```
Title:       Expandable(P) Template
Description: Simple Data Structure for storage of multiple objects of a single class
Abstract:    How does one store multiple versions of a single class?  Use a hash table to get fast lookups but it
             can't be sorted.  Use a tree (balanced or otherwise) and the complexity of the code is significant.
             Sorting a tree multiple ways is also not trivial.  Use a linked list and sorting is really
             difficult.  Scanning a linked list backwards requires two pointers (also messy).

             Speed or efficiency are important in real time applications, not so much in Man-Machine
             Applications.  Expandable(P) takes the idea of a one column c++ array (i.e. a vector) in which
             the element is a class (i.e. the description of an object) and turns it into a storage vehicle.

Author:      Bob Van Tuyl
Email:       rrvt@swde.com

Language:    c++
Platform:    Any c++ platform
Technology:
Topic:       data

Section      c++ Tip or Trick
SubSection   ?

License:     Enter the license (MIT)
```

**Download Example Source - 83 Kb**

## Preface to This Update

After writing the article and submitting it I went about updating several applications to use the new versions of Expandable. During that effort it became clear that the binary search implementation I put forward in the article had a glaring flaw in that the equality test came first and uselessly for most of the search loop. The second observation is the sorted insertion operator (e.g. data = datum; where data is an Expandable vector and datum is a node to be inserted in the vector) could be improved by searching backwards through the vector so that if the data was already sorted the insertion would be a little more efficient. The last observaion made during use of the new versions of Expandable centered around the insertion. The insertion sometimes involves copying the data from one node to another as it is put into the vector. However, it is sometimes useful to have a pointer to the node that is in the vector after the insertion. Thus I redefined the operator= and operator += to return a pointer to the resident node in the vector. Most of the time it is not used, but sometimes it results in not having to perform a search for the node one just put in the vector!

## 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++ array (i.e. 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.

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 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 typedef may be defined by using the IterT template. The typedef 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).

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 typedef 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 typename.

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(P) -- An expanding c++ Vector

### Example Code

The code included is a simple console application that uses MFC. Expandable(P) doesn't need MFC but it is useful for other things in the example.

I've arranged the console application to demonstrate the various operations that may be performed with Expandable(P). The code contains two versions of Store and one main function. The examples may be executed by specifying one or more command line arguments designating the operation to be performed (i.e. `ExpandableExample Display Sort LoadSorted`). Here is a list of the operations that may be performed.

- Display -- Simple array references for load and display
- Sort -- Simple load into data, sort the array and then display the result
- Append -- Load the vector with the append operator: `data += dtm;`
- LoadSorted -- Load the array with the insert unique object into sorted vector: `data = dtm;`
- IterDisplay -- Display vector using an iterator
- BSearch -- Perform a sorted load and then do a binary search for every object in the Store object using the key from each object.
- LinearSrch -- Perform a sorted load and then do a linear search for every object in the Store object using the key from each object.
- LoadStoreP -- Load an ExpandableP object with a slightly more complex object for each line and display the resulting vector with an iterator.
- LoadSortedP -- Load an ExpandableP object with the insertion sort option
- BSearchP -- Perform a sorted load and then do a binary search for every object in the StoreP object using the key from each object.
- LinearSrchP -- Perform a sorted load and then do a linear search for every object in the Store object using the key from each object.

The code was compiled with Visual Studio 2017 (VS17) with the target being Win10 in the latest version of VS17. The code was compiled using Unicode characters and the MFC dll library and my personal library. The parent ExpandableExample directory contains a Solution Directory, ExpandableExample and a Project Directory, ExpandableExample.prj. The parent ExpandableExample directory should have the same parent as the Library directory in order to compile with Visual Studio with no changes to the project properties.

A short note on identifiers:

- A user defined class or typedef is capitalized (e.g. Store is a class)
- A user defined object is not capitalized (e.g. store is an object)

## The First Example -- Simple load and store into an Expandable Object

### Store Header File (i.e. Store.h)

The header file defines data, an expandable vector. The vector is initially 2 objects in length. The object in the vector is a line from the file and the line number. The line number will be useful when the sort is performed. Every object stored in an Expandable(P) object must contain a copy constructor, `Datum(Datum& dtm)`, and an copy assignment operator, `Datum& operator= (Datum& dtm)`.

```
// Store Example


#pragma once
#include "Expandable.h"
#include "FileIO.h"


class Datum {

public:

ulong  key;                         // A 5 digit key created to be unique
String line;                                                // Line from the file

  Datum() : key(0) { }
  Datum(Datum& dtm) {copy(dtm);}

  Datum& operator=(Datum& dtm) {copy(dtm); return *this;}

private:

  void copy(Datum& dtm) {key = dtm.key;  line = dtm.line;}
  };


class Store {

String               path;
Expandable<Datum, 2> data;

public:

  Store() { }
 ~Store() { }

  void load(TCchar* filePath);

  void display();
  };
```

```
extern Store store;
```

### Loading a File into Data

The simplest use of an Expandable(p) object is demonstrated in the following code that loads a file into the "data" object. My FileIO module interfaces with Windows through MFC to open and read from a file. The key Expandable code is: `Datum& dtm = data[i];`. It looks like an array reference, behaves like an array reference but quietly expands the array when necessary. The vector size is doubled when it needs to expand. Hopefully this mitigates the expense of finding the right size.

```
void Store::load(TCchar* filePath) {
FileIO fil;
int    i;
String line;
Random rand(213);

  path = filePath;   data.clear();

  if (fil.open(path, FileIO::Read)) {
    wcout << filePath << " opened" << endl;

    for (i = 0; fil.read(line); i++) {
      Datum& dtm = data[i];
      ulong  r   = ulong(rand.next() * 1000.0);             // construct a random number

      dtm.key = r * 100 + i; dtm.line = line.trimRight();   // Shift the random number and make unique
      }
    }

  fil.close();
  }
```

### Displaying the Data

Using a reference variable is unnecessary as the display function demonstrates.

```
void Store::display() {
int    n = data.end();
int    i;
String s;

  for (i = 0; i < n; i++) {
    s.format(_T("%5i: "), data[i].key);

    wcout << s.str() << data[i].line.str() << endl;
    }
  }
```

### Qsort requires a Vector and Two Datum Boolean Operators

Qsort operates on a vector. Add the following code to the Store class: `void sort() {qsort(&data[0], &data[data.end()-1]);}`, call it and displaying the results will yield the file content sorted in the Expandable(P) object. However, when one attempts to compile the code, two functions will be missing from Datum:

```
  bool operator>  (Datum& dtm) {return line >  dtm.line;}
  bool operator<= (Datum& dtm) {return line <= dtm.line;}
```

Datum is sorted in this case by comparing the two lines of data but it could just as well be the unique keys created during the load process. Since Datum is a class composed by the programmer the comparison can be anything desired.

### Two Alternative Load Functions

Expandable(P) is a class so additional functions may be added to it. Instead of computing `data[i]` for each load, one can use the append operator `data += dtm`, where dtm is a Datum object.

If the objects in the vector are unique and two additional boolean operators are added to the Datum class then a Datum may be inserted into data sorted: `data = dtm`. Here are the two boolean operators required in Datum for this to work:

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

Note that in this case the unique key is used to sort the data on input. This is done to support demonstrating a binary search later. If the Datum object is found in the vector during the insertion processing then it is not inserted. This is required by the "uniqueness" clause.

## Expandable Operations

Here is a list of the operations supported by Expandable:

| | |
|---|---|
| `datum = data[i];` | Where 0 <= i < data.end() (i.e. endN) |
| `data[i] = datum;` | Array expands to encompass i |
| `data.clear();` | Content is ignored but number of elements (endN) is set to zero |
| `data.end();` | Returns the number of elements that contain Datum objects that have been added to the vector. |
| `data = datum;` | A unique datum is inserted into the sorted array at the correct position. The ">=" and "==" operators in the Datum class must be defined |
| `data += datum;` | Datum is appended to array (at new last element) |
| `Datum& d = data.nextData();` `data.nextData() = datum;` | A reference to new last element of array is returned. It may used as shown or immediately with a dot operator or even as a target of an assignment (where a Datum operator= is defined) |
| `data(i, datum);` | Datum is inserted at index i, the contents at i and above are moved up one element |
| `Datum* d = find(key);` | Perform a linear search of the Expandable object for a datum with a component with the same value as key. |
| `Datum* d = bSearch(key);` | Perform a binary search of the Expandable object for a datum with a component with the same value as key. |
| `data.del(i);` | The datum at index i is deleted and the elements above are moved down to fill in the hole. The number of elements in the array is reduced by one |

## Cautions

C++ allows the address of an array (i.e. vector) to be placed in a pointer defined by the class that inhabits the vector: `Datum* p = &data[0]` or `Datum* p = data`. There is nothing wrong with using a pointer as illustrated, however, incrementing the pointer indiscriminately may result in a memory violation.

Expandable copies the old vector into a new vector when it needs to expand. So getting a pointer or reference to a Datum object in the vector is only good until the next addition to the array. Most of the time the pointer will be valid but there will be that one time when holding a pointer while adding an entry into the array and the application will crash with an exception. The code will look good and it will not work and it will be hard to debug. So the rule is: Pointers/references to Expandable element objects are only good if there are no insertions while the pointer/reference is active, period!

The pointer problem mentioned above is solved by ExpandableP with some additional memory overhead. See the description of ExpandableP below.

## Iterator

The typical loop uses an index and an array reference using the index:

```
    for (i = 0; i < data.end(); i++) {
    Datum& dtm = data[i];
    o o o
```

Another method for performing a loop uses an iterator. An iterator template is defined for the class in which a single Expandable(P) object is defined. If the Iterator typedef for data is StoreIter then this is how it would be used:

```
void Store::display2() {
StoreIter iter(*this);
Datum*    dtm;
String    s;

  for (dtm = iter(); dtm; dtm = iter++) {
    s.format(_T("%3i/%5i: "), iter.index(),  dtm->key);

    wcout << s.str() << dtm->line.str();
    }
  }
```

### Iterator Declaration

The iterator for a class holding an Expandable object is specialized for the class. This means that the IterT is a class template. Furthermore, we may need two or more iterators in the same function for the same Expandable(p) object so the template defines a typedef. Here is how the Iterator for Store is defined:

```
class Store;
typedef IterT<Store, Datum> StoreIter;
```

These declarations are placed after the Datum class and before the Store class. When the iterator is to be used it must be defined with the name of an object of class Store (note: capitalized). In the example there is one Store object and it is called "store" (note, not capitalized). Here are the two declarations needed to use the iterator:

```
StoreIter iter(store);
Datum*    dtm;
```

Of course within a Store function, the Store object may be represented as `*this`.

So an iterator does not operate in a vacuum. The Store class requires a few little functions and a friend declaration. Here they are:

```
class Store {

    o o o

private:

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

  Datum* datum(int i) {return 0 <= i && i < nData() ? &data[i] : 0;}       // or data[i].p

  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 StoreIter;
```

The IterT.h header file contains a comment in which these functions may be found. The names of the Expandable(P) object and the Datum object will need to be changed.

### Iterator Functions

| | |
|---|---|
| `StoreIter iter(store);` `Datum*    dtm;` | Initialization Constructor which names the object in which the Expandable object resides |
| `StoreIter jter(iter);` | Copy constructor which initializes jter to the current state of iter. |
| `dtm = iter(Fwd) or iter();` `dtm = iter(Rev);` | Initialize iterator in the forward (i.e. Fwd) or reverse (i.e. Rev) direction, defaults to Fwd. Returns a p[pointer the first element of the scan or zero. |
| `dtm = iter++;` | Increments the scan index if less than the end of the vector and returns a pointer to the next element of the vector or zero. |
| `dtm = iter--;` | Decrements the scan index if greater than zero and returns a pointer to the next element of the vector or zero. |
| `int index = iter.index();` | returns the current index in the iterator, it may not be a legal index |
| `dtm = iter. current();` | Returns the current element of the vector or zero |
| `if (iter.isLast()) ...` | Returns true if the current element is at the largest index containing elements |
| `if (iter.isFirst()) ...` | Returns true if current element is at the zero index |
| `iter.remove(Fwd) or remove();` `iter.remove(Rev);` | Remove the current element from the vector and adjust the iterator index according to the direction. |

## Linear and Binary Searches

The Expandable(P) function "find" performs a linear search for a key and "bSearch" performs a binary search for a key. Actually, they are both templates that allow any key type to be used. Of course they both use boolean operations on the Datum object that must be defined. They return a pointer to a Datum object of zero.

```
template<class Key>
Data* find(Key key) {
  o o o
  }

template<class Key>
Data* bSearch(Key key) {
  o o o
  }
```

Here are the boolean operations that must appear in the Datum object for the two searches (only equaility is needed for the linear search):

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

## ExpandableP

When the class used in an Expandable(P) vector is large the cost of expanding the vector becomes fairly large. For example:

```
class Words {
public:

String zero;
String one;
String two;
String three;
String rest;

  Words() { }
  Words(Words& wrds) {copy(wrds);}
 ~Words() { }

  Words& operator= (Words& wrds) {copy(wrds); return *this;}

  void load(String& line);

  void display();

private:

  void copy(Words& wrds);

  bool nextWord(String& s, String& word);
  };
```

For this example I intend to trim the line of leading and trailing spaces, tabs and end of line characters. Then when loading the file the first four words are placed in the Strings zero, one, two and three with the rest of of the line in the String rest.

ExpandableP solves the expensive copy issue at the cost on one pointer for every object in the array. On the plus side, since the pointers are moved (and not the Datum objects) the getting a pointer to an object in the ExpandableP vector is permanent until the element is deleted. That is, no worries about possessing a `Datum* ptr` and inserting new elements anywhere in the ExpandableP vector.

It is used in almost the same manner as Expandable but has some additional functions. The declaration of the ExpandableP object is a bit more complex. The RcdPtrT template provides all the operations needed by ExpandableP to manage the Words objects in the vector.

```
typedef RcdPtrT<Words> WordsP;


class StoreP {

ExpandableP<Words, WordsP, 2> data;

public:
  o o o
```

The iterator is defined in the same way for an ExpandableP object but the private datum function is defined a bit differently:

```
class StoreP;
typedef IterT<StoreP, Words> StorePIter;

class StoreP {

ExpandableP<Words, WordsP, 2> data;
  o o o
private:

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

  Words* datum(int i) {return 0 <= i && i < nData() ? data[i].p : 0;}       // note: data[i].p
```

With Expandable, datum includes the expression `&data[i]`. ExpandableP includes the expression `data[i].p`.

## ExpandableP Operations

| | |
|---|---|
| `datum = data[i];` | where 0 <= i < data.end() (i.e. endN) |
| `data[i] = datum;` | array expands to encompass i |
| `data.clear();` | content is ignored but number of elements is set to zero |
| `data.end();` | returns the number of elements that contain Datum objects that have been added to the vector. |

| | |
|---|---|
| `data = datum;` `data = &datum;` | A unique datum is copied into an allocated heap node and then inserted into the sorted vector at the correct position. If the datum already exists in the vector then the function merely returns. The ">=" and "==" operators in datum must be defined |
| `data += &datum;` | A pointer to a datum is considered to be an already allocated (see allocate() below) object and is appended to the array |
| `data += datum;` | A object is assumed to be a local variable and not already allocated in the heap. So a record is allocated and datum is copied to the new record. The new record is appended to array (at new last element) |
| `Datum& d = data.nextData();` `data.nextData() = datum;` | A new record is allocated and stored in the new last element of array. A reference to the new record is returned. It may used as shown or immediately with a dot operator or even as a target of an assignment (where a Datum operator= is defined) |
| `Datum& d = data.getData(i);` | return a reference to a record at index i allocating a record if necessary |
| `data(i, datum);` | datum is inserted at index i, the contents at i and above are moved up one element. A new record is allocated and datum copied into the new record. |
| `data(i, &datum);` | The pointer indicates that the record has already been allocated and it is inserted at index i, the contents at i and above being moved up one element. |
| `Datum* d = find(key);` | Perform a linear search of the ExpandableP object for a datum with a component with the same value as key. |
| `Datum* d = bSearch(key);` | Perform a binary search of the ExpandableP object for a datum with a component with the same value as key. |
| `data.del(i);` | The datum at index i is deleted and the elements above are moved down to fill in the hole. The number of elements in the array is reduced by one |
| `Datum* d = data.allocate();` | Allocate one record and return a pointer to it |
| `data.deallocate(&datum);` | Deallocate one record |
| `RcdPtr* rcdP = data.getRcdPtr(i);` | Returns a pointer to a RcdPtr class. This should be used sparingly if at all. I used it once to deal with deallocating a case where the pointer in RcdPtr contained a base pointer with two variants. Thus, the standard deallocation scheme failed to release all the memory since it only know about the base class |

Since all the Words objects are stored in heap nodes, there are some additional functions to allocate, add and deallocate Words nodes. This gives the programmer some additional flexibility with respect to creating nodes. It should also be noted that Words constructors are executed during creation of the node. When the node is contained in the vector and the entire ExpandableP vector is deleted and/or an element of the vector is deleted the corresponding destructors are called. The object itself is responsible for construction and destructing all of its components. Note, the String component has its own constructor and destructor. Simple integer, doubles, etc. do not require constructor and destructors. If there is allocated heap objects in the Datum object the the Datum destructor is responsible for deallocating the heap objects.

## Conclusion

When I was studying Computer Science a long time ago we created elaborate data structures to optimize time and space. The more complex data structures were a marvel, ... but with complexity comes difficulty getting them right all the time. I once spent about a week looking for a bug that would occur after executing the code for 45 minutes. The code was large and complex and had been modified by someone in the previous year or so. The fix: move two statements from one class to two daughter classes. The moral of this story is keep the code simple.

Expandable(P) grew out of a simple hash table which when it was around 80% full would be increased in size. It was implemented in a vector similar to an Expandable vector. In that case the move was done by rehashing each entry into the new space. The trick was that the heap was managed by the application so the hash table was increased by merely adding a chunk of memory to the existing table. The rehash took an entry to the same place or somewhere else in the expanded table.

The libraries included with Visual Studio 2017 provide some storage classes and I have used one in an important application. It allows the data store to be defined, provides an iterator or other functions. It might be implemented as a tree too. Do I trust it, yes. It just seemed clumsy to me. It returned a tuple sometimes, which I had to translate into something useful in the program. Did I use it again in another application, nope. I reinvented the wheel that I perceived to be simpler and provided just the services that I needed.