The **`vector`**

The **`vector`** is a type-safe, sequential container class that behaves like an array. You can set the size of the **`vector`** up front, you can use **`operator[]`** to access and modify individual entries, and you can splice new elements in anywhere you want and let the **`vector`** do all of the shifting for you. The primary win of the **`vector`** over the array comes from its ability to grow and shrink automatically in response to insertion and deletion. Because arrays are useful data structures all by themselves, and because **`vector`**s are designed to not only behave like arrays but to interact with the programmer using the same syntax, **`vector`**s are used more often than any other STL class.

Programs making use of the **`vector`** first need to **`#include`** some boilerplate at the top of any file making use of it:

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

The most commonly used **`vector`** operations are summarized in the abbreviated class definition presented here:

```
template <class T>
class vector {
   public:
      vector();
      vector(const vector<T>& originalMap);

      typedef implementation_specific_class_1 iterator;
      typedef implementation_specific_class_2 const_iterator;

      bool empty() const;         // true iff logical length is 0
      long size() const;          // returns logical length of vector
      void clear();               // empties the vector, sets size to 0

      void push_back(const T& elem);
      void pop_back();

      T& operator[](int i);
      const T& operator[](int i) const;
      iterator insert(iterator where, const T& elem);
      iterator erase(iterator where);

      iterator begin();
      iterator end();
      const_iterator begin() const;
      const_iterator end() const;
};
```

Here's a simple function that populates an empty **`vector`** with the lines of a file:

```
static void readFile(ifstream& infile, vector<string>& lines)
```

```
{
   assert(lines.size() == 0);    // assert aborts program if test fails
   assert(infile.good());        // verify ifstream refers to legit file

   string line;
   while (ifstream.peek() != EOF) {
      getline(ifstream, line);   // reassign line to be next line of file
      lines.push_back(line);     // append
   }

   cout << "All done! (Number of lines: " << lines.size() << ")" << endl;
}
```

push_back tacks something new to the end of the **vector**. Whenever the argument to **push_back** is a direct object (as opposed to a pointer), the **vector** makes a deep, independent copy of that object. One can append pointers as well, but expect the pointer and nothing more to be replicated behind the scenes; any memory referenced by that pointer will be referenced from within the **vector** as well.

Traversing the **vector**'s elements is trivial. You iterate over the **vector** using the same semantics normally used to traverse traditional arrays.

```
vector<double> transactionAmounts;
// initialization code omitted

double totalSales = 0.0;
for (int i = 0; i < transactionAmounts.size(); i++)
   totalSales += transactionAmounts[i];
```

While it may not be obvious, **operator[]** is called repeatedly, each call returning some dollar and cent amount to be added to the running total.

The **vector** specification exports **begin** and **end** methods—routines producing start the past-the-end iterators. Iterators behave like pointers—in fact, they are often defined to be actual pointers when the encapsulated elements really are laid out sequentially in memory. The STL's intent is to provide an iterator with each and every container type it defines, making it the responsibility of the iterator to mimic the behavior of true pointers while providing sequential access to all contained elements.

Some programmers prefer the iterator over the traditional array-indexing idea, even when using the **vector**.

```
vector<double> transactionAmounts;
// initialization code omitted

double totalSales = 0.0;
vector<double>::const_iterator curr = transactionAmounts.begin();
vector<double>::const_iterator end = transactionAmouts.end();
for (; curr != end; ++curr)
   totalSales += *curr;
```

A more interesting example:

```
struct flight {
   char flightNum[8];   // embedded C-string, e.g "USA177"
   string origin;       // leaving San Francisco
   string destination;  // arriving Puerto Vallarta
   short firstClass;    // number of passengers flying first class
   short coach;         // number of passengers flying coach
};
```

Pretend that USAirways needs to cancel any and all undersold flights. Functionality designed to filter a **`vector`** of flight records to remove such flights might look like this:

```
void cancelLowCapacityFlights(vector<flight>& flights, int minPassengers)
{
   vector<flight>::iterator curr = flights.begin();
   while (curr != flights.end()) {
      if (curr->firstClass + curr->coach < minPassengers)
         curr = flights.erase(curr);
      else
         ++curr;
   }
}
```

Each iteration inspects a flight, and in the process decides whether or not to cancel. When the capacity requirement is met, our job is easy: we leave the flight alone and advance the iterator to the next flight in the sequence. Otherwise, we rely on **`erase`** to splice out the flight addressed by **`curr`**. Changes to a **`vector`**—and you certainly get changes when you **`erase`** an element—invalidate all iterators. The **`vector`** should have the flexibility to resize, compact, and/or relocate memory behind the scenes (and you know what types of things might happen behind the abstraction wall, so you shouldn't be surprised.). As a result, previously generated iterators could reference meaningless data. **`erase`** supplies a new iterator identifying the element that **would** have been next had we not changed anything. In this case there's no reason to manually advance the iterator, since **`erase`** effectively does that for us.

If you understand why each call to **`erase`** necessarily invalidates existing iterators, then you'll also understand why the **`flights.end()`** resides within the test of the **`while`** loop, thereby requiring it to be called with every iteration.

The `map`

The **`map`** is the STL's generic symbol table, and it allows you to specify the data type for both the key and the value. The boilerplate required to use the STL **`map`** is:

```
#include <map>
using namespace std;
```

The most commonly used **`map`** constructors and methods are summarized here:

```
template <class Key, class Value>
class map {
   public:
      map();
      map(const map<Key, Value>& originalMap);

      // typedefs for iterator and const_iterator

      pair<iterator, bool> insert(const pair<Key, Value>& newEntry);
      iterator find(const Key& key);
      const_iterator find(const Key& key) const;

      Value& operator[](const Key& key);

      iterator begin();
      iterator end();
      const_iterator begin() const;
      const_iterator end() const;
};
```

While there are more methods that those listed above—many more, in fact—these operations are the most common. For the full story, you should search either of the two web sites mentioned on page one.

Let's build a **`map`** of some of my favorite chords. I'll start out small.

```
typedef map<string, vector<string> > chordMap;
chordMap jazzChords;

vector<string> cmajor;
cmajor.push_back("C");
cmajor.push_back("E");
cmajor.push_back("G");

pair<chordMap::iterator, bool> result =
   jazzChords.insert(make_pair(string("C Major"), cmajor));
```

The **`insert`** method is stubborn, because it doesn't permit existing keys to be reassigned to new values—that is, had **`"C major"`** been previously inserted, **`insert`** would leave the original entry intact and report the fact that no changes were made back to the client via the returned pair. This isn't a limitation of the **`map`**, as there are other ways to **erase** and/or otherwise modify an existing key-value pair. **insert** just isn't one of them.

The return value (which is often ignored if we know the key isn't in the **map** prior to the **insert** call) returns a **pair**. The **second** field of the return value reports whether or not the key is already bound to some other value; **true** means that the insertion really did something—that the key is new to the **map**, the insertion really modified the structure, and that the insertion increased the key count by one. **false** means that the key is already present and that no changes were made. The **first** field of the **pair** stores an **iterator** addressing the **pair** stored inside the map on behalf of the key. Regardless of whether **insert** added the key or not, we're guaranteed to have some key-value pair after the call.

Code to examine the result of the **insert** call could be used as follows:

```
pair<chordMap::iterator, bool> result =
   jazzChords.insert(make_pair(string("C Major"), cmajor));
if (result.second)
   cout << "\"" << result.first->first << "\" was successfully inserted." << endl;
else
   cout << "\"" << result.first->first << "\" already present." << endl;
```

If you know there's zero chance the key was already in there, you could ignore the return value and proceed without checking it. If nothing else, however, you might include a test asserting that the insertion was successful:

```
pair<chordMap::iterator, bool> result =
   jazzChords.insert(string("C Major"), cmajor);
if (!result.second) {
   cerr << "ERROR: \"" << result.first->first << "\" already present!";
   cerr << end;
   exit(1);
}
```

The **find** method isn't quite as complicated. There are really three overloaded versions of **find**, but I'm commenting only on the one I expect you'll be using, at least for CS107 purposes. Curious minds are welcome to cruise the **dinkumware** to read up on the full suite of **find** operations.

If you're not sure whether or not some key of interest resides within your map, but you need to know, then **find** is the method for you. You wake up in the middle of the night, sweating, panicky, and nauseous because you forget how to play an F# minor 11 chord, but you calm down once you remember that **find** can tell you anything about any chord ever invented.

```
chordMap::const_iterator found = jazzChords.find("F# minor 11");[1]
if (found == jazzChords.end())
   cout << "Chord was never recorded." << endl;
else {
   cout << "Notes of the " << found->first << " chord:" << endl;
   for (vector<string>::const_iterator note = found->second.begin();
        note != found->second.end(); ++note)
      cout << "\t" << *note << endl;
}
```

The nature of the **find** call itself shouldn't surprise you. The comparison of the return value to **jazzChords.end()** might. We're used to sentinel values of **NULL** and –1, but the STL has each instance of each container define its own sentinel value, and that value is categorically reported by the container's **end** method. The check against **end()** should generally be made—particularly if you're not sure the key is even present.

Finally, the **operator[]** method allows programmer to access and even update a **map** using array-like semantics. The **portfolio** example you saw earlier:

```
map<string, int> portfolio;

portfolio.insert(make_pair(string("LU"), 400));
portfolio.insert(make_pair(string("AAPL"), 80));
portfolio.insert(make_pair(string("GOOG"), 6500));
```

could be rewritten to make use of **operator[]** instead:

```
map<string, int> portfolio;

portfolio["LU"] = 400;
portfolio["AAPL"] = 80;
portfolio["GOOG"] = 6500;
```

You'll notice from the prototype for **operator[]** returns a **Value&**; in a nutshell, it returns an automatically dereferenced pointer (we know such things as references) into the space within a **map** that stores a value of some key-value pair. The beauty here is that **operator[]** allows us to update an existing value to something new—something the **insert** operation doesn't allow. The purchase of 300 additional shares of Lucent could be encoded by the following:

```
portfolio["LU"] += 300;
```

What's important to understand here is that the **+=** doesn't act on **myStockPortfolio**, but rather on the **int&** returned by **operator[]**.

[1] Because **jazzChords** is non-**const**, the non-**const** version of **find** will always be called. We expect an **iterator**, not a **const_iterator**, to be returned. However, **found** doesn't require anything but read access to the map entry **iterator** produce by **find**, so in the interest of safety and good style, we declare **found** to be a **const_iterator**.

Incidentally…

All large data structures—whether they come in the form of a record, a client-defined class, or an STL container—should typically be passed around either by address or by reference. C++ purists tend to pass large structures around by reference, because to pass a structure around by value is to invoke its copy constructor: the constructor that understands how to initialize an object to be a clone on an existing one. Passing around by address also avoids the copy, but those passing by reference enjoy the convenience of direct object semantics (**.** is prettier than ***** and **->**). In general, properly defined copy constructors take a noticeable amount of time to run, and improperly implemented ones can unintentionally share information (via embedded pointers, for example) between the original and the copy.

Iterating Over The Map

A function designed to sell half of any stock you feel you own too much of might be implemented this way:

```
void sellStocks(map<string, int>& stocks, int threshold)
{
   map<string, int>::iterator curr = stocks.begin();
   while (curr != stocks.end()) {
      if (curr->second >= threshold) curr->second /= 2;
      ++curr;
   }
}
```

A function that counts the total number of shares you own would need to traverse the entirety of the **map** in a similar way.

```
int countStocks(const map<string, int>& stocks)
{
   int stockCount = 0;
   map<string, int>::const_iterator curr = stocks.begin();
   while (curr != stocks.end()) {
      stockCount += curr->second;
      ++curr;
   }
   return stockCount;
}
```

Notice that this last function only requires **const** access to the **map**. Therefore, the reference passed is marked as **const**, and the local iterators used to examine the collection of stock items are of type **const_iterator**. Notice that a **const_iterator** itself isn't frozen—it responds to **operator++** just fine. A **const_iterator** merely respect the **const**ness of whatever it's addressing.