23.6 — Container classes

In real life, we use containers all the time. Your breakfast cereal comes in a box, the pages in your book come inside a cover and binding, and you might store any number of items in containers in your garage. Without containers, it would be extremely inconvenient to work with many of these objects. Imagine trying to read a book that didn't have any sort of binding, or eat cereal that didn't come in a box without using a bowl. It would be a mess. The value the container provides is largely in its ability to help organize and store items that are put inside it.

Similarly, a **container class** is a class designed to hold and organize multiple instances of another type (either another class, or a fundamental type). There are many different kinds of container classes, each of which has various advantages, disadvantages, and restrictions in their use. By far the most commonly used container in programming is the array, which you have already seen many examples of. Although C++ has built-in array functionality, programmers will often use an array container class (std::array or std::vector) instead because of the additional benefits they provide. Unlike built-in arrays, array container classes generally provide dynamic resizing (when elements are added or removed), remember their size when they are passed to functions, and do bounds-checking. This not only makes array container classes more convenient than normal arrays, but safer too.

Container classes typically implement a fairly standardized minimal set of functionality. Most well-defined containers will include functions that:

- Create an empty container (via a constructor)
- Insert a new object into the container
- Remove an object from the container
- Report the number of objects currently in the container
- Empty the container of all objects
- Provide access to the stored objects
- Sort the elements (optional)

Sometimes certain container classes will omit some of this functionality. For example, arrays container classes often omit the insert and remove functions because they are slow and the class designer does not want to encourage their use.

Container classes implement a member-of relationship. For example, elements of an array are members-of (belong to) the array. Note that we're using "member-of" in the conventional sense, not the C++ class member sense.

Types of containers

Container classes generally come in two different varieties. **Value containers** are compositions that store copies of the objects that they are holding (and thus are responsible for creating and destroying those copies). **Reference containers** are aggregations that store pointers or references to other objects (and thus are not responsible for creation or destruction of those objects).

Unlike in real life, where containers can hold whatever types of objects you put in them, in C++, containers typically only hold one type of data. For example, if you have an array of integers, it will only hold integers. Unlike some other languages, many C++ containers do not allow you to arbitrarily mix types. If you need containers to hold integers and doubles, you will generally have to write two separate containers to do this (or use templates, which is an advanced C++ feature). Despite the restrictions on their use, containers are immensely useful, and they make programming easier, safer, and faster.

An array container class

In this example, we are going to write an integer array class from scratch that implements most of the common functionality that containers should have. This array class is going to be a value container, which will hold copies of the elements it's organizing. As the name suggests, the container will hold an array of integers, similar to `std::vector<int>`.

First, let's create the IntArray.h file:

```
#ifndef INTARRAY_H
#define INTARRAY_H

class IntArray
{
};

#endif
```

Our IntArray is going to need to keep track of two values: the data itself, and the size of the array. Because we want our array to be able to change in size, we'll have to do some dynamic allocation, which means we'll have to use a pointer to store the data.

```
#ifndef INTARRAY_H
#define INTARRAY_H

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

#endif
```

Now we need to add some constructors that will allow us to create IntArrays. We are going to add two constructors: one that constructs an empty array, and one that will allow us to construct an array of a predetermined size.

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <cassert> // for assert()

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

public:
    IntArray() = default;

    IntArray(int length):
        m_length{ length }
    {
        assert(length >= 0);

        if (length > 0)
            m_data = new int[length]{};
    }
};

#endif
```

We'll also need some functions to help us clean up IntArrays. First, we'll write a destructor, which simply deallocates any dynamically allocated data. Second, we'll write a function called erase(), which will erase the array and set the length to 0.

```
~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 destroyed immediately after this function anyway
}

void erase()
{
    delete[] m_data;

    // We need to make sure we set m_data to nullptr here, otherwise it will
    // be left pointing at deallocated memory!
    m_data = nullptr;
    m_length = 0;
}
```

Now let’s overload the [] operator so we can access the elements of the array. We should ensure the index parameter has a valid value, which we can do via by using the assert() function. We’ll also add an access function to return the length of the array. Here’s everything so far:

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <cassert> // for assert()

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

public:
    IntArray() = default;

    IntArray(int length):
        m_length{ length }
    {
        assert(length >= 0);

        if (length > 0)
            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 destroyed immediately after this function anyway
    }

    void erase()
    {
        delete[] m_data;
        // We need to make sure we set m_data to nullptr here, otherwise it will
        // be left pointing at deallocated memory!
        m_data = nullptr;
        m_length = 0;
    }

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

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

#endif
```

At this point, we already have an IntArray class that we can use. We can allocate IntArrays of a given size, and we can use the [] operator to retrieve or change the value of the elements.

However, there are still a few thing we can't do with our IntArray. We still can't change its size, still can't insert or delete elements, and we still can't sort it. Copying the array will also cause problems, since that will shallow copy the data pointer.

First, let's write some code that will allow us to resize an array. We are going to write two different functions to do this. The first function, reallocate(), will destroy any existing elements in the array when it is resized, but it will be fast. The second function, resize(), will keep any existing elements in the array when it is resized, but it will be slow.

```
#include <algorithm> // for std::copy_n

    // reallocate resizes the array.  Any existing elements will be destroyed.  This
function operates quickly.
    void reallocate(int newLength)
    {
        // First we delete any existing elements
        erase();

        // If our array is going to be empty now, return here
        if (newLength <= 0)
            return;

        // Then we have to allocate new elements
        m_data = new int[newLength];
        m_length = newLength;
    }

    // resize resizes the array.  Any existing elements will be kept.  This function
operates slowly.
    void resize(int newLength)
    {
        // if the array is already the right length, we're done
        if (newLength == m_length)
            return;

        // If we are resizing to an empty array, do that and return
        if (newLength <= 0)
        {
            erase();
            return;
        }

        // Now we can assume newLength is at least 1 element.  This algorithm
        // works as follows: First we are going to allocate a new array.  Then we
        // are going to copy elements from the existing array to the new array.
        // Once that is done, we can destroy the old array, and make m_data
        // point to the new array.

        // First we have to allocate a new array
        int* data{ new int[newLength] };

        // Then we have to figure out how many elements to copy from the existing
        // array to the new array.  We want to copy as many elements as there are
        // in the smaller of the two arrays.
        if (m_length > 0)
        {
            int elementsToCopy{ (newLength > m_length) ? m_length : newLength };
            std::copy_n(m_data, elementsToCopy, data); // copy the elements
        }

        // Now we can delete the old array because we don't need it any more
```

```
        delete[] m_data;

        // And use the new array instead!  Note that this simply makes m_data point
        // to the same address as the new array we dynamically allocated.  Because
        // data was dynamically allocated, it won't be destroyed when it goes out of
scope.
        m_data = data;
        m_length = newLength;
    }
```

Whew! That was a little tricky!

Let’s also add a copy constructor and assignment operator so we can copy the array.

```
IntArray(const IntArray& a): IntArray(a.getLength()) // use normal constructor to set
size of array appropriately
{
    std::copy_n(a.m_data, m_length, m_data); // copy the elements
}

IntArray& operator=(const IntArray& a)
{
    // Self-assignment check
    if (&a == this)
        return *this;

    // Set the size of the new array appropriately
    reallocate(a.getLength());
    std::copy_n(a.m_data, m_length, m_data); // copy the elements

    return *this;
}
```

Many array container classes would stop here. However, just in case you want to see how insert and delete functionality would be implemented we’ll go ahead and write those too. Both of these algorithms are very similar to resize().

```
void insertBefore(int value, int index)
 {
     // Sanity check our index value
     assert(index >= 0 && index <= m_length);

     // First create a new array one element larger than the old array
     int* data{ new int[m_length+1] };

     // Copy all of the elements up to the index
     std::copy_n(m_data, index, data);

     // Insert our new element into the new array
     data[index] = value;

     // Copy all of the values after the inserted element
     std::copy_n(m_data + index, m_length - index, data + index + 1);

     // Finally, delete the old array, and use the new array instead
     delete[] m_data;
     m_data = data;
     ++m_length;
 }

 void remove(int index)
 {
     // Sanity check our index value
     assert(index >= 0 && index < m_length);

     // If this is the last remaining element in the array, set the array to empty
and bail out
     if (m_length == 1)
     {
         erase();
         return;
     }

     // First create a new array one element smaller than the old array
     int* data{ new int[m_length-1] };

     // Copy all of the elements up to the index
     std::copy_n(m_data, index, data);

     // Copy all of the values after the removed element
     std::copy_n(m_data + index + 1, m_length - index - 1, data + index);

     // Finally, delete the old array, and use the new array instead
     delete[] m_data;
     m_data = data;
     --m_length;
 }
```

Here is our IntArray container class in its entirety.

IntArray.h:

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <algorithm> // for std::copy_n
#include <cassert> // for assert()

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

public:
    IntArray() = default;

    IntArray(int length):
        m_length{ length }
    {
        assert(length >= 0);

        if (length > 0)
            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 destroyed immediately after this function anyway
    }

    IntArray(const IntArray& a): IntArray(a.getLength()) // use normal constructor to
set size of array appropriately
    {
        std::copy_n(a.m_data, m_length, m_data); // copy the elements
    }

    IntArray& operator=(const IntArray& a)
    {
        // Self-assignment check
        if (&a == this)
            return *this;

        // Set the size of the new array appropriately
        reallocate(a.getLength());
        std::copy_n(a.m_data, m_length, m_data); // copy the elements

        return *this;
    }

    void erase()
    {
```

```
        delete[] m_data;
        // We need to make sure we set m_data to nullptr here, otherwise it will
        // be left pointing at deallocated memory!
        m_data = nullptr;
        m_length = 0;
    }

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


    // reallocate resizes the array.  Any existing elements will be destroyed.  This
function operates quickly.
    void reallocate(int newLength)
    {
        // First we delete any existing elements
        erase();

        // If our array is going to be empty now, return here
        if (newLength <= 0)
            return;

        // Then we have to allocate new elements
        m_data = new int[newLength];
        m_length = newLength;
    }

    // resize resizes the array.  Any existing elements will be kept.  This function
operates slowly.
    void resize(int newLength)
    {
        // if the array is already the right length, we're done
        if (newLength == m_length)
            return;

        // If we are resizing to an empty array, do that and return
        if (newLength <= 0)
        {
            erase();
            return;
        }

        // Now we can assume newLength is at least 1 element.  This algorithm
        // works as follows: First we are going to allocate a new array.  Then we
        // are going to copy elements from the existing array to the new array.
        // Once that is done, we can destroy the old array, and make m_data
        // point to the new array.

        // First we have to allocate a new array
```

```
        int* data{ new int[newLength] };

        // Then we have to figure out how many elements to copy from the existing
        // array to the new array.  We want to copy as many elements as there are
        // in the smaller of the two arrays.
        if (m_length > 0)
        {
            int elementsToCopy{ (newLength > m_length) ? m_length : newLength };
            std::copy_n(m_data, elementsToCopy, data); // copy the elements
        }

        // Now we can delete the old array because we don't need it any more
        delete[] m_data;

        // And use the new array instead!  Note that this simply makes m_data point
        // to the same address as the new array we dynamically allocated.  Because
        // data was dynamically allocated, it won't be destroyed when it goes out of
scope.
        m_data = data;
        m_length = newLength;
    }

    void insertBefore(int value, int index)
    {
        // Sanity check our index value
        assert(index >= 0 && index <= m_length);

        // First create a new array one element larger than the old array
        int* data{ new int[m_length+1] };

        // Copy all of the elements up to the index
        std::copy_n(m_data, index, data);

        // Insert our new element into the new array
        data[index] = value;

        // Copy all of the values after the inserted element
        std::copy_n(m_data + index, m_length - index, data + index + 1);

        // Finally, delete the old array, and use the new array instead
        delete[] m_data;
        m_data = data;
        ++m_length;
    }

    void remove(int index)
    {
        // Sanity check our index value
        assert(index >= 0 && index < m_length);

        // If this is the last remaining element in the array, set the array to empty
and bail out
```

```
        if (m_length == 1)
        {
            erase();
            return;
        }

        // First create a new array one element smaller than the old array
        int* data{ new int[m_length-1] };

        // Copy all of the elements up to the index
        std::copy_n(m_data, index, data);

        // Copy all of the values after the removed element
        std::copy_n(m_data + index + 1, m_length - index - 1, data + index);

        // Finally, delete the old array, and use the new array instead
        delete[] m_data;
        m_data = data;
        --m_length;
    }

    // A couple of additional functions just for convenience
    void insertAtBeginning(int value) { insertBefore(value, 0); }
    void insertAtEnd(int value) { insertBefore(value, m_length); }

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

#endif
```

Now, let’s test it just to prove it works:

```
#include <iostream>
#include "IntArray.h"

int main()
{
    // Declare an array with 10 elements
    IntArray array(10);

    // Fill the array with numbers 1 through 10
    for (int i{ 0 }; i<10; ++i)
        array[i] = i+1;

    // Resize the array to 8 elements
    array.resize(8);

    // Insert the number 20 before element with index 5
    array.insertBefore(20, 5);

    // Remove the element with index 3
    array.remove(3);

    // Add 30 and 40 to the end and beginning
    array.insertAtEnd(30);
    array.insertAtBeginning(40);

    // A few more tests to ensure copy constructing / assigning arrays
    // doesn't break things
    {
        IntArray b{ array };
        b = array;
        b = b;
        array = array;
    }

    // Print out all the numbers
    for (int i{ 0 }; i<array.getLength(); ++i)
        std::cout << array[i] << ' ';

    std::cout << '\n';

    return 0;
}
```

This produces the result:

```
40 1 2 3 5 20 6 7 8 30
```

Although writing container classes can be pretty complex, the good news is that you only have to write them once. Once the container class is working, you can use and reuse it as often as you like without any additional programming effort required.

A few additional improvements that could/should be made: First, we could have made this a template class, so that it would work with any copyable type rather than just int. Second, we should add const overloads of various member functions to properly support const IntArrays. Third, we should add support for move semantics (via adding a move constructor and move assignment).

One more thing: If a class in the standard library meets your needs, use that instead of creating your own. For example, instead of using IntArray, you're better off using `std::vector<int>`. It's battle tested, efficient, and plays nicely with the other classes in the standard library. But sometimes you need a specialized container class that doesn't exist in the standard library, so it's good to know how to create your own when you need to. We'll talk more about containers in the standard library once we've covered a few more fundamental topics.