CS032 2009 Lab 10

# Standard Template Library

## Introduction

Last week, we discussed using system calls, the lowest useable level of code available. Obviously you could write in Assembly if you wanted. Hell, you could even use a magnifying glass, a magnetized needle, and a very steady hand[1], but that would be stupid. Regardless, just because you know how to manipulate the system doesn't mean you have to prove it at every available opportunity. In all honesty, someone out there is probably a more leet haxor and has implemented your desired data structure better than you ever could. It's a better idea to just swallow your pride and use his[2] version. This week, we'll be doing exactly that.

The Standard Template Library (STL) is a C++ library which gives you access to all of the most common data structures and algorithms. In this lab we'll cover three useful container classes from the STL: list, map, and deque. SGI has excellent documentation available on the STL. Keep it bookmarked and refer to it forever:

`http://www.sgi.com/tech/stl/`

It's important enough to mention again, even though you just saw it!

`http://www.sgi.com/tech/stl/`

We won't tell you everything you need to know. You will need to look at this documentation for the specifics on how to use the STL!

`http://www.sgi.com/tech/stl/`

The T in STL comes from the fact that most objects in the library are C++ templates. A template behaves just like an object except that it can be parameterized by type, which is a fancy way for saying that you can specify what types are used internally to the object from outside of the object. Examples: `list < People >` is a list of People objects; `list < double >` is a list of doubles; `map < int, string >` is a map with integer keys and string values. This can be done because a container can be easily ignorant of what you put into it, so you can use the same code and just parameterize what types it will operate on. Java does this now, but back in the dark ages of pre 1.5, containers could only hold things that of type Object. Also, the compiler couldn't give you any help on determining what type you'd get out of the Java containers, so you had to cast them yourself and hope you got it right. With templates you can do true compile-time type checking, which is a good thing[3]

Template objects take, as parameters, the types that they are going to use, inside the < and > symbols. By using templates, the STL allows you to use a single list class to create a list of People or a list of Recipes or a list of Lists.

## Containers

- List
  An STL list is a doubly-linked list, supporting both forward and backward traversal. `std::list <` *`type`*`> myList;`

  1. This is a linked list, so everything you remember from CS 16, CS 18 or CS 19 will apply.
     It's great for iterating forwards and backwards and for constant time manipulation( inserting and deleting ), but accessing an arbitrary element will take linear time.
  2. Singly linked lists are known as `slist`s and may be more efficient if you only need to go in one direction.

[1] http://xkcd.com/378

[2] Probably his, not hers, let's be real for a minute

[3] Note that Java's generics in versions 1.5 and later are comparable to, but not exactly like, templates. Take CS173 to learn more about the differences.

- Map
  A `map` is implemented as a red-black tree(though you've probably forgotten how they're implemented) that stores and sorts data by separate key values. You can think of it as an array that you can index with anything.

  `std::map <` *`key type, data type`*`> myMap;`

  1. Each key value can only be associated with a single datum. If you'd like to have multiple data per key use a multimap.
  2. To get something out, use the find function. More on this when we get to iterators. You can also use subscripting (foo = myMap[3];) to access data, but if there is no datum for the key you subscript with one will be created for you, and you might not want that. Keep it in mind.
  3. Insertion into a map is done with subscripting: myMap[3] = foo; will put the data from the variable foo into the map for the key 3. This will overwrite any entry that's already there for that key, so make sure that's what you want to do[4].
  4. You can choose how your map sorts its keys by adding an optional comparator to the template arguments (right after the required data type). It defaults to less < key type >, but others (like greater) are available and you can write your own. Comparators are an excellent example of the power and flexibility of templates.

  `http://www.sgi.com/tech/stl/Map.html`

- Deque[5]
  A `deque` is a Double-Ended QUEue that supports constant time insertion at the beginning and end of the sequence, and constant time access of elements. Use it when you might have used a vector or an array in another language.

  `std::deque <` *`type`*`> myDeq;`

  1. Befitting their name, `deque`s have `push_back(`*`type`*`) , pop_back(`*`type`*`)` and other functions that make them very stack or queue-like.
  2. They also support vector-like behavior, unlike `list`s, such as constant-time random access of elements using subscripting. `foo = myDeq[0];` sets foo to the value of the first element of the `deque`. Be warned that subscripting is not range checked for `deque`s, and if you try to access more elements than exist in your `deque`, your program may encounter a segmentation fault.

  `http://www.sgi.com/tech/stl/Deque.html`

# Iterators

An iterator is an abstraction of the pointer to an individual item within the data structure. For example, if you are using a `deque` of `string`s, a `deque` iterator represents a pointer to one string within the `deque`. An iterator for our `deque` of `string`s is defined as follows:

```
deque < string > :: iterator myIter;
```

Iterators are useful for access and traversing through your data structures. The following is sample code on how to use an iterator to run through a deque of strings[6]:

```
// define in header to avoid writing out template syntax each time
typedef std::deque< std::string > StringDeque;
typedef StringDeque::iterator StringDequeIterator;
```

[4]This is especially important if your map is storing pointers to objects as data. Replacing a pointer in the map will not change the memory that that pointer pointed to, nor will it delete that memory.

[5]Pronounce it "deck"

[6]It might not make a tangible difference to you, but algorithmically, it's probably better to use the pre-increment operator ++var rather than the post-increment operator var++, since the compiler will not have to make a temporary copy of the object for the iteration.

```
// put this code where you want to do your traversal
StringDequeIter nomad;
for(nomad = myStringDeque.begin(); nomad != myStringDeque.end(); ++nomad) {
     do nomad stuff
}
```

For most containers, you can access the first item using the `begin()` member function. The `end()` function returns a special "past-the-end" iterator. "Past" is the key word here, because this iterator does not refer to any element of the container, it refers to the theoretical space directly after the last element. It may be confusing at first, but it's handy to have an iterator that isn't associated with an element. Iterators are objects that get allocated on the stack, but the magic of C++'s operator overloading allows them to work syntactically as if they were straight pointers. You should know by now that * is used to dereference a pointer, that is to say, to get the thing this pointer points to. Iterators work just the same way [7].

`http://www.sgi.com/tech/stl/Iterators.html`

## Warning! Danger Will Robinson!

When using STL containers you must still delete your own memory. The `erase()` (removes a single element) and `clear()` (empties the entire container) functions won't do your clean-up for you. If your data structure contains pointers to heap-allocated objects, you must iterate through it and delete all of the objects pointed to by the iterators to free the memory. But only delete things that you have newed yourself, don't try to clean up the container by deleting the iterators! This will corrupt the container and give you weird errors. In this lab we expect you to manage your memory and delete everything when you are done. Depending on the data structure, iterators will or will not hold up when you modify their parent container. list iterators can handle insertions and removals on the list, but insertions and removals on a deque will cause all iterators for that deque to not work anymore. You'd need to get new ones by calling `begin()`, etc.

## Algorithms

The STL also contains a large collection of algorithms that are useful in manipulating the information and data stored in containers, and can do it in general ways thanks to templates. The following are some useful examples. See the STL documentation online for more information and a complete listing of all available algorithms.

- `reverse`: Reverses the data of a container in a given range.
- `find`: Returns the first found iterator between to other iterators that points to the given value. min element: Finds the smallest element in the given range.
- `sort`: Using just less-than operator sorts all the elements between two iterators. Your days of sort-algorithm implementation are officially over.
- `random shuffle`: Shuffles the elements between two iterators. Your days of shuffling randomly using malfunctioning sort algorithm implementations are officially over.

## Assignment

Now it's time for fun with the STL! Now that we're about three quarters of the way through the semester, the TAs have noticed that writing down grades on greasy used napkins isn't working too well. So, we want you, our awesome students, to make a grades database for us. We know you will make extra sure that no grades will be lost, because, well, they're yours.

To get started, grab the support code from /course/cs032/pub/stl using `cp -r /course/cs032/pub/stl /u/account/course/cs032/` . We provide a Makefile for this assignment. To compile type `make all`. To run type `./grade`.

[7] When you're storing pointers in your container the iterator will be a pointer to a pointer, so dereferencing the iterator once will give you a pointer, and dereferencing twice (**) will give you what that pointer points to.

We have given you a fully functional UI that links up to a GradesDatabase object that stores the information for it. What you have to do is set up the database to keep track of the grades:

- Keep track of all students in a `list`
- Keep track of all assignments a `list`
- Keep the grades in a `map` (You should map pairs of strings to doubles)

We have given you 4 files: common.H, grades.H, grades.C, and main.C, as well as a Makefile. The only file you need to edit is grades.C, which is full of empty methods for you to fill in, but you should take a quick look at the headers first.

There is a demo for you to try: /course/cs032/demos/stl

## Exercises

1. We have filled in addStudent, printStudent, overwriteGrade, and printGradesByStudent to give you examples of how to work with stl iterators and functions.

2. Fill in the list methods: removeStudent, addAssignment, removeAssignment and printAssignments. (HINT: some of these are VERY similar to the TA provided code)

   Checkpoint 1: Show your lab TA the code for your list methods.

   TA Initials: _____ (40 pts)

3. Fill in {set,get}Grade and doesGradeExist. You should now be using a map, as described above.

4. Now that we have grades, we need to change removeStudent and removeAssignment to remove the grades for that student/assignment. Change the methods to do this.

   Checkpoint 2: Show your lab TA the code for your grade methods and the updated removes.

   TA Initials: _____ (20 pts)

5. Finally, fill in the display methods: printGradesByAssignment, getFinalGrade and getAssignmentSummary. getFinalGrade should calculate the average of all of a student's assignment grades, giving them a 0 for any assignment they do not have a grade for. getAssignmentSummary should also calculate the average (do NOT give missing students 0's here), as well as the standard deviation. For anyone who hasn't taken statistics, the standard deviation is:

   $$\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

   Where the $x_i$'s are our numbers, and $\bar{x}$ is their average.

   Checkpoint 3: Show your lab TA the code for the entire database.

   TA Initials: _____ (40pts)

# C++ Appendix

1. **Creating a C++ class**

   If you look in grades.H, you will see a definition of the class GradesDatabase:

```
using namespace std;  \\All the files in the C++ standard library declare all
of its functions within the std namespace. We include this statement so that
we can use cout and cin, functions of the iosteam to print out our values to
the terminal.

class GradesDatabase {

public:  \\public methods and variables go here

  GradesDatabase(); \\constructor
  ~GradesDatabase(); \\destructor

  void printStudents(); \\public void method
  int addStudent(const string* stud); \\public method that returns an integer
  int removeStudent(const string* stud);

  ...

private: \\private methods + variables go here
   map<pair<string, string>, double>   gradeMap; \\instance variable
                                                        gradeMap of type map
   list<string>                       studentList;
   list<string>                       assignmentList;
}; // semi-colon ends the definition
```

   Note that the .H file only creates the definition of the class. You need to then fill in all the methods inside the corresponding .C class.

2. **Defining methods**

   Note the headers for the methods in the grades.C file:

```
 int
GradesDatabase::removeStudent(const string* stud)
{
}
```

   The *scope operator*:: is used when declaring methods. I have a class called `GradesDatabase` and it has a method called `removeStudent`, when defining the function in the .C file, I call it `GradesDatabase::removeStudent()`. The scope operator is needed because a .C file could contain method definitions for multiple classes, so we need to know for which class each method is being defined. Note that we don't need to name our .C file or our .H after the class, as in GradesDatabase.H/C. Though we would advise you to seperate your classes into seperate .C and .H files when writing complex projects.

3. **Constructor/Destructor**

   The constructor and destructor methods look like this in C++:

```
\* constructor *\
GradesDatabase::GradesDatabase()
{
}
```

```
\* destructor *\
GradesDatabase::~GradesDatabase()
{
/* clears the STL structures */
   studentList.clear();
   assignmentList.clear();
   gradeMap.clear();
}
```

The constructor method name is the same as the name of the class, this is very similar to Java and is automatically called when the class is initialized. To initialize an object from the heap, you need to use `new`:

```
GradesDatabase *gradesDB = new GradesDatabase();
```

The destructor is implied in Java (though you can write one yourself). In C++, since we don't have a garbage collector, the classes that are initialized need to be deleted. When they are deleted, the deconstructor is called. The deconstructor should contain all the cleanup code, ex: free all the memory that the class initialized for classes and structures. In our class, we simply clear our STL lists and map. You can delete a class with `delete`:

```
delete gradeDB;
```

Every class that is allocated with a `new` should be deallocated with a `delete`. This similar to C where you need to `free` for every `malloc`.

You can also initialize the class on the stack as we do in main.C:

```
GradesDatabase db;
```

This makes the db a local variable in main, but since main won't exit till the program exits, we don't need to worry about this object going away, we also don't need to delete it. Be very careful when allocating objects on the stack, usually reserve this practice for objects initialized in main().