray
{
private:
    int m_length;
    int *m_data;

public:
    IntArray() :
        m_length(0), m_data(nullptr)
    {
    }

    IntArray(int length) :
        m_length(length)
    {
        m_data = new int[length];
    }

    IntArray(const std::initializer_list<int> &list) : // allow IntArray to be initialized via list ini
tialization
        IntArray(list.size()) // use delegating constructor to set up initial array
    {
        // Now initialize our array from the list
        int count = 0;
        for (auto &element : list)
        {
            m_data[count] = element;
            ++count;
        }
    }

    ~IntArray()
    {
        delete[] m_data;
        // we don't need to set m_data to null or m_length to 0 here, since the object will be destroye
d immediately after this function anyway
    }

    IntArray& operator=(const std::initializer_list<int> &list)
```

```
    {
        // If the new list is a different size, reallocate it
        if (list.size() != static_cast<size_t>(m_length))
        {
            // delete any existing elements
            delete[] m_data;

            // reallocate array
            m_length = list.size();
            m_data = new int[m_length];
        }

        // Now initialize our array from the list
        int count = 0;
        for (auto &element : list)
        {
            m_data[count] = element;
            ++count;
        }

        return *this;
    }

    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() { return m_length; }
};

int main()
{
    IntArray array { 5, 4, 3, 2, 1 }; // initializer list
    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    std::cout << '\n';

    array = { 1, 3, 5, 7, 9, 11 };

    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```


10.x -- Chapter 10 comprehensive quiz

Index

10.6 -- Container classes