Assignment 3

Due: November 1st, 2015 before 11:59pm

Objectives

- Introduction to the C++ Standard Template Library (STL)
- Exposure to generic abstract data types using C++ templates
- Practice using iterators
- Exposure to an $O(n^2)$ sorting algorithm: bubble sort.

Introduction

To this point in the course our data structures have always stored a specific fixed type – for example, the lists in assignment 2 stored elements of type `int`. If you wanted to modify your assignment 2 code to store elements of type `string` you would have to create a new list class. Obviously it is too much work to create a new list class for each type you want to store in the list. The solution, in C++, is to use something called *templates*.

This assignment will also introduce you to *iterators* – a mechanism for accessing elements in a container.

You will practice using templates and iterators by implementing a simple sorting algorithm, named bubble sort.

Finally, complete the A3 quiz on conneX to answer questions relating to the efficiency of algorithms using big O notation.

Quick Start:

1. Read this entire document
2. Read `part1.h` carefully and implement functions in `part1.cpp` until all tests passed
3. Read `part2.h` carefully and implement functions in `part2.cpp` until all tests passed
4. Complete the online quiz for A3 on conneX

The C++ Standard Template Library (STL)

The STL contains implementations of many of the Abstract Data Types we will see in the course. In this assignment, we will use two implementations of the List ADT:

`vector` – a list implemented using arrays

`list` – a list implemented using doubly linked lists

More information about these classes can be found here:

http://www.cplusplus.com/reference/vector/vector/

http://www.cplusplus.com/reference/list/list/

Templates

A simple example illustrates the use of templates below. You will note that when you declare an instance of `vector`, you must also specify the type of the elements you want to put in the `vector`:

```
#include <vector>
#include <list>

vector<string>      stringVector;
vector<int>         intVector;

stringVector.push_back("hello");
stringVector.push_back("world");

intVector.push_back(99);
intVector.push_back(23);
```

The type you want to store in the container is specified between < and >. The example below illustrates declaring an instance of `list` that stores elements of type `string` :

```
list<string>   stringList;
```

operator[]

An instance of `vector` can be used almost exactly like an array using the `[]` operator. For example, accessing all the elements in a `vector` can be accomplished like this:

```
vector<string>      stringVector;


for (unsigned int i=0;i<stringVector.size();i++) {
          cout << stringVector[i] << endl;
}
```

The `[]` operator on `vector` is constant time, O(1).

The `list` does not provide an implementation of `[]`, because accessing an arbitrary element in a linked list is O(N) and most programmers would expect that using the `[]` operator would be constant time.

However, it is still possible to access elements one after another in a list, and the mechanism for accomplishing that are called *iterators*.

Iterators

An iterator is an object which represents position in a container. The implementation of `vector` and `list` provide iterators.

The iterator is typically used to traverse all the elements in the container, like:

```
list<int> theList;
list<int>::iterator i = theList.begin();
while (i != theList.end()) {
      cout << *i << endl;
      ++i;
}
```

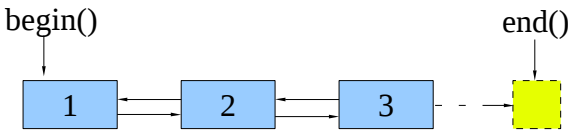
The use of iterators looks like pointers. To access the element currently referenced by the iterator i, you write: `*i` To move the iterator forward in the list, you write : `i++` To move the iterator backwards in the list, you write: `i--`

In C++, containers such as `list` and `vector` implement the `begin()` and `end()` methods that return iterators.

In a non-empty list, the `begin()` method returns an iterator referring to the first element in the list.

The `end()` method returns an iterator representing the position past the last element in the list. In an empty `list` named `theList`, `theList.begin()` == `theList.end()`.

Pictorially, the list {1,2,3} with `begin()` and `end()`:



`end()` doesn't reference an actual node in the list. It can only be used to test if the iterator has been advanced past the last element with a statement like:

```
while (i != theList.end()) {
      // there are still elements in the list
      i++;
}
```

Multi-file C++ Programs

As in the previous assignments, you've been given multiple files.

This assignment has two "main" programs: `part1_tester.cpp` and `part2_tester.cpp`.

In order to create the test program for `part1`, type:

```
g++ -Wall part1.cpp part1_tester.cpp -o part1
```

To create the tester for part2, type:

```
g++ -Wall part2.cpp part2_tester.cpp -o part2
```

Part I

Implement simple functions on vector and list

Read the comments in `part1.h` and then in the file `part1.cpp`, implement the functions `max`, `min` and `count_values` that operate on `list` and `vector`.

When you've finished your implementation, the output of the test program should be:

```
test_vector passed.
test_list passed.
Passed: 2
```

Part II

In part II, you will implement the bubble sort algorithm on both the vector and the list. The pseudo code for bubble sort is shown below[1]:

```
procedure bubbleSort( A : list of sortable items )
   n = length(A)
   repeat
     swapped = false
     for i = 1 to n-1 inclusive do
       /* if this pair is out of order */
       if A[i-1] > A[i] then
         /* swap them and remember something changed */
         swap( A[i-1], A[i] )
         swapped = true
       end if
     end for
   until not swapped
end procedure
```

The implementation of this algorithm for `vector` is relatively straight-forward. You will need to think more carefully when implementing it for `list`, since you don't have the [] operator and instead must use iterators. Your instructor's solution used two iterators.

When your implementation is complete, the output of `part2_tester.cpp` should be:

```
test_vector_bubblesort passed.

test_list_bubblesort passed.

Passed: 4
```

Part III

Complete the A3 quiz on conneX.

Submission

Submit your `part1.cpp` and `part2.cpp` using conneX. Complete the A3 quiz on conneX.

As usual, it is acceptable (and encouraged) for you to talk about your assignment with your classmates, and you are encouraged to design solutions together, but each student must implement their own solution. Plagiarism detection software will be run on all assignment submissions.

1 Retrieved from Wikipedia on Oct 18, 2015: http://en.wikipedia.org/wiki/Bubble_sort

Grading

If you submit something that does not compile, you will receive a grade of 0 for the assignment. It is your responsibility to make sure you submit the correct files.

Part I

Requirement	Marks
Your code passes the test cases in `part1_tester.cpp`	Up to 2

Part II

Requirement	Marks
Your code passes the test cases in `part2_tester.cpp`	Up to 4

Part III

Requirement	Marks
You answer the questions in the conneX quiz correctly	Up to 4

Total 10