

10.4 — Container classes

BY ALEX, ON DECEMBER 14TH, 2007

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
8  #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.

```
1  #ifndef INTARRAY_H
2  #define INTARRAY_H
3
4  class IntArray
5  {
6  private:
7      int m_nLength;
8      int *m_pnData;
9  };
10
11 #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.

```
1  #ifndef INTARRAY_H
2  #define INTARRAY_H
3
4  class IntArray
```

```

5     {
6     private:
7         int m_nLength;
8         int *m_pnData;
9
10    public:
11        IntArray()
12        {
13            m_nLength = 0;
14            m_pnData = 0;
15        }
16
17        IntArray(int nLength)
18        {
19            m_pnData = new int[nLength];
20            m_nLength = nLength;
21        }
22    };
23
24    #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.

```

1     ~IntArray()
2     {
3         delete[] m_pnData;
4     }
5
6     void Erase()
7     {
8         delete[] m_pnData;
9         // We need to make sure we set m_pnData to 0 here, otherwise it will
10        // be left pointing at deallocated memory!
11        m_pnData = 0;
12        m_nLength = 0;
13    }

```

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.

```

1     #ifndef INTARRAY_H
2     #define INTARRAY_H
3
4     #include <assert.h> // for assert()
5
6     class IntArray
7     {
8     private:
9         int m_nLength;
10        int *m_pnData;
11
12    public:
13        IntArray()
14        {
15            m_nLength = 0;
16            m_pnData = 0;
17        }
18
19        IntArray(int nLength)
20        {
21            m_pnData = new int[nLength];
22            m_nLength = nLength;
23        }

```

```

24
25     ~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     int GetLength() { return m_nLength; }
46 };
47
48 #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 things 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.

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.

```

1     // Reallocate resizes the array. Any existing elements will be destroyed.
2     // This function operates quickly.
3     void Reallocate(int nNewLength)
4     {
5         // First we delete any existing elements
6         Erase();
7
8         // If our array is going to be empty now, return here
9         if (nNewLength <= 0)
10            return;
11
12        // Then we have to allocate new elements
13        m_pnData = new int[nNewLength];
14        m_nLength = nNewLength;
15    }
16
17    // Resize resizes the array. Any existing elements will be kept.
18    // This function operates slowly.
19    void Resize(int nNewLength)
20    {
21        // If we are resizing to an empty array, do that and return
22        if (nNewLength <= 0)
23        {
24            Erase();
25            return;
26        }
27
28        // Now we can assume nNewLength is at least 1 element. This algorithm
29        // works as follows: First we are going to allocate a new array. Then we

```

```

30 // are going to copy elements from the existing array to the new array.
31 // Once that is done, we can destroy the old array, and make m_pnData
32 // point to the new array.
33
34 // First we have to allocate a new array
35 int *pnData = new int[nNewLength];
36
37 // Then we have to figure out how many elements to copy from the existing
38 // array to the new array. We want to copy as many elements as there are
39 // in the smaller of the two arrays.
40 if (m_nLength > 0)
41 {
42     int nElementsToCopy = (nNewLength > m_nLength) ? m_nLength : nNewLength;
43
44     // Now copy the elements one by one
45     for (int nIndex=0; nIndex < nElementsToCopy; nIndex++)
46         pnData[nIndex] = m_pnData[nIndex];
47 }
48
49 // Now we can delete the old array because we don't need it any more
50 delete[] m_pnData;
51
52 // And use the new array instead! Note that this simply makes m_pnData point
53 // to the same address as the new array we dynamically allocated. Because
54 // pnData was dynamically allocated, it won't be destroyed when it goes out of
55 scope.
56 m_pnData = pnData;
57 m_nLength = nNewLength;
58 }

```

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()`.

```

1 void InsertBefore(int nValue, int nIndex)
2 {
3     // Sanity check our nIndex value
4     assert(nIndex >= 0 && nIndex <= m_nLength);
5
6     // First create a new array one element larger than the old array
7     int *pnData = new int[m_nLength+1];
8
9     // Copy all of the elements up to the index
10    for (int nBefore=0; nBefore < nIndex; nBefore++)
11        pnData[nBefore] = m_pnData[nBefore];
12
13    // Insert our new element into the new array
14    pnData[nIndex] = nValue;
15
16    // Copy all of the values after the inserted element
17    for (int nAfter=nIndex; nAfter < m_nLength; nAfter++)
18        pnData[nAfter+1] = m_pnData[nAfter];
19
20    // Finally, delete the old array, and use the new array instead
21    delete[] m_pnData;
22    m_pnData = pnData;
23    m_nLength += 1;
24 }
25
26 void Remove(int nIndex)
27 {
28     // Sanity check our nIndex value
29     assert(nIndex >= 0 && nIndex < m_nLength);
30

```

```

31 // First create a new array one element smaller than the old array
32 int *pnData = new int[m_nLength-1];
33
34 // Copy all of the elements up to the index
35 for (int nBefore=0; nBefore < nIndex; nBefore++)
36     pnData[nBefore] = m_pnData[nBefore];
37
38 // Copy all of the values after the inserted element
39 for (int nAfter=nIndex+1; nAfter < m_nLength; nAfter++)
40     pnData[nAfter-1] = m_pnData[nAfter];
41
42 // Finally, delete the old array, and use the new array instead
43 delete[] m_pnData;
44 m_pnData = pnData;
45 m_nLength -= 1;
46 }
47
48 // A couple of additional functions just for convenience
49 void InsertAtBeginning(int nValue) { InsertBefore(nValue, 0); }
50 void InsertAtEnd(int nValue) { InsertBefore(nValue, m_nLength); }

```

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

```

1  #ifndef INTARRAY_H
2  #define INTARRAY_H
3
4  #include <assert.h> // for assert()
5
6  class IntArray
7  {
8  private:
9      int m_nLength;
10     int *m_pnData;
11
12 public:
13     IntArray()
14     {
15         m_nLength = 0;
16         m_pnData = 0;
17     }
18
19     IntArray(int nLength)
20     {
21         m_pnData = new int[nLength];
22         m_nLength = nLength;
23     }
24
25     ~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 // Reallocate resizes the array. Any existing elements will be destroyed.
45 // This function operates quickly.
46 void Reallocate(int nNewLength)
47 {
48     // First we delete any existing elements
49     Erase();
50
51     // If our array is going to be empty now, return here
52     if (nNewLength <= 0)
53         return;
54
55     // Then we have to allocate new elements
56     m_pnData = new int[nNewLength];
57     m_nLength = nNewLength;
58 }
59
60 // Resize resizes the array. Any existing elements will be kept.
61 // This function operates slowly.
62 void Resize(int nNewLength)
63 {
64     // If we are resizing to an empty array, do that and return
65     if (nNewLength <= 0)
66     {
67         Erase();
68         return;
69     }
70
71     // Now we can assume nNewLength is at least 1 element. This algorithm
72     // works as follows: First we are going to allocate a new array. Then we
73     // are going to copy elements from the existing array to the new array.
74     // Once that is done, we can destroy the old array, and make m_pnData
75     // point to the new array.
76
77     // First we have to allocate a new array
78     int *pnData = new int[nNewLength];
79
80     // Then we have to figure out how many elements to copy from the existing
81     // array to the new array. We want to copy as many elements as there are
82     // in the smaller of the two arrays.
83     if (m_nLength > 0)
84     {
85         int nElementsToCopy = (nNewLength > m_nLength) ? m_nLength : nNewLength;
86
87         // Now copy the elements one by one
88         for (int nIndex=0; nIndex < nElementsToCopy; nIndex++)
89             pnData[nIndex] = m_pnData[nIndex];
90     }
91
92     // Now we can delete the old array because we don't need it any more
93     delete[] m_pnData;
94
95     // And use the new array instead! Note that this simply makes m_pnData point
96     // to the same address as the new array we dynamically allocated. Because
97     // pnData was dynamically allocated, it won't be destroyed when it goes out of
98     scope.
99     m_pnData = pnData;
100     m_nLength = nNewLength;
101 }
102
103 void InsertBefore(int nValue, int nIndex)
104 {
105     // Sanity check our nIndex value
106     assert(nIndex >= 0 && nIndex <= m_nLength);
107 }
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:

```

1  #include <iostream>
2  #include "IntArray.h"
3
4  using namespace std;
5
6  int main()
7  {
8      // Declare an array with 10 elements
9      IntArray cArray(10);
10
11     // Fill the array with numbers 1 through 10
12     for (int i=0; i<10; i++)
13         cArray[i] = i+1;

```



```

14
15 // Resize the array to 8 elements
16 cArray.Resize(8);
17
18 // Insert the number 20 before the 5th element
19 cArray.InsertBefore(20, 5);
20
21 // Remove the 3rd element
22 cArray.Remove(3);
23
24 // Add 30 and 40 to the end and beginning
25 cArray.InsertAtEnd(30);
26 cArray.InsertAtBeginning(40);
27
28 // Print out all the numbers
29 for (int j=0; j<cArray.GetLength(); j++)
30     cout << cArray[j] << " ";
31
32 return 0;
33 }

```

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:

