An Expandable Vector in c++

Introduction

Expandable is a template. It provides a vector of objects that grows as needed. It solves the problem of reading a file of data of unknown length with a simple data structure. Most of the overhead of using Expandable comes at the beginning when the data structure is growing. After that it may be treated as a c++ one column array (i.e. vector v[i], where v is an Expandable object and i is an integer). It may be sorted with quicksort. It may be scanned by a simple for loop. If sorted it may be searched with a binary search. Elements may be added or deleted at any time using the same procedure as with a simple c++ vector (i.e. moving elements of the array as needed).

Unlike a simple c++ vector, the Expandable template can provide some of the procedural aspects of addition and deletion of elements in the vector at any position in the array. In addition to adding a new element to the vector at the end or at an index not at the end, an element may be added in a manner that sorts the data in the vector. (i.e. insertion sort). Note the insertion sort also ensures that the insertions are unique based on a user define key.

Expanding the vector involves moving the data. This could be expensive if the element is very large so an alternative template, ExpandableP, behaves very much like an Expandable object but only a pointer to the datum appears in the vector. The rest of the application sees the objects that are stored but the management of the pointer is left to ExpandableP.

When a class contains a single Expandable(P) object an iterator type may be defined by using the IterT template. The type may then be used to create an iterator that will return a pointer to each element in the Expandable(P) object (i.e. scan the vector). This is usually done in a simple “for loop”.

What is the programming cost of using an Expandable(P) object? One cost is that the element in the vector must have a few operations defined. If an iterator type is defined, then the class holding the Expandable(P) object will need a few very small operations (mostly private) and a friend declaration for the iterator type.

Expandable (both versions) manage the heap objects (allocating and deallocating) and executing the constructors and destructors of the element objects at the proper times. Assistance is provided by ExpandableP for external allocation and deallocation of objects that are stored in the ExpandableP object if that becomes desirable from a programming point of view.

Expandable Basics

Suppose one needs to store a vector of an unknown number of widgets. With Expandable this is how it would be done:

```
#include “Expandable.h”

Class Widget { … }

Class WidgetStore {
Expandable<Widget, 2> data;
Public:
  WidgetStore() { }
  …
  }
```

The data object initially contains space for 2 Widget objects. They are data[0] and data[1]. If one references data[2] then data is doubled in size and all the data is copied from the old data to the new data. Widget must contain at least the following functions:

```
Class Widget {
…
Public:
```

```
  Widget(Widget& w) {copy(w);}

  Widget& operator= (widget& w) {copy(w); return *this;}

Private:

  Void copy(Widget& w) {datum0 = w.datum0; … }
  }
```

ExpandableP Basics

If the widget is characterized by only a few variables then Expandable is good enough. But it can be expensive when the copy function gets large. So why not move a pointer to the widget data rather than the actual data. Using the Widget class declaration above add right after the Widget class declaration:

```
DatumPtrT<Widget> WidgetP;

Class WidgetStore {

ExpandableP<Widget, WidgetP, 2> data;

Public:
  …
  }
```

Now no copy operation is required in Widget as it is done in WidgetP. Indexing into the vector is performed in exactly the same way as with Expandable: data[0], data[1], data[2]. With the latter index causing the vector to double in size.

Scanning the Data

Adding data is easy, just add one to the index for each new data item. Then scanning the data later should also be easy:

```
void WidgetStore::doSomething {
int i;
int n = data.end();

  for (i = 0; i < n; i++) {
    Widget& w = data[i];
    …
    }
```

This works for Expandable but if it is an ExpandableP vector then the Widget reference would be:

```
Widget& w = *data[i].p;
```

Note that data is usually private so access outside of the class is difficult.

The Iterator Template

An iterator is an object that returns a pointer to each object in the vector, one at a time. Here is an example:

```
WidgetStore widgetStore;


Void doSomething {
WSIter  iter(widgetStore);
Widget* w;

  for (w = iter(); w; w = iter++) {
```

```
    <use w>
    }
```

The pointer to w is either a pointer to a Widget or zero. The test in the for loop is for true which in c is any non-zero value. Furthermore, the function, doSomething does not need to be a WidgetStore member function.

Implementation of the iterator requires creating the type WSIter and installing a few private functions in the WidgetStore class. Here are the member functions needed to implement the iterator for WidgetStore:

```
#include "IterT.h"

Class WidgetStore;
Typedef IterT< WidgetStore, Widget> WSIter;


Class WidgetStore {

Expandable<Widget, 2> data;

Public:
  WidgetStore() { }
  …
Private:

  // returns either a pointer to data (or datum) at index i in array or 0

  Widget* datum(int i) {return 0 <= i && i < nData() ? &data[i] : 0;}

  int     nData() {return data.end();}
                                  // returns number of data items in array

  void    removeDatum(int i) {if (0 <= i && i < nData()) data.del(i);}

  friend typename WSIter;
  }
```

The above function for datum works with Expandable. The datum reference must be changed from "&data[i]" to "data[i].p" for ExpandableP. The header files for both versions contains the iterator store member functions and declarations.

Adding Widgets to the Vector

An extension of the vector model is to allow an easy way to add elements to the vector. There are two methods in each template for adding an element to the vector:

```
data += widget;

data = widget;
```

The first operator, "+=", adds the new element at the end of the vector. The second operator, "=". Is intended to perform an insertion sort. In order for the insertion sort to work the following member functions must be included in the Widget class:

```
Class Widget {
Public:

  Widget();

  …
```

```
// Required for Insertion Sort, i.e. data = dtm;
bool operator >= (Widget& w) {return key >= w.key;}
bool operator == (Widget& w) {return key == w.key;}
…
}
```

Adding Widgets to an Expandable Vector

The "+=" operator adds an element to end of the vector. In order to do that it must copy the Widget object presented in the expression to the vector. Likewise to insert the entry in the sort position the Widget object must be copied into the vector.

Another method of adding an element to the end of the vector is to request a reference to the next available element in the vector:

```
Widget& w = data.nextData();
```

The reference, w, in this case must be initialized with data where the call is made to nextData(). This is slightly more efficient than the "+=" operator.

Adding Widgets to an ExpandableP Vector

The "+=" operator behaves differently for a reference to a Widget and a pointer to a Widget. If one defines a Widget, say on the stack, the object is copied into an allocated node and added to the end of the data vector:

```
void WidgetStore::doSomething {
Widget w;

  <add data to w>

  data += w;      // Data copied into allocated node and added to vector
  …
  }
```

If a pointer to a Widget is appended to the vector it is added directly to the end of the vector. This mechanisms assumes that the object has been allocated from the heap. There is a function to do that for you:

```
void WidgetStore::doSomething {
Widget* w = data.allocate();

  <add data to *w object>

  Data += w;      // Node is add to end of vector
  …
  }
```

The insertion sort operator, "=" behaves in a similar manner with respect to references and pointers.

Binary Search

When the data is sorted it then is possible to perform a binary search to find an element in the vector. There is a member function for performing the search:

```
Widget* w = data.bSearch(key);
```

In order to implement this the Widget class must contain the following member functions:

```
// Required for Binary Search
bool operator== (Key* key) {return this->key == key;}
bool operator<  (Key* key) {return this->key <  key;}
bool operator>  (Key* key) {return this->key >  key;}
```

Where the Key type is determined by the Widget class. The interesting thing about binary search is that it is really useful when the vector is big.

Linear Search

A linear search is also available:

```
Widget* w = data.find(key);
```

Iterator Details

The iterator template produces an object with a link to the class and an index value that contains several attributes. One can use it to scan the vector in the forward and reverse directions:

```
for (w = iter();            w; w = iter++) …    // scan in forward dir
for (w = iter(WSIter::Fwd); w; w = iter++) …    // scan in forward dir
for (w – iter(WSIter::Rev); w; w = iter--) …    // scan in reverse dir
```

When the iterator is active the following attributes of the iterator are available:

```
int     i       = iter.index();    // The current index in the vector
Widget* w       = iter.current();  // The current entry in the vector
bool    isFirst = iter.isFirst();  // index is zero
bool    isLast  = iter.isLast();   // index is last entry in vector
```

Sometimes one starts a scan and then needs to start a new scan from the middle of the vector where the iterator is located. This can be achieved by copying one iterator to another. This can be done two different ways:

```
for (w = iter(); w; w = iter++) {
  …
  if (w.x == key) {
    WSIter iter2(iter);          // Initizlize iter2 from iter
    for (t = iter2(); …) …
    }
  If (w.y == key) {
    WSIter iter3(widgetStore);
    iter3 = iter;                // copy iter to iter3
    }
  }
```

Conclusion

While this is not a highly efficient with respect to cpu cycles but it is highly convenient.