10.4 — Container classes

BY ALEX ON DECEMBER 14TH, 2007 | LAST MODIFIED BY ALEX ON OCTOBER 1ST, 2015

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 it's 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 class. 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 instead because of the additional benefits it provides. Unlike built-in arrays, array container classes generally provide dynamically resizing (when elements are added or removed) 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 delete functions because they are slow and the class designer does not want to encourage their use.

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 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, C++ generally does not allow you to mix types inside a container. If you want one container class that holds integers and another that holds doubles, you will 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 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 its organizing.

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

```
1 #ifndef INTARRAY_H
2 #define INTARRAY_H
3
4 class IntArray
5 {
6 };
7
```

```
#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_nLength;
    int *m_pnData;
};

#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

class IntArray
{
private:
    int m_nLength;
    int *m_pnData;

public:
    IntArray()
    {
        m_nLength = 0;
        m_pnData = 0;
    }

    IntArray(int nLength)
    {
        m_pnData = new int[nLength];
        m_nLength = nLength;
    }
};

#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_pnData;
    }

    void Erase()
    {
        delete[] m_pnData;
        // We need to make sure we set m_pnData to 0 here, otherwise it will
        // be left pointing at deallocated memory!
        m_pnData = 0;
        m_nLength = 0;
    }
```

Now let's overload the [] operator so we can access the elements of the array. We should bounds check the index to make sure it's valid, which is best done using the assert() function. We'll also add an access function to return the length of the array.

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <assert.h> // for assert()

class IntArray
{
private:
    int m_nLength;
    int *m_pnData;

public:
    IntArray()
    {
        m_nLength = 0;
        m_pnData = 0;
    }

    IntArray(int nLength)
    {
        m_pnData = new int[nLength];
        m_nLength = nLength;
    }

    ~IntArray()
    {
        delete[] m_pnData;
    }

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

    int& operator[](int nIndex)
    {
        assert(nIndex >= 0 && nIndex < m_nLength);
        return m_pnData[nIndex];
    }

    int GetLength() { return m_nLength; }
};

#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 it's size, still can't insert or delete elements, and we still can't sort it.

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.

```
    // Reallocate resizes the array.  Any existing elements will be destroyed.
    // This function operates quickly.
    void Reallocate(int nNewLength)
    {
        // First we delete any existing elements
        Erase();

```

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

        // Then we have to allocate new elements
        m_pnData = new int[nNewLength];
        m_nLength = nNewLength;
    }

    // Resize resizes the array.  Any existing elements will be kept.
    // This function operates slowly.
    void Resize(int nNewLength)
    {
        // If we are resizing to an empty array, do that and return
        if (nNewLength <= 0)
        {
            Erase();
            return;
        }

        // Now we can assume nNewLength 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_pnData
        // point to the new array.

        // First we have to allocate a new array
        int *pnData = new int[nNewLength];

        // 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_nLength > 0)
        {
            int nElementsToCopy = (nNewLength > m_nLength) ? m_nLength : nNewLength;

            // Now copy the elements one by one
            for (int nIndex=0; nIndex < nElementsToCopy; nIndex++)
                pnData[nIndex] = m_pnData[nIndex];
        }

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

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

Whew! That was a little tricky!

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 nValue, int nIndex)
    {
        // Sanity check our nIndex value
        assert(nIndex >= 0 && nIndex <= m_nLength);

        // First create a new array one element larger than the old array
        int *pnData = new int[m_nLength+1];

        // Copy all of the elements up to the index
        for (int nBefore=0; nBefore < nIndex; nBefore++)
            pnData[nBefore] = m_pnData[nBefore];
```

```

        // Insert our new element into the new array
        pnData[nIndex] = nValue;

        // Copy all of the values after the inserted element
        for (int nAfter=nIndex; nAfter < m_nLength; nAfter++)
            pnData[nAfter+1] = m_pnData[nAfter];

        // Finally, delete the old array, and use the new array instead
        delete[] m_pnData;
        m_pnData = pnData;
        m_nLength += 1;
    }

    void Remove(int nIndex)
    {
        // Sanity check our nIndex value
        assert(nIndex >= 0 && nIndex < m_nLength);

        // First create a new array one element smaller than the old array
        int *pnData = new int[m_nLength-1];

        // Copy all of the elements up to the index
        for (int nBefore=0; nBefore < nIndex; nBefore++)
            pnData[nBefore] = m_pnData[nBefore];

        // Copy all of the values after the removed element
        for (int nAfter=nIndex+1; nAfter < m_nLength; nAfter++)
            pnData[nAfter-1] = m_pnData[nAfter];

        // Finally, delete the old array, and use the new array instead
        delete[] m_pnData;
        m_pnData = pnData;
        m_nLength -= 1;
    }

    // A couple of additional functions just for convenience
    void InsertAtBeginning(int nValue) { InsertBefore(nValue, 0); }
    void InsertAtEnd(int nValue) { InsertBefore(nValue, m_nLength); }
```

Here is our IntArray container class in it's entirety:

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <assert.h> // for assert()

class IntArray
{
private:
    int m_nLength;
    int *m_pnData;

public:
    IntArray()
    {
        m_nLength = 0;
        m_pnData = 0;
    }

    IntArray(int nLength)
    {
        m_pnData = new int[nLength];
        m_nLength = nLength;
    }

    ~IntArray()
```

```
26      {
27          delete[] m_pnData;
28      }
29
30      void Erase()
31      {
32          delete[] m_pnData;
33          // We need to make sure we set m_pnData to 0 here, otherwise it will
34          // be left pointing at deallocated memory!
35          m_pnData = 0;
36          m_nLength = 0;
37      }
38
39      int& operator[](int nIndex)
40      {
41          assert(nIndex >= 0 && nIndex < m_nLength);
42          return m_pnData[nIndex];
43      }
44
45      // Reallocate resizes the array.  Any existing elements will be destroyed.
46      // This function operates quickly.
47      void Reallocate(int nNewLength)
48      {
49          // First we delete any existing elements
50          Erase();
51
52          // If our array is going to be empty now, return here
53          if (nNewLength<= 0)
54              return;
55
56          // Then we have to allocate new elements
57          m_pnData = new int[nNewLength];
58          m_nLength = nNewLength;
59      }
60
61      // Resize resizes the array.  Any existing elements will be kept.
62      // This function operates slowly.
63      void Resize(int nNewLength)
64      {
65          // If we are resizing to an empty array, do that and return
66          if (nNewLength <= 0)
67          {
68              Erase();
69              return;
70          }
71
72          // Now we can assume nNewLength is at least 1 element.  This algorithm
73          // works as follows: First we are going to allocate a new array.  Then we
74          // are going to copy elements from the existing array to the new array.
75          // Once that is done, we can destroy the old array, and make m_pnData
76          // point to the new array.
77
78          // First we have to allocate a new array
79          int *pnData = new int[nNewLength];
80
81          // Then we have to figure out how many elements to copy from the existing
82          // array to the new array.  We want to copy as many elements as there are
83          // in the smaller of the two arrays.
84          if (m_nLength > 0)
85          {
86              int nElementsToCopy = (nNewLength > m_nLength) ? m_nLength : nNewLength;
87
88              // Now copy the elements one by one
89              for (int nIndex=0; nIndex < nElementsToCopy; nIndex++)
90                  pnData[nIndex] = m_pnData[nIndex];
91          }
92
```

```
93          // Now we can delete the old array because we don't need it any more
94          delete[] m_pnData;
95
96          // And use the new array instead!  Note that this simply makes m_pnData point
97          // to the same address as the new array we dynamically allocated.  Because
98          // pnData was dynamically allocated, it won't be destroyed when it goes out of scope.
99          m_pnData = pnData;
100         m_nLength = nNewLength;
101     }
102
103
104         void InsertBefore(int nValue, int nIndex)
105     {
106         // Sanity check our nIndex value
107         assert(nIndex >= 0 && nIndex <= m_nLength);
108
109         // First create a new array one element larger than the old array
110         int *pnData = new int[m_nLength+1];
111
112         // Copy all of the elements up to the index
113         for (int nBefore=0; nBefore < nIndex; nBefore++)
114             pnData[nBefore] = m_pnData[nBefore];
115
116         // insert our new element into the new array
117         pnData[nIndex] = nValue;
118
119         // Copy all of the values after the inserted element
120         for (int nAfter=nIndex; nAfter < m_nLength; nAfter++)
121             pnData[nAfter+1] = m_pnData[nAfter];
122
123         // Finally, delete the old array, and use the new array instead
124         delete[] m_pnData;
125         m_pnData = pnData;
126         m_nLength += 1;
127     }
128
129     void Remove(int nIndex)
130     {
131         // Sanity check our nIndex value
132         assert(nIndex >= 0 && nIndex < m_nLength);
133
134         // First create a new array one element smaller than the old array
135         int *pnData = new int[m_nLength-1];
136
137         // Copy all of the elements up to the index
138         for (int nBefore=0; nBefore < nIndex; nBefore++)
139             pnData[nBefore] = m_pnData[nBefore];
140
141         // Copy all of the values after the inserted element
142         for (int nAfter=nIndex+1; nAfter < m_nLength; nAfter++)
143             pnData[nAfter-1] = m_pnData[nAfter];
144
145         // Finally, delete the old array, and use the new array instead
146         delete[] m_pnData;
147         m_pnData = pnData;
148         m_nLength -= 1;
149     }
150
151     // A couple of additional functions just for convenience
152     void InsertAtBeginning(int nValue) { InsertBefore(nValue, 0); }
153     void InsertAtEnd(int nValue) { InsertBefore(nValue, m_nLength); }
154
155     int GetLength() { return m_nLength; }
156 };
157
158 #endif
```

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

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

using namespace std;

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

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

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

    // Insert the number 20 before the 5th element
    cArray.InsertBefore(20, 5);

    // Remove the 3rd element
    cArray.Remove(3);

    // Add 30 and 40 to the end and beginning
    cArray.InsertAtEnd(30);
    cArray.InsertAtBeginning(40);

    // Print out all the numbers
    for (int j=0; j<cArray.GetLength(); j++)
        cout << cArray[j] << " ";

    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.

It is also worth explicitly mentioning that even though our sample IntArray container class holds a built-in data type (int), we could have just as easily used a user-defined type (eg. a point class).

 **11.1 -- Introduction to inheritance**

 **Index**

 **10.3 -- Aggregation**

Share this: