Software Design Using C++

STL: Vectors

What are Vectors?

The Standard Template Library's vectors are somewhat like arrays but are considerably more flexible. Unlike a typical array, a vector can grow and shrink in size as needed, although there can sometimes be a performance penalty when th size is increased. But like an array, a vector can be indexed using the [] notation. There are class functions available for inserting a new item just before the item pointed to by an iterator, erasing (removing) an item, pushing an item onto the e of the vector, popping an item off of the end of the vector, returning the size of the vector, telling you if the vector is empt clearing out the contents of the vector, etc. There are also several constructors for the vector class, with the default constructor building an empty vector. One can also use the various algorithms, such as those for sorting and searching, which are available for vectors and the other container classes.

Example

- **vector.cpp example**

Basics of Using Vectors

The above example illustrates some of what can be done with vectors. To use vectors you must include the `vector` header file. The `std` namespace is also needed in order to have access to the names used in the STL for the classes ar functions. If you want to use any of the algorithms provided by the STL, you also need to include the `algorithm` heade Remember that these algorithms are separate from the functions provided by the vector container class itself.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

To create a vector one uses something like the following. Note the use of `int` in angle brackets to specify that we want a vector of integers. The name of the vector is Num and it starts out with size 8.

```
vector<int> Num(8);
```

We can then use indexing to access individual items in the Num vector. For example, we can assign an item to the 0-th vector slot as follows:

```
Num[0] = 17;
```

Note the first section of code for printing out the contents of the vector. It, too, uses indexing. It also illustrates the use of `size` function to look up the current size of the vector. That is certainly more convenient than keeping a separate intege counter like we did with arrays.

```
for (k = 0; k < Num.size(); k++)
   cout << Num[k] << " ";
```

There are several ways to grow the vector. One is to push a new item onto the end (back) of the vector. In our example program this is done as follows:

```
Num.push_back(-55);
```

Adding a new item to the end of a vector can usually be done in constant time, but on occasion the vector will have to be relocated in memory, an operation that takes linear time, that is, Theta(n) time. On average, though, `push_back` would be expected to take constant time.

Iterators

We can also use the `insert` class function to insert a new item at a specific place in the vector. There are actually several versions of `insert`. Our sample program uses the simplest; it inserts a single item. To use this function we need an iterator that points to the location before which to insert. (Remember that an iterator is like a pointer.) Note how our example creates an iterator named p for use on a vector of integers. The iterator can be initialized by assigning it a value such as `Num.begin()`. The `begin` class function returns an iterator to the first item in the vector. In our example we add 4 to this, which gives a pointer to the item at index 4 in the vector.

```
vector<int>::iterator p;
cout << "We can insert -44 before the item at index 4." << endl;
p = Num.begin() + 4;
Num.insert(p, -44);
```

Next, let's look at how the example program uses an iterator in printing the contents of the vector. This is an alternative to using indexing. Note that iterator p is initialized to point to the first item in the vector. Each time around the loop, p is incremented with the ++ operator. The loop test is to see whether p is not equal to `Num.end()`, which returns an iterator to the item one past the end of vector. (Just off the end is just right for loop control!) Each time around the loop we output the value `*p`, which is the value pointed at (as with regular pointers).

```
for (p = Num.begin(); p != Num.end(); p++)
   cout << *p << " ";
```

The code to print the contents of a vector can also be packaged up as a (generic) function. Look at the function which is shown below. It can print a vector having any type of component (as long as that type is printable using the << stream operator.) Note that the vector parameter is made a reference parameter for efficiency.

```
template <class T> void Print(vector<T> & Vec)
   {
   vector<T>::iterator p;

   cout << "Contents of vector:" << endl;
   for (p = Vec.begin(); p != Vec.end(); p++)
      cout << *p << " ";
   cout << endl << endl;
   }
```

Algorithms

Finally we look at some of the standard algorithms available for the vector and other container classes. One very useful one is `find`, which does a search for a given item within a certain section of the container (a vector in this case). The following code from our example searches for a 7 in the `Num` vector. Here we search the whole vector since the first and second parameters are iterators to the start and end (one off the end, actually) of the vector. The `find` returns an iterator to the first place where the item was found. If the item was not found, `find` returns an iterator to the end of the container (really one past the last item, as always). Note that we do not know (or need to know) the code for the `find` routine. All we know is how to call it and the fact that the STL guarantees that it runs in linear time.

```
p = find(Num.begin(), Num.end(), 7);
if (p == Num.end())
   cout  << "Could not find 7 in the vector"  << endl;
else
   cout  << "Have found 7 in the vector"  << endl;
```

Another very useful algorithm is `sort`, which is used to sort a section of the container. In the example below, we sort the entire vector by passing in as parameters iterators to the start and one past the end of the vector. This certainly makes it easy to sort as there is no code to write for the sort routine! Supposedly the STL guarantees that the `sort` algorithm runs in Theta(n * lg(n)) time, which is very good for sorting. There is no guarantee about what underlying algorithm might be used to perform the sorting. (For example, it might be mergesort or heapsort, but we do not know for sure.)

```
sort(Num.begin(), Num.end());
```

There are many more algorithms and much more that can be done with vectors, but we will not go further with those topics here. See the **references** for more information.

Related Items

- **Pointers**
  Intermediate Topic.
- **Basics of Arrays**
  Introductory Topic.
- **Arrays**
  Intermediate Topic.

Back to the main page for *Software Design Using C++*

Authors: **Br. David Carlson** and Br. Isidore Minerd
Last updated: August 27, 2009