

## CS106 Library Classes

---

Kudos to Julie Zelenski for this fantastic summary of the CS106 utility and container classes.

This winter I'm going to try something that another lecturer successfully tried last quarter. After we cover the basics of class interface design, publicity versus privacy, constructors, destructors, and the mechanics of implementation, we're going to put on our client caps and learn some of the utility classes written specifically for the CS106 courses. In the past, we've focused on both the design and the implementation of each class, one at a time, alternating between user and implementer. This time, we're going to learn all the classes up front, internalizing their interface, their behavior, their runtime characteristics, and the situations where they come in most handy. The ideas modeled by these classes are common enough that all modern programming languages provided similar library classes—Java provides things like `StringTokenizer`, `ArrayList`, and `HashMap`, whereas the C++ STL provides classes like `vector`, `set`, `map`, and `hash_map`. In principle, we could just learn the STL classes, but they're cumbersome and difficult to use for someone programming in C++ for the very first time. So we instead go with some simpler versions of the very same ideas. This way we can emphasize concept instead of syntax.

Unfortunately, the reader hasn't been updated to reflect our new stance on teaching these classes, so you'll have to rely on this handout, the reader's appendix, and Jerry's brilliant explanations for a few days until we start talking about implementation.

### Scanner

The `scanner` class provides functionality for dividing a string into separate words or tokens. A scanner comes in quite handy when doing any sort of string or file processing. Here's a few ideas to get you thinking about its uses:

- breaking apart a command entered at the console (e.g. `"turn left"`, or `"find price < 5"`)
- 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 `/usr/class/cs106x/www/handouts/01-Course-Information.pdf`)
- evaluating an arithmetic expression, such as `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);
    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 string to be scanned, and then enter a loop that calls `ReadToken` while `MoreTokensExist` returns `true`. Here's a simple example that uses the `scanner` to reports the number of tokens entered in a user's response:

```
void CountTokens()
{
    cout << "Please enter a sentence: ";
    string response = GetLine();

    int count = 0;
    Scanner scanner;
    scanner.setInput(response);

    while (scanner.hasMoreTokens()) {
        scanner.nextToken();
        count++;
    }

    cout << "You entered " << count << " tokens." << endl;
}
```

The next example shows finding the longest token read from a file. In the outer loop, the `getline` function retrieves each line from the file and in the inner loop, that line is tokenized using a `scanner` object.

```

// 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) {
        string line;
        getline(in, line);
        if (in.fail()) break; // no more lines to read
        scanner.setInput(line);
        while (scanner.hasMoreTokens()) { // inner loop tokenizes
            string token = scanner.next_token();
            if (token.length() > longest.length()) {
                longest = token;
            }
        }
    }

    return longest;
}

```

There are various options that you can set on a scanner to determine how it handles certain special tokens. Most of these options are only needed for advanced use, but one that you commonly manipulate is the `spaceOption`, which controls whether whitespace tokens are returned from `readToken` or skipped over. The default is for spaces to be returned, but you can cause them to be ignored by changing the option:

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

This is handy when you are interested in processing only the non-space tokens. Note that the `spaceOptionT` enum is defined within the `Scanner` class, so when referring to those values as a client, we 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.

## 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 attributes that a car has. Container classes are incredibly leveraged functionality because pretty much all programs need to store things and the ways they need to organize it follows some very standard patterns.

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. Class templates are discussed in section 10.1 of the reader.

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 support 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 is bit clunky 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.util.ArrayList`!) A `vector` serves the same basic purpose as the built-in C++ array. Why, then, would you want to use a `vector`?

- A `vector` knows its length
- 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 make/close up space

Once you start playing with the `vector`, you'll never go back to the standard array. This isn't to say the traditional array is useless: The `vector` class (and many other classes) are backed by arrays. But more often than not your linear sequence of data need to expand and contract at runtime, and the `vector` manages all of the allocation, resizing, and shifting and splicing in of data that you'd otherwise have to manage yourself.

The 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.

```
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);
};
```

Our `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 (i.e. same as array selection syntax). 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.

Pretty much anything you could have used an array for, using a `vector` will be easier and less error-prone. Here's a sample program that reads scores from the user into a vector of `ints`.

```
// 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 put the value in ascending order, with this small change to the above code:

```
// use this code instead of v.add(num) above
int pos = 0;
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!

And lastly, here is some sample code that shows reading the words from a file and arranging 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 an instance of the right-shift operator. Luckily, most compilers produce a pretty

reasonable error message if you forget the space (Metrowerks says "Illegal template argument") so hopefully you'll quickly learn to avoid this little pitfall. By the way, you can use typedef to introduce a shorthand name for any C++ type. That nickname can then be 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;
```

## 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 on the class web site 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)`. (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.)

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 reference to avoid making expensive copies when not explicitly needed.

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
}
```

## Stack

The `stack` is another simple container class—even simpler than `vector`. A `stack` is 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)
- Temporary shadowing (pushed element shadows those further down the stack)
- 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 full `stack.h` interface on the class web site 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();
}
```

## Queue

The `queue` is a cousin to the `stack` that provides a list that operates in first-in-first-out (or FIFO) order. New elements are added to the tail of the `queue` and elements are removed from the head. A `queue` thus models the standard waiting line of restaurants, banks, etc. There are many practical uses for queues—managing traffic on a shared network connection, handling user keystrokes, ordering jobs for a printer, implementing a breadth-first search, and more. Although it offers no functionality not already present in `vector`, a `queue` 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 on the class web site 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;
        }
    }
}
```

## Associative Containers

So far we've discussed the `vector`, `Grid`, `stack`, and `Queue` storage classes. The classes are sometimes collectively referred to as sequential containers, because elements are stored and retrieved in order of the sequence in which they were inserted. While these containers are quite useful, we also need containers that are a bit fancier. How about being able to quickly lookup an element, automatically maintaining the elements in sorted order, or being able to easily combine two collections into a new result? 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!

### 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 stellar performance.

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
- Access: keys are student id numbers, value is student's transcript
- A document index: words are keys, value is a set of pages where the word is referenced
- 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

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

```
template <typename ValueType>
class Map {

    public:
        Map(int sizeHint = DefaultStartingSize);
        ~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 be 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(istream & in)
{
    Map<int> map;
    string word;
    while (true) {
        in >> word;
        if (in.fail()) break;
        if (map.containsKey(word)) // if already have seen this word
            map.add(word, map.getValue(word)+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 creates 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 be 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(istream & in)
{
    Map<int> map;
    string word;
    while (true) {
        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, 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.