CS106B
J Zelenski

Handout #14
Jan 16, 2008

The fun life of CS106 class client

Our CS106 library contains a set of classes that model the classic CS data structures—vectors, stacks, sets, maps, and more. We will spend quite a bit of time this quarter exploring these classes and considering them from two perspectives. Initially, we will interact with the classes solely in the position of a client. Later in the quarter, we will revisit the classes from an implementer's perspective and work through different implementation options and their tradeoffs. This handout provides information on the client-side use of the classes to share with you on what's available and how to put it to work. The later chapters of the reader will dig into the internals of the classes and explore various implementation strategies.

Objects and classes

From your CS106A (or equivalent) experience, we are expecting that you are familiar with using classes. Object-oriented programming in C++ is quite similar in concept to Java, but there are some various syntactic differences. Reader section 8.1 gives an overview of object-oriented programming in general and our earlier "Getting started in C++" handout notes how the features are expressed in C++. You already have seen using C++ objects— the string and stream classes—so by now you're on your way to becoming an old pro!

One key difference between Java and C++ objects is that Java creates all objects in the heap and accesses them via pointers (although they are called "references" and don't explicitly mention `*`, they are pointers, nonetheless). In C++, you have the option of allocating objects on the stack or in the heap, but by default we use stack allocation as it is convenient and efficient.

One feature of Java's "objects-are-always-pointers" strategy is that you are automatically always passing/assigning pointers, which is more efficient than making full object copies. In C++, we can achieve similar efficiency by taking care to pass objects by reference whenever possible and only using assignment between objects when a full copy is truly needed. It's a good idea to get used to these habits now since they are endemic among C++ programmers, and, of course, you want to fit in at all the hippest parties.

CS106 class library

The classes in our library are the **`Scanner`**, **`Vector`**, **`Grid`**, **`Stack`**, **`Queue`**, **`Set`**, and **`Map`**.[1] This handout provides an introduction to these classes with a listing of the class interfaces and some sample client code for each.

The CS106 classes represent many of the classic, fundamental data structures of computer science and you'll soon discover that they provide tremendously useful functionality. It is a real delight to use these classes as a client. Without any concern for how it works internally, you will be able to create a set, for example, and use its member functions to add and remove elements or intersect with other sets and so on.

There is a huge amount of leverage gained by having such commonly needed functionality in a shared library. For starters, the range of problems that you can attempt to solve is vastly enlarged by having better tools. My colleague Nick Parlante likes to describe this as "living higher on the food chain"—you can focus solving more interesting problems when you're not bogged down in the

[1] The official C++ Standard Template Library (STL) contains classes similar in spirit to those in our library. As much I love to teach to standards, the design of the STL is hairy enough that I believe it would interfere with our pedagogical goals. We may spend a little time at the end of the quarter on the STL, but only after you have experience using our kinder, gentler version. The optional lab CS106L will discuss the STL if you want to drop in to become acquainted with it.

petty details of directly managing an array and so on. The classes are very efficiently implemented, by using advanced techniques we will later learn, but in the meantime, you are still able to take advantage of that streamlined performance by using the classes, without having to know how they work. By having each class model a clear abstraction (such as the waiting line analog for a queue), you can choose the right data model for your program and naturally express your computation in terms of member function calls on that object, which leads to clear and clean code.

I just can't speak highly enough of how much more reliable, efficient, and just plain fun coding becomes when you are able to work as a client of well-written libraries, so let's get started!

Scanner

The **`Scanner`** class provides functionality for dividing a string into separate words or *tokens*. Although the standard stream extraction >> has some features that could be used for this, that approach is somewhat primitive and inflexible and involves some tedious coding. A scanner is designed to simply the tasks of tokenization and comes in quite handy when doing any sort of string or file processing. Here are a few ideas to get you thinking about its uses:

- dividing a document into component words for textual analysis, such as a word frequency count, preparing an index, or building a concordance
- separating a file pathname into its components (e.g. the sub-directories within HardDisk/Classes/CS106/Handouts/H36 Section 5.doc)
- parsing a user's command (e.g. "turn left", "find price < 5") in order to act upon the request
- evaluating a arithmetic expression (3 + 4 * 7)

The basic public interface of the **`Scanner`** class is listed below:

```
class Scanner {
    public:
        Scanner();       // constructor (invoked when allocated)
        ~Scanner();      // destructor (invoked when deallocated)

        void setInput(string str); // set string to be scanned

        string nextToken();
        bool hasMoreTokens();

        void saveToken(string token);

        enum spaceOptionT { PreserveSpaces, IgnoreSpaces };

        void setSpaceOption(spaceOptionT option);
        spaceOptionT getSpaceOption();

       // other advanced options excerpted for clarity
};
```

The idiomatic pattern for using a scanner is to create a new object, set the input string to be scanned, and then enter a loop that calls **`nextToken`** while **`hasMoreTokens`** returns true. Here is some sample code that uses the scanner to reports the number of tokens entered in a user's response:

```
void CountTokens()
{
      Scanner scanner;
      cout << "Please enter a sentence: ";
      scanner.setInput(GetLine());
      int count = 0;
      while (scanner.hasMoreTokens()) {
         scanner.nextToken();
         count++;
      }
      cout << "You entered " << count << " tokens." << endl;
}
```

The next bit of code shows finding the longest token read from a file. In the outer loop, `getline` read a line from the file and in the inner loop, the line is tokenized by a scanner.

```
// Given an opened input stream, reads contents line by line,
// uses scanner to break line into tokens and tracks the
// longest token seen, which is returned from the function
string FindLongestToken(ifstream & in)
{
   string longest = "";
   Scanner scanner;

   while (true) {       // outer loop reads file line-by-line
      string line;
      getline(in, line);
      if (in.fail()) break; // no more lines to read
      scanner.setInput(line);
      while (scanner.hasMoreTokens()) {    // inner loop tokenizes a line
         string token = scanner.nextToken();
         if (token.length() > longest.length())
            longest = token;
      }
   }
   return longest;
}
```

There are various scanner options that control how certain inputs are scanned. Most of these options are only used in limited situations, but one option that is commonly manipulated is the `SpaceOption`. This options controls whether whitespace tokens are returned from `nextToken` or skipped over and ignored. The default is for every token, including spaces, to be returned, but you can cause them to be discarded by changing the option:

```
scanner.setSpaceOption(Scanner::IgnoreSpaces);
```

This is handy when you want to ignore spaces and only process non-space tokens. Note that the `spaceOptionT` enum is defined within the Scanner class, so when referring to those values as a client, you must fully specify the name including the scope qualification `Scanner::`. This tells the compiler to look for the name within the Scanner class instead of assuming it is top-level entity. This is like the `string::npos` constant.

You can read more about the `Scanner` class and its details in section 8.6 of the reader. The full `scanner.h` file with extensive comments is listed in the reader appendix. The `Scanner` class interface is also browsable from the "Documentation" link on our class web site.

Class templates

Most of the classes in the CS106 class library are *container* classes. Container classes are used to store data, such a list of students enrolled in a class, a table of dorm room assignments, or a set of skills that a job applicant has. Pretty much all programs store data and often the ways they need to organize and access it fits into standard patterns. Thus, a few well-designed container classes can go a long way toward meeting many common data storage needs.

All of our container classes are provided as *class templates*. A class template means that the class is defined in terms of a "placeholder" for the element type being stored. Rather than defining the container to store only students or only numbers, the placeholder allows the template to be used to store any kind of element the client might choose. The client fills in the placeholder when declaring and allocating the container object. The client can thus create a set of numbers, a set of strings, or a set of attributes, as needed.

Mostly all this means to you as a client is that you can use a container to store whatever you need. You will need to annotate the container class name with the element type you wish to store in angle-brackets when declaring and allocating the container object, e.g. `Vector<int>` or `Vector<string>`, which is a little bit clunky, but a small price to pay for such flexibility!

Templates are a marvelous C++ feature that supports *generic programming*. No one wants to write a whole gob of duplicate vector classes, one that holds ints, another for strings, and yet another for students. The template is a generic form capable of being specialized on demand. The template code is written, tested, optimized, debugged, and commented just once, yet can be used for a wide variety of needs. And C++ templates are completely type-safe— the compiler keeps it straight and will not confuse a vector of floats with one holding bools. However, these benefits come with a little bit of goop, most notably that the syntax can be cumbersome and that the compiler errors reported when you've made a mistake using a template can sometimes be confusing. Be sure to take advantage of the wisdom of our course staff and visit us in the LaIR or send an email if you need some help deciphering one of these cryptic error messages.

Vector

The `Vector` class is the simplest of all the container classes. The abstraction it provides is a linear, indexed homogeneous collection. (Think Java's ArrayList) A vector serves the same basic purpose as the built-in C++ array. What makes the vector better than the raw array?

- A vector knows its size
- Access to elements is bounds checked
- The vector handles all memory-management (shrink/grow as elements added/removed)
- Insert and remove handle shuffling elements up and down to open/close up space
- It supports deep-copying, when you need to clone the collection

In my opinion, once you have the `Vector` class in your toolkit, there is little reason to bother with raw array, since a vector is a better version of the same thing.

The public member functions of the `Vector` class are listed below. Note that it is a class template, so the arguments and return type for the member functions refer to the placeholder `ElemType` that is filled by the client when declaring and allocating a vector. The excerpt below just lists the prototypes, the full `vector.h` interface includes comments describing the use and operations.

```
template <typename ElemType>
 class Vector {

   public:
      Vector();
      ~Vector();

      int size();
      bool isEmpty();

      ElemType getAt(int index);
      void setAt(int index, ElemType value);

      void add(ElemType value);
      void insertAt(int pos, ElemType value);
      void removeAt(int pos);
};
```

Vector also implements a few overloaded operators for convenience. The square brackets can be applied to a vector to access an individual element by index. The class also provides deep-copying support so that assigning one vector to another or passing a vector by value will make a full, distinct copy of the entire vector contents. Copying can be expensive, especially for a large vector, so you typically avoid copying whenever possible, but when you do need a copy, the class makes it easy to get one.

The Vector is a great tool for any homogenous collection. Here's a sample function that reads scores into a vector and prints the sequence:

```
 // Fills vector with a sequence of integers read from console
 void ReadScoresFromUser(Vector<int> & v) // pass-by-ref since updating v
 {
    while (true) {
       cout << "Enter number (-1 if done): ";
       int num = GetInteger();
       if (num == -1) break;
       v.add(num);             // append to end of vector
    }
 }

// Prints contents of vector, one number per line
 void PrintVector(Vector<int> & v)       // pass-by-ref for efficiency
 {
    for (int i = 0; i < v.size(); i++)
       cout << v[i] << endl;   // or equivalent v.getAt(i)
 }

 int main()
 {
    Vector<int> scores;

    ReadScoresFromUser(scores);
    PrintVector(scores);
    return 0;
 }
```

Instead of adding each value to the end of the vector, you could use the **insertAt** member function to store the values in ascending order, with this small change to the above code:

```
    // use these lines instead of v.add(num) in ReadScores function

        int pos = 0; // find correct pos to insert num in sorted order
        while (pos < v.size() && num > v[pos])
             pos++;
        v.insertAt(pos, num);
```

Note that all the bothersome details of allocated extra capacity and shuffling elements over to make space for the inserted one are conveniently handled by the **insertAt** member function for us!

This final vector example shows reading the words from a file and storing them in a vector of lines, where each line is actually a vector of the tokens in that line. Wow, a vector containing vectors!

```
int LongestLineCount(ifstream & in)
{
   Vector<Vector<string> > lines; // note space between > >
   Scanner scanner;
   scanner.setSpaceOption(Scanner::IgnoreSpaces); // discard white

   while (true) {
      string line;
      getline(in, line);
      if (in.fail()) break; // no more lines to read
      scanner.setInput(line);
      Vector<string> tokens;
      while (scanner.hasMoreTokens())
         tokens.add(scanner.nextToken());
      lines.add(tokens);
   }

   // this code searches to find line with most tokens
   Vector<string> max = lines[0];
   for (int i = 1; i < lines.size(); i++) {
      if (lines[i].size() > max.size())
         max = lines[i];
   }
   return max.size(); // return # of tokens in longest line
}
```

In the declaration for the nested template, you'll note that there is a space character deliberately inserted between the closing angle bracket for the inner template name and the closer for the outer. Without that space, the compiler misidentifies the **>>** sequence as the right-shift (or stream extraction) operator. Luckily, most compilers produce a pretty reasonable error message if you forget the space (Xcode says **'>>' should be '> >' within a nested template argument list**") so hopefully you'll quickly learn to avoid this little pitfall.[1]

By the way, you can use **typedef** to introduce a shorthand nickname for any C++ type. That nickname can then used interchangeably with the full name. Nicknames are particularly handy for the cumbersome specialized template names, such as the examples shown below:

```
typedef Vector<binkyT> BinkyVector;
typedef Vector<Vector<string> > VecVecString;
```

[1] There are plans to address this in the future revision of C++, so in some time in this millennium you can hope that the extra space will not longer be necessary.

Despite its incredible utility, the **`Vector`** class is not discussed much in the reader (left as an exercise in Chapter 10). The full **`vector.h`** file with comments appears in the reader appendix. The class interface is also browsable from the "Documentation" link on our class web site.

Grid

The **`Grid`** class stores a rectangular two-dimensional indexed grid of homogeneous elements. The class is designed for the simple case where the grid is rectangular (i.e. all rows have the same number of columns) and the number of rows and columns stays fixed. This is the most common use of 2-dimensional arrays and grid uses a very simple design to address this basic need. It is possible to build more flexible two-dimensional data structures using vectors containing vectors (as shown in the last vector example above). That strategy requires a bit more hands-on management but allows for unequal sizes across the dimensions as well as changing the dimensions midstream.

Features to appreciate about the grid class:

- A grid knows its dimensions
- Access is bounds-checked in both dimensions
- Handles all details of memory allocation/deallocation
- Provides deep-copying support

The public member functions of the **`Grid`** class are listed below, in template form. The full **`grid.h`** interface contains comments on the features.

```
template <typename ElemType>
 class Grid {

   public:
      Grid();
      Grid(int numRows, int numCols); // constructor is overloaded
      ~Grid();

      int numRows();
      int numCols();

      ElemType getAt(int row, int col);
      void setAt(int row, int col, ElemType value);

      void resize(int numRows, int numCols);
};
```

The value at a particular row,col location in the grid can be retrieved/changed via the getAt/setAt member functions. Locations can also be accessed using a shorthand notation that overloads the parentheses operator i.e. **`grid(row, col)`**.[1]

Note that grid and vector have a different design with regard to size/capacity. The vector starts empty and its size changes dynamically as elements are added and removed. The grid, on the other hand, is sized to the dimensions chosen when declaring a grid (using the two-argument form of the constructor) or using the resize member function. If a grid has been sized to 3 rows and 4 columns, it will always contain exactly 12 elements, one for each location in the grid. Before a grid element has been assigned, its value is the default value for its type, i.e. for strings, the default value is empty string, for ints, it is uninitialized.

Just like Vector, the Grid class provides deep-copying support, and we typically use pass by

[1] There is a slightly obscure technical detail why it does not use the expected double square bracket **`grid[row][col]`**, come ask me why if you're curious.

reference to avoid making expensive copies when not explicitly needed.
You would find a grid handy for:

- managing a rectangular game board (chess, scrabble, battleship, connect four, sudoku, go, …)
- maintaining a weekly calendar, where each day has slots for each hour
- storing a mathematical matrix
- representing the pixels in an image
- handling the contents of an HTML table

Here is some sample code that initializes a new grid object to serve as a tic-tac-toe board:

```
// Returns a new 3x3 grid of chars, where each
// elem is initialized to space character
Grid<char> CreateEmptyBoard()
{
   Grid<char> board(3, 3); // create 3x3 board of chars

   for (int row = 0; row < board.numRows(); row++)
      for (int col = 0; col < board.numCols(); col++)
          board(row, col) = ' '; // assign space char to elem
         // equivalent to board.setAt(row, col, ' ')
   return board;  // btw, it's ok to return object
}
```

Like `Vector`, the `Grid` class doesn't make much of an appearance in the reader text, but the commented `grid.h` file is given in the reader appendix. The class interface is also browsable from the "Documentation" link on our class web site.

Stack

The `Stack` is another simple container class, in fact, even simpler than vector. A stack is like a vector that allows adding and removing elements only from one end, the *top* of the stack. A stack has a real-world analog in a stack of plates or pancakes. When a new item is added to a stack, it is placed on top of any others and when removing an item, it is the topmost item that is removed. No reaching into the middle of the stack is allowed. Because the first item removed is always the last one added, a stack is said to operate in *last-in-first-out* fashion, commonly abbreviated LIFO. Adding an item to a stack is called *pushing* onto the stack and removing the topmost is called *popping* from the stack. The corresponding member functions are thus named `push` and `pop`.

What sort of things might you find a stack useful for?

- reversing data (push all then pop until empty returns data in reverse order)
- managing a sequence of user actions or game moves that a user can undo
- temporary shadowing (pushed element shadows those further down the stack)
- tracking the history of visited pages during web browsing
- simulating the mechanics of nested function calls
- implementing an RPN calculator

The stack might be seen as redundant with vector, since anything you could do with a stack, you could simulate with a vector. However, it is conceptually clean to have an object that models LIFO behavior and ensures you don't accidentally violate access (such as by reaching into the middle of the stack to insert or remove an element). We will also later see how it is possible for the internal design of the stack class to be implemented very efficiently by taking advantage of the restriction of LIFO access.

The public interface of the **`Stack`** class template is outlined below. The **`stack.h`** interface contains comments on the features.

```
template <typename ElemType>
  class Stack {

      public:
         Stack();
         ~Stack();

         int size();
         bool isEmpty();

         void push(ElemType element);
         ElemType pop();
         ElemType peek();
};
```

Here's some sample code that reads a line of user input, and uses a stack to store the individual characters which are then popped and printed in reverse order:

```
void ReverseResponse()
{
    cout << "What say you? ";
    string response = GetLine();
    Stack<char> stack;
    for (int i = 0; i < response.length(); i++)
      stack.push(response[i]);

    cout << "That backwards is :";
    while (!stack.isEmpty())
      cout << stack.pop();
}
```

The stack is discussed in reader sections 8.2-8.4 and 10.1. There is an excellent complete example in section 8.4 that demonstrates using a stack and scanner to implement a simple RPN calculator. The full **`stack.h`** file with comments is listed in the reader appendix. The class interface is also browsable from the "Documentation" link on our class web site.

Queue

The **`Queue`** is a cousin to the Stack that provides a vector-like collection that operates in first-in-first-out (or FIFO) order. Elements are added to the end of the queue and removed from the front. A queue thus models the standard waiting line of stores, banks, technical support phone lines, and so on.

There are many practical uses for queues, such as:

- managing traffic on a shared network connection
- handling user keystrokes
- ordering jobs for a printer
- implementing a breadth-first search

Although the queue offers no functionality not already present in vector, this class is still valuable in that it offers a clean conceptual model and an opportunity for an efficient design when the situation calls for FIFO behavior.

The public interface of the **`Queue`** class is outlined below. The full **`queue.h`** interface contains comments on the features.

```
template <typename ElemType>
   class Queue {

      public:
         Queue();
         ~Queue();

         int size();
         bool isEmpty();

         void enqueue(ElemType element);
         ElemType dequeue();
         ElemType peek();
};
```

Here's a sample use that shows processing a simple waiting line. At the prompt, the user can either enter a name (a customer to add to the line) or respond "next" for the next customer to be handled.

```
void ManageQueue()
{
   Queue<string> queue;

   while (true) {
      cout << "? ";
      string response = GetLine();
      if (response == "") break;
      if (response == "next") {
         if (queue.isEmpty())
            cout << "No one waiting!" << endl;
         else
            cout << "Handle customer " << queue.dequeue() << endl;
      } else {
         queue.enqueue(response);
         cout << "Add customer " << response << " to end." << endl;
      }
   }
}
```

The **`Queue`** class is discussed in reader sections 10.3-10.4. In section 10.4, there is a large example using queues in a simulation program. The full **`queue.h`** file with comments is found in the reader appendix. The **`Queue`** class interface is also browsable from the "Documentation" link on our class web site.

Associative containers

The container classes discussed so far (Vector, Grid, Stack, Queue) are referred to as *sequential containers*, because elements are stored and retrieved in order of sequence. While these containers are quite useful, we also need containers that are a bit fancier. How about being able to quickly search for an element or automatically maintaining the elements in sorted order or being able to easily merge two collections into a new combination? The *associative containers*, such as Map and Set, use information about the value of the elements being stored to arrange them in a manner to support these operations very efficiently. Sound good? Read on, to learn about the CS106 Map and Set classes…

Map

The **`Map`** class is an incredibly nifty container. It provides the abstraction of a *map*, which is the dorky-computer-scientist name for a collection of key-value pairs. The client associates a value with a key and can lookup by key to get the associated value back. Other names used for this data structure include *symbol table*, *dictionary*, or just plain *table*. One cool feature of a map is that adding an entry and looking up by key are typically implemented very efficiently, so that even when the map has thousands or millions of entries, these operations are practically instantaneous. As a client, we might scratch our heads and marvel at this, but even with no clue how it works, we can still repeat the benefits of this incredible efficiency.

You can do lots of handy things with a map, here are just a few ideas:

- A dictionary: words are the keys, associated value is the word's definition
- Axess: keys are student id numbers, value is student's transcript
- A document index: words are keys, value is a set of page numbers containing the word
- A symbol table for a compiler: variable name mapped to memory location
- DMV: key is license plate number, value is the registration information for the vehicle
- Even Wikipedia and Google are backed by enormous map data structures!

The public interface of the **`Map`** class template is outlined below. Note keys are always of type **`string`** (there is a reason for this restriction, we will discuss this when we talk about implementation), but the values can be of whatever type the client needs. The interface is written using the template placeholder **`ValueType`** that is filled by the client when declaring a map.

```
template <typename ValueType>
      class Map {
         public:

            Map();
            ~Map();

            int size();
            bool isEmpty();

            void add(string key, ValueType value);
            void remove(string key);

            bool containsKey(string key);
            ValueType getValue(string key);
   };
```

When calling the **`add`** member function, the client provides the key and value and the map stashes the pair for later access. Attempting to add a value for key that already has an entry in the map will replace the previous value with the newly added one. Calling the **`remove`** member function will remove the entry for a key.

The **`getValue`** member function allows the client to retrieve the value associated with a given key. This function has a slightly unusual design with regard to its return value. In the case where **`getValue`** finds the key in the map, it can return the associated value easily enough. However, what if the key being looked up isn't in the map? What, then, should the function return? Ideally, the returned value would some sort of "not found" sentinel that reports to the client that the key isn't contained in the map. But that sentinel needs to be of type **`ValueType`**, which is just a placeholder. The map is written as a template and doesn't make any assumptions about the true nature of **`ValueType`**. It might be a string, an integer, a structure, a pointer, or something else. Each of these types has its own unique value for what might make a good sentinel and it isn't the same for all of them. There is no easy way for a generic version of **`getValue`** to decide which is the right kind of sentinel value to return when the key is not found. Thus, the member function is written to raise an error if the client attempts to retrieve the value for a key that doesn't exist. It becomes the client's responsibility to first call **containsKey** to verify that a key does indeed have an entry in the map before attempting to get its value.

Here is a sample use that uses a map to record the frequencies of characters occurring in a file:

```
// Reads words from input stream until EOF/read failure.
// Each word is used as a key into the map, the associated
// value is the number of times that word has been read.
// When a new word is read, it is added to map with freq = 1.
// Each subsequent time a word is read, its frequency in the map
// is incremented by one.
// At function end, size of table is number of distinct words.

void MapTest(ifstream & in)
{
   Map<int> map;
   string word;

   while (true) {
      in >> word;
      if (in.fail()) break;
      if (map.containsKey(word)) { // if already have seen this word
        int count = map.getValue(word);
        map.add(word, count + 1); // incr value by 1 & update
      } else
        map.add(word, 1);         // add first occurrence of this word
   }
   cout << "Found " << map.size() << " unique words." << endl;
}
```

The map also provides a shorthand syntax for setting and retrieving values by overloading the square brackets operator, e.g. **map[key]**. The argument within the square brackets is not an integer index, but a string key, and the expression evaluates to the value associated with that key in the map. If there is no entry for the key in the map, using this expression <u>creates</u> a new entry for key in map. The associated value for that key will be the default value for **ValueType**, i.e. what you get when you declare a variable of that type with no explicit initialization. For strings, this would be the empty string, for an integer, it would uninitialized. The use of square brackets returns the value by reference, which allows you to use the expression to either get the value or to change it. The code below is a rewrite of the previous example, using square brackets instead of getValue/add.

```
void MapTest(ifstream & in)
{
   Map<int> map;

   while (true) {
      string word;
      in >> word;
      if (in.fail()) break;
      if (map.containsKey(word))     // if already have seen this word
        map[word]++;                 // increment value already in map
      else
        map[word] = 1;               // add first occurrence of this word
   }
   cout << "Found " << map.size() << " unique words." << endl;
}
```

Although you may find the use of square brackets a little bit funky at first (I know I did!), it is fast becoming the de facto syntax for accessing maps (in C++ as well as other programming languages) and once you becomes accustomed to it, you may even find it tidy and convenient.

Earlier we noted that the map assumes that keys are of **`string`** type. This can be seen as a bit of a restriction, but conveniently, it's not difficult to convert most data into a string representation to make it suitable for use as a key. For example, consider the problem of storing student information for the Axess system. The desired key is the student id number, which is of integer type, so you might think you were out of luck. However, one quick call to the **`IntegerToString`** function converts the id to a string equivalent, which makes a fine key:

```
struct studentT {
   string first, last;
   int idNum;
   string emailAddress;
};

void AddToMap(Map<studentT> & map, studentT student)
{
   // convert id number to string to use as key in map
   string idAsString = IntegerToString(student.idNum);
   map.add(idAsString, student);
}

string GetNameForId(Map<studentT> & map, int idNum)
{
   string idAsString = IntegerToString(idNum);  // convert id->string
   if (map.containsKey(idAsString)) {
      student found = map.getValue(idAsString);
      return found.first + " " + found.last;
   } else
      return "";
}
```

You might also see it is limiting that there can be only one value associated with a given key. Attempting to enter another value for an existing key replaces the previous value. However, there is an easy way to get a one-to-many mapping if you desire it: simply store a vector as the value for each key and add values to that vector. For example, consider a map that manages the index for a document. Each key in the map is a word and the associated value is a list of page numbers where that word appears. The list of page numbers can be stored using a vector of integers. Here is a bit of code that shows how you would add a new entry into such a map:

```
// This map represents a book index.  The keys are words in the book,
// the associated value is vector of page numbers where that word appears
// For example, the word "binky" might appear on pages 3, 5, 8, and 10.
// This function is given a partially complete index, a word, and
// a page # and updates the index to show that this word is referenced
// on that page.
void AddReference(Map<Vector<int> > & index, string word, int pageNum)
{
     index[word].add(pageNum);
     // wow, the above one-liner does a lot!
     // the [] retrieves either the existing vector already contained
     // in map, or creates a new empty vector and adds to map under key
     // that vector is then asked to add a new page number
}
```

Note that use of a nested template— a map where the values are vectors of integers. The syntax is a little crufty (be careful about the need for the space between the two closing angle brackets), but hopefully the expressive power that enables building these sophisticated structures makes it all worthwhile!

Iterating over a map

A map provides key-value access. If you have the key, you can look up the associated value, or enter a new value for that key. However, it's not obvious how to browse the map entries when you don't know the key you want. For example, consider finding the word with the most occurrences in the frequency map or identifying all students with last names beginning with Z or just printing all the entries in any map. The techniques that worked for the sequential containers don't apply here— the map is not a linear collection, the elements are not stored in order, and you cannot run a loop over the collection's size calling **`getAt`** or **`dequeue`** to get each element in turn.

Instead, the map allows the client to get access to the entries by providing an *iterator*. An iterator is a helper object that works in conjunction with a container object (in this case, the map) to step through the elements contained in the collection. When you want to browse the entries in a map, you ask the map to create an iterator for you, and you use that iterator object to visit the keys from the map one by one in a loop. Because the iterator is tightly integrated with its container, it is defined as a nested class, i.e. within the scope of the collection class. The name for the map's iterator is **`Map<ValueType>::Iterator`**. An **`Iterator`** has two public member functions: a predicate **`hasNext`** that returns true if there are more elements remaining in the iteration and a **`next`** function that returns the next element in the iteration. The idiomatic iteration loop is similar to using a scanner: while there are still elements remaining, retrieve the next one. Consider this loop that prints the keys within the map:

```
void PrintKeysFromMap(Map<int> & map)
{
    Map<int>::Iterator itr = map.iterator();
    while (itr.hasNext())
       cout << itr.next() << endl;
}
```

Each key in the map will be accessed once and only once in an iteration, but the keys can be visited in any order (e.g. the map iterator does not guarantee to return keys in alphabetical order or in the order they were inserted).

You'll notice that the iterator returns only the key. If you also need access to the value, it's just a call away. For example, to print the frequency entries, this code does the trick:

```
void PrintFrequencies(Map<int> & map)
{
    Map<int>::Iterator itr = map.iterator();
    while (itr.hasNext()) {
      string key = itr.next();
      cout << key << " = " << map[key] << endl;
    }
}
```

Mapping over a map

Iterators are one way that a container can provide the client access to the entire collection of data being stored. Another common strategy is for the class to provide a *mapping* operation. (Confusingly, the use of the verb "map" for this operation has no relation to the fact that the class we are talking about is also called "map") Typically what the client wants is to "do something" for each entry in a collection (output it, write to a file, add to a summary, examine to find max). The collection class can offer to take the client's desired action and apply it to each entry in the map. The only tricky part is how the client and implementer communicate about what the client wants done. The mechanism used is the *callback function*. The client writes a function that will do the desired action on a single entry. The client passes that function to the collection object, which then invokes the client's callback once for each entry. In the case of the **`Map`** class, the mapping operation might be defined something like this:

```
template <typename ValueType>
 class Map {
   public:
      // rest of public features removed for clarity

      void mapAll(void (clientfn)(string, ValueType));
};
```

Previously we showed printing the frequency map using an iterator, in order to use mapping instead, we first write a callback function that has the correct prototype and prints a single entry:

```
void PrintEntry(string key, int value)
{
      cout << key << " = " << value << endl;
}
```

To print the entire map, we ask the map to **`mapAll`** using our callback. The **`mapAll`** member function will iterate through the map entries and call the **`PrintEntry`** callback function once for each key-value pair in the map. Pretty slick, eh?

```
void PrintFrequencyMap(Map<int> & map)
{
      map.mapAll(PrintEntry);
}
```

To make a mapping operation more generally useful, it's necessary to have an enhanced version where the prototype for the callback function has the usual arguments for the entry information plus one additional parameter. The type of that additional parameter is left as a placeholder (using a template) that allows the client to specialize the type to fit their needs. This extra parameter is called the *client data*. It gives the client the opportunity to pass in some additional information into the callback function to give any needed context for the callback to do its work. The client passes the initial value of the client data to the **`mapAll`** member function and it is then passed back to the client on each invocation of the callback. Thus, this version looks like this:

```
template <typename ValueType>
 class Map {
   public:
      // rest of public features removed for clarity

      template <typename ClientDataType>
        void mapAll(void (fn)(string, ValueType, ClientDataType &),
                        ClientDataType & data);
};
```

Now that the client data parameter is passed by reference. In most situations, the client data is being updated during the mapping operation and those changes are expected to be persistent, thus the data is passed by reference to easily provide for this.

This client data parameter is essential if there is any additional context needed in the callback function when taking action on the entry. For example, if you wanted to print the entries to a file stream, instead of the console, the callback function is going to need to know which stream object to use for output. You can use the client data parameter to pass that stream into the callback function, as shown below:

```
void PrintEntryToFile(string key, int value, ofstream & out)
{
      out << key << " = " << value << endl;
}

void PrintFrequencyMap(Map<int> & map)
{
      ofstream out("frequencies.txt");
      if (out.fail()) Error("Could not open file");
      map.mapAll(PrintEntryToFile, out);
}
```

In the example below, you wish to search a map to find the longest key. The client data parameter can be used to store the longest key seen so far. On each invocation of the callback, you compare the current key with the longest and update if this entry is longer. After all entries have been mapped, the longest key will be stored in the client data parameter.

```
void FindLongest(string key, int value, string & longest)
{
      if (key.length() > longest.length())
         longest = key;
}

string LongestKeyInMap(Map<int> & map)
{
      string str = "";
      map.mapAll(FindLongest, str); // str updated in the callback
      return str;
}
```

Mapping and iteration can be used to accomplish similar tasks, so it might help to consider why you might prefer one to the other. One difference is in terms of who handles the loop logistics. Using an iterator, the client takes a bit more responsibility, i.e the setup-retrieve-advance sequence (admittedly this is not a lot of code, nor is it tricky), as opposed to mapping where all of the looping is handled in the implementation. However, taking control of the loop gives the client the flexibility to end the loop early or skip over entries. If using mapping, the client callback function provides a nice small unit of decomposition and allows the overall operation to be neatly coded in terms of performing one action on each entry. However, sometimes it can be awkward to wedge all the needed context into the client data parameter.

Set

Next up, we have our `Set` class. A set is used to store a collection of unique elements and serves as the analog to the sets you know from mathematics. Some of its key features are:

- Unlike a vector, set elements are not stored linearly, and are not assigned/retrieved by index. The sequence in which elements are added is not of consequence.
- Sets do not store duplicate elements. Attempting to add an element that the set already contains is a no-op.
- A set can quickly determine if it contains a given element.
- Sets provide nifty set-level operations like equals, subset, and union/intersect/subtract that do

comparisons and combinations of one entire set with another. These operations are typically implemented quite efficiently.

- The set needs to compare elements to see if they are identical, to check for membership, enforce non-duplication, and so on. It will attempt to use the built-in relational operators to compare elements, but if that is not legal or appropriate for your needs, you have the option of supplying a comparison callback function when creating the set. (More on how this works later in this handout)

What are sets good for? All sorts of things! The set of synonyms for a given word, the set of email addresses on a mailing list, the set of requirements to graduate, the set of options on a car, you name it! You can throw a bunch of items into a set just to coalesce duplicates. The set is especially snazzy when you need set-level operations, like intersecting the pizza topping preferences of your friends to find a pizza they would all enjoy.

The public interface of the `Set` class is outlined below. The comparison function passed to the constructor is discussed in more detail below.

```
template <typename ElemType>
  class Set {
    public:

      Set(int (cmpFn)(ElemType, ElemType) = OperatorCmp);
      ~Set();

      int size();
      bool IsEmpty();

      void add(ElemType element);
      void remove(ElemType element);
      bool contains(ElemType element);

      bool equals(Set & otherSet);
      bool isSubsetOf(Set & otherSet);


      void unionWith(Set & otherSet);
      void intersect(Set & otherSet);
      void subtract(Set & otherSet);

      Iterator iterator();
      void mapAll(void (clientfn)(ElemType));
      template <typename ClientDataType>
        void mapAll(void (clientfn)(ElemType, ClientDataType &),
                    ClientDataType &data)
  };
```

First, off, here is a trivial example that shows creating the set of unique characters from a string:

```
void SetTest(string str)
{
   Set<char> set;
   for (int i = 0; i < str.length(); i++)
      set.add(str[i]);
   cout << "There are " << set.size() << " unique chars.";
}
```

Now, let's use sets to help ferret out just how random the random library is. The idea is to count how long a sequence of random numbers you can generate before you get a repeat. We use a set to store the random sequence and check each number for membership in the set to determine if it has previously been used.

```
void RandomTest()
{
   Set<int> seenSoFar;
   while (true) {
      int num = RandomInteger(1, 100);
      if (seenSoFar.contains(num)) break;
      seenSoFar.add(num);
   }
   cout << seenSoFar.size() << " unique numbers before repeat.";
}
```

Similar to the map, the set provides an iterator to give access to the elements. You ask the set to create an iterator and then use the standard iteration loop to walk through the elements one by one. Here is a function that prints a set of doubles using the iterator:

```
void PrintSet(Set<double> set)
{
   Set<double>::Iterator itr = set.iterator();
   while (itr.hasNext())
      cout << itr.Next() << " ";
}
```

The Set iterator does guarantee that it will visit the set elements in sorted order, where "sorted" is either determined by the natural ordering of the elements (e.g. numeric or alphabetic order) or according to client's wishes specified by the comparison function (discussed below). There is also a mapping operation that visits the elements in sorted order, calling the client's callback once for each element. Here is an example of printing a set using **`mapAll`**.

```
void PrintElem(double d)
{
   cout << d << " ";
}

void PrintSet(Set<double> set)
{
   set.mapAll(PrintElem);
}
```

Sets really shine when your algorithm requires the high-level operations that work on entire sets, such as testing for set equality or combining two sets to compute the intersection or difference. Want to schedule a meeting for everyone in your project group? Intersect the set of times each member has available. Want to check if a student can graduate? Verify that the set of required courses is a subset of the set the student has taken. Compound database queries are naturally expressed using set operations. You want to find people who live in your dorm who like skiing or Bartok but not Thai food? Use intersection, union, and difference to construct the result.

Here's a simple example that is given two people and their set of friends and enemies and will use set operations to construct a guest list for a joint part that contains all friends and excludes either's enemies:

```
struct personT {
   string name;
   Set<string> friends, enemies;
};

Set<string> MakeGuestList(personT one, personT two)
{
   Set<string> result = one.friends;   // start by copying one's friends
   result.unionWith(two.friends);      // join with two's friends
   result.subtract(one.enemies);       // take out one's enemies
   result.subtract(two.enemies);       // and two's enemies, too
   return result;
}
```

The Set class is unique in that it needs more information about the type of element being stored than the other template containers. Vectors, stacks, queues, and maps just take the client's data and stash it somewhere to be returned later. Those containers never examine the elements or do any operations on them. But a set needs to be able to compare two elements to determine if they are the same (this is needed to avoid entering duplicates, to check if an element is contained, to compute the intersection, and so on). But the set is written as a template, when means the element type is merely a placeholder, and the "real" element type isn't known until the client allocates the set. So how then can the `Set` class do its job without the full information?

As a default, the set tries to use the built-in relational operations (e.g. ==) to compare two elements. For some element types (such as int or string), this will work just fine. However, let's say the client tries to create a set containing student structs, something like this:

```
struct studentT {
   string first, last;
   int idNum;
   string emailAddress;
};

int main()
{
   Set<studentT> students; // compile error!
```

The above code will generate a compile error that is reported something like this:

```
Error: no match for 'operator==' in 'one == two'
Error   : illegal operands 'studentT' == 'studentT'
 (point of instantiation: 'main()')
  (instantiating: OperatorCmp<studentT>(studentT, studentT)')
cmpfn.h line 25       if (one == two) return 0;
```

The compiler tries to instantiate the set of students using operator-based comparison which attempts to compare two student structs using ==, yet == does not have a valid meaning for structure types. What's happening here is that there isn't a default interpretation for what it means for two student records to be the same. Does every field have to match or just the name or id? The client who defined the struct is the only one who truly knows what it means for two of those structs to be considered equal. How can the client inform the set how to compare two student structs? The client writes a function that takes two structs and returns the result of comparing the two. The client then passes that function to the set constructor. The new set will call the client's function it needs to compare two elements to determine their equality. Here's how to fix the non-compiling example from above:

```
// comparator function for two student structs
// compares the id field to determine equality/ordering
int CompareStudents(studentT one, studentT two)
 {
   if (one.idNum < two.idNum) return -1;
   else  if (one.idNum == two.idNum) return 0;
   else return 1;
}

int main()
{
   Set<studentT> students(CompareStudents);
```

The client defines the function `CompareStudents` that takes two studentT structs and returns the comparison result based on the id number. A comparison function is expected to return –1 (or some other negative number) if the first parameter is "less than" the second, 0 if the two are equal, and +1 (or some other positive number) if the first is "greater than" the second.

This function `CompareStudents` is then passed to the constructor when the set is created. This configures the set to skip the default comparison (e.g. ==) and instead call the client's supplied function whenever the set needs to compare two elements. This kind of function is called a *callback* function because the set calls back out to the client.

Cool combinations

Each of the container classes is very useful in its own right but in combination, things just get even better. Need to store information for a list of class lists? A vector of vectors could do the job. Need to do model a thesaurus? A map associating each word to a set of other words will do nicely. Need to simulate the many check-out lines in a supermarket? A vector of queues could be just the thing. Need to represent the nested scopes for a C++ compiler? A stack of maps is perfect for the task.

In all these cases, your work is made much easier and fun because you get to focus on writing the neat applications, having been freed up from dealing with gunky, mundane code required for common data structures! The CEO of an unnamed black workstation company I once worked for used to call building on a great class library like "starting construction on the 25th floor", so let's hear it for building our own fabulous skyscraper!

A last word on templates

Container classes are supplied as C++ templates, which is a generic form capable of storing whatever type data the client needs with complete type safety. This is a huge benefit. However, templates require some special handling and the syntax can be clunky. Here are some guidelines to keep in mind as a client of our 106 template classes:

- Be sure to **`#include "`*`classname`*`.h"`** for each template class you are using.
- In client use, the template class name must always be followed by the completed placeholder type in angle brackets, e.g. `Vector<string>`. This means everywhere: in variable declarations, parameter types, return types, calls to new, etc. You can define a `typedef` nickname if you find using the full template name too cumbersome. Nested template types (e.g. Maps containing Vectors) are particularly benefited by this.
- The errors reported by the compiler when you've made a mistake using a template can be frustrating and unhelpful. If you're stuck, please bring it to one of the course staff who can help interpret what the compiler is complaining about.

Q: Why was the Java programmer drowning?
A: He wasn't above C level.