

## 1 Introduction

This lab is a series of exercises to get you back in the swing of C++ and Data Structures programming. You should have seen all (or nearly all) of the material in COSC 220.

## 2 Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Lab01, put each programming exercise into its own subdirectory of this directory, zip the entire Lab01 directory up into the file Lab01.zip, and then submit this zip file to Lab #1.

Make sure that you:

- Follow the coding and documentation standards for the course as published in the MyClasses page for the class.
- Make sure that each project includes a makefile that will compile the program on the Linux HPCL lab machines.
- Check the contents of the zip file before uploading it. Make sure all the files are included.
- Make sure that the file was submitted correctly to MyCLasses.

## 3 Programming Exercises

1. For this program you will be asking the user for the size of an array, creating the array dynamically, filling the array with random numbers, sorting the array with a templated sort function, time the sorting process, and output the result of another templated function that determines if the array is sorted. The specific specifications are as follows.

- Ask the user for the size of the array.
- Create an dynamically allocated array of integers of the user specified size.
- Populate the array with integers between -1000 and 1000 inclusively.
- Create a templated sorting function that does either the Bubble Sort, Insertion Sort, or Selection Sort on an array of any data type, in ascending order. You may assume that the templated type has overloads for the relational operators. The prototype for the function is to be above the main and the implementation for the function is to below the main. In other words, the top function is to be the main. Try to write the sorting algorithm without looking it up, but if you need some help take a look at the course materials.
- Put a timer on the process. There are a lot of ways to do this but in this class we will use the chrono library as it has a nice, high-resolution timer. First, include the library at the top of the program with

```
#include <chrono>
```

Then include another namespace for convenience,

```
using namespace std::chrono;
```

Finally, in your code put the timer around the process to be times (in this case it is a single call to the sort function) using the following code.

```
auto start = high_resolution_clock::now();
// Process to be timed.
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
cout << "Time to sort: " << duration.count() / 1000000.0 << " seconds" << endl;
```

Or using the actual data types and not relying on `auto` you would have.

```
high_resolution_clock::time_point start = high_resolution_clock::now();
// Process to be timed.
high_resolution_clock::time_point stop = high_resolution_clock::now();
duration<double> duration = duration_cast<microseconds>(stop - start);
cout << "Time to sort: " << duration.count() / 1000000.0 << " seconds" << endl;
```

- Create a templated function (as always, prototype at the top and implementation below the main) that determines if an array is sorted. The output of this function should be a boolean, true if the array is sorted in ascending order and false if not. You may assume that the templated type has overloads for the relational operators.
- Make sure you run valgrind on your program to test for memory leaks. For example, if the program name is `Lab01_P1` the following command will do the run.

```
$ valgrind ./Lab01_P1
==36830== Memcheck, a memory error detector
==36830== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==36830== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==36830== Command: ./Lab01_P1
==36830==
Input the size of the array: 10000
Time to sort: 2.98109 seconds
Array is sorted.
==36830==
==36830== HEAP SUMMARY:
==36830==     in use at exit: 0 bytes in 0 blocks
==36830==   total heap usage: 4 allocs, 4 frees, 114,752 bytes allocated
==36830==
==36830== All heap blocks were freed -- no leaks are possible
==36830==
==36830== For lists of detected and suppressed errors, rerun with: -s
==36830== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

A run of the program should simply output the time the sort took and if the array is sorted or not. For example.

```
Input the size of the array: 10000
Time to sort: 2.98109 seconds
Array is sorted.
```

When you are testing you will probably want to print the array to make sure that your population and sorting is working correctly.

2. In a new project copy over the file you used for the last exercise and make the following changes.
  - Change the array type from `int` to `double`.
  - Change the population code to populate the array with random doubles between 0 and 1, specifically in the interval  $[0, 1]$ .

Make no other changes to the program. Test the program and if you need to make other changes go back to the previous exercise and make the changes there as well. Your solution for the previous exercise should also run this one with only the two above changes made.

3. Create a class structure that has two integer private data members named A and B. Include the following member functions for the class. As with all non-templated classes the structure should be in to files, a guarded .h for the specification and a .cpp file for the implementation. There is to be no in-line code for the specification, all implementation code is to be in the cpp file.
  - Constructor that takes in two integer parameters, the first to be assigned to A and the second to B. The value of B cannot be 0 so if 0 is input for B then the program should change it to 1. Also include default values for the arguments, 0 for A and 1 for B.

- Destructor, this is not technically needed here since our member variables are not dynamic, but we will include one anyway.
- getA returns the value of A.
- getB returns the value of A.
- setA sets the value of A.
- setB sets the value of B. Again, B cannot be 0 so change a 0 input to 1.
- set takes two integer parameters, first for A and second for B. Do the same error checking on B as above.
- Overlaoded == The == operator is defined as follows.  $X == Y$  means that  $X.A * Y.B == X.B * Y.A$ .
- Overlaoded != The != operator is defined as follows.  $X != Y$  means that  $X.A * Y.B != X.B * Y.A$ .
- Overlaoded < The < operator is defined as follows.  $X < Y$  means that  $X.A * Y.B < X.B * Y.A$ .
- Overlaoded > The > operator is defined as follows.  $X > Y$  means that  $X.A * Y.B > X.B * Y.A$ .
- Overlaoded <= The <= operator is defined as follows.  $X <= Y$  means that  $X.A * Y.B <= X.B * Y.A$ .
- Overlaoded >= The >= operator is defined as follows.  $X >= Y$  means that  $X.A * Y.B >= X.B * Y.A$ .
- Overlaoded stream out operator. The output here should put the data items A and B into an ordered pair with parentheses, for example if A is 2 and B is 3 the output would be (2, 3).

Write a main that will test all of the functionality of the class. For example, the following main will produce the following output.

```
#include <iostream>
#include "Thing.h"

using namespace std;

int main() {
    Thing t1(2, 3);
    Thing t2(1, 3);
    Thing t3(1, 2);
    Thing t4(2, 4);

    cout << t1 << endl;
    cout << t2 << endl;
    cout << t3 << endl;

    cout << (t1 < t2) << endl;
    cout << (t1 > t2) << endl;
    cout << (t1 <= t2) << endl;
    cout << (t1 >= t2) << endl;
    cout << (t1 == t2) << endl;
    cout << (t1 != t2) << endl;
    cout << (t3 == t4) << endl;

    cout << t1.getA() << " " << t1.getB() << endl;

    t1.setA(5);
    t1.setB(7);

    cout << t1 << endl;
    cout << t1.getA() << " " << t1.getB() << endl;

    t1.set(10, 17);
    cout << t1 << endl;

    return 0;
}
```

Output:

```
(2, 3)
(1, 3)
(1, 2)
0
1
0
1
0
1
1
2 3
(5, 7)
5 7
(10, 17)
```

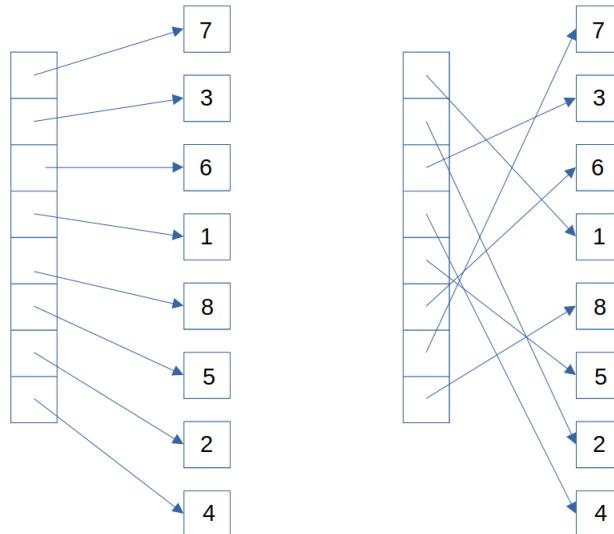
4. In a new project copy over the file you used for the first exercise and the Thing.h and Thing.cpp files you used from the previous exercise. Then make the following changes to the main program.

- Include the Thing.h file at the top.
- Change the array type from int to Thing.
- Change the population code to populate the array with random sets of two integers each. You can just use rand() here for each integer value.

Make no other changes to the program. Test the program and if you need to make other changes go back to the first two exercises and make the changes there as well. Your solution for the first exercise should also run this one with only the three above changes made.

5. This is yet another sorting exercise but this time you will be sorting the array by moving pointers and not moving data items. Again, copy over your solution from the first exercise to this one. Here are the changes to make,

- Change the array from holding integers to an array holding pointers to single integers.
- Populate the array with new integers, each holding a random number between 1 and 10000 inclusively.
- Revise the sort function to take in an array of pointers to of the templated type.
- In the sort function, instead of interchanging values change only the pointer to the value. For example, in the image below, your array is on the left with pointers to data times on the right. The left image is the original population of the array. After the sort, the data has not changed nor has its position in memory. The only thing that is updated are the pointers from your array to the dynamically allocated positions of the data.



- Make the same alteration to the sorted function so that you are checking the data relationships and not the pointer relationships.
  - Make sure that you clear all allocated memory before the program ends. Run valgrind on your program to check for memory leaks.
6. This one is a bit more involved and deals with inheritance of templated class structures. As you know from COSC 220 there are several ways to handle the file structure with templated classes. The one that does not work is to use strictly separate header and implementation files. While some programmers put an include of the cpp at the end of the header file we will not use this method in this course, we will put the specification and implementation into the header file only. The specification portion will have no inline code unless stated otherwise.
- Create a templated class structure named Thing1. This is essentially going to be a class structure that will encapsulate a doubly linked list. Recall that a doubly linked list is a linked list where each node has two pointers, one called next that points to the next node and one called previous that points to the previous node. The list node itself will be templated for storing a data type of any type. We will assume that this data type has all the needed functionality, such as copy constructors, overloaded assignment, relational overloads, etc. The list node could be a struct or a class and it could be an inner class or separate. we will use a separate class structure for this exercise.
    - In the Thing1.h file create a templated ListNode class that stores a value of templated type and two pointers, next and previous to ListNode objects. In this class all data is to be public. There is only one function, the constructor, that will set the two pointers to nullptr. This may be written inline.
    - Create the Thing1 class structure, with only two data items that are to be visible to the class itself and any derived classes of Thing1. The data items are pointers to ListNode types named head and tail. In general head will always point to the head node of the list and tail will point to the last node in the list. If the list is empty then both of these will be nullptr.
    - Create the following functions with the stated functionality.
      - \* Constructor, default only, sets the pointers to nullptr.
      - \* Destructor, obviously removes the list items from memory so that we have no memory leaks. As always, valgrind is your friend.
      - \* pushback takes a parameter of the templated type, creates a new node storing that value and adjoins it to the end of the list.
      - \* popback removes the last item from the list and returns its value. In the case where the list is empty the return will be the default value of the data type.
      - \* pushfront takes a parameter of the templated type, creates a new node storing that value and adjoins it to the front of the list.
      - \* popfront removes the first item from the list and returns its value. In the case where the list is empty the return will be the default value of the data type.
      - \* size returns the number of nodes in the list.
      - \* clear removes all the items from the list.
      - \* printforward prints the list to the console in forward order. This also takes a single boolean parameter defaulted to false that determines if the list should be written on one line or on one line per data item. If false the list is written on one line and if true each item gets its own line.
      - \* printbackward prints the list to the console in reverse order. This also takes a single boolean parameter defaulted to false that determines if the list should be written on one line or on one line per data item. If false the list is written on one line and if true each item gets its own line.
      - \* find takes an item value as a parameter and returns true if the item is in the list and false if it is not.

- \* findpos takes an item value as a parameter and returns the position of the first node that contains the item and -1 if the item is not in the list.
  - \* isempty returns true if the list is empty and false otherwise.
  - \* getValue takes as a parameter an integer position and returns the data item in that position in the list. As with arrays we assume that the first node is position 0, second node position 1 and so on. If the position is not inside the current list the return will be the default value of the templated data type.
  - \* setValue takes two parameters, an integer position and a data item value. It replaces the value in the position node with the input parameter value if the position is inside the list. If the position is not inside the current list no alteration to the list will be made.
  - \* Optional Extra Credit: Overload the out stream operator << to stream out the contents of the list in forward order.
- Create a templated class structure named Thing2 that inherits Thing1, both specification and implementation of Thing2 is to be in Thing2.h, guarded, with no inline code in the specification. This is essentially going to be a class structure that will add an iterator to the Thing1 linked list class. In practice we would incorporate the iterator into Thing1 but it makes a good exercise to add it on with inheritance. This class should have a single data member which is a pointer to a ListNode type called iter. Create the following functions with the stated functionality.
    - Constructor, default only, sets the pointer to nullptr.
    - Destructor
    - resetIteratorToFront puts the iterator at the head of the list.
    - resetIteratorToBack puts the iterator at the tail of the list.
    - Overloaded prefix and postfix operator ++ that moves the iterator one node down the list. If the iterator is null there is to be no change and if the pointer is at the tail of the list it stays there.
    - Overloaded prefix and postfix operator -- that moves the iterator one node up the list. If the iterator is null there is to be no change and if the pointer is at the head of the list it stays there.
    - Overloaded += that moves the iterator down the list by that number of nodes. If the iterator is null there is to be no change and if the pointer reaches the tail of the list it stays there.
    - Overloaded -= that moves the iterator up the list by that number of nodes. If the iterator is null there is to be no change and if the pointer reaches the head of the list it stays there.
    - get returns the value stored in the node the iterator is pointing to. If the iterator is null the return value is to be the default value of the templated type.
    - insertval takes in an item of templated type, creates a new node from it and inserts it into the list after the current position of the iterator. If the iterator is null the item is not to be inserted.
    - deleteval removes the node that is currently being pointed to by the iterator. If the iterator is null there is no deletion.

Given the following test program the output should be like the output below, with obvious randomized differences. If you chose not to overload the stream out operator you can change statements like cout << list1 << endl; to ones like list1.printforward();.

```
#include <iostream>
#include <cstdlib>
#include <ctime>

#include "Thing1.h"
#include "Thing2.h"

using namespace std;

int main() {
    srand(time(0));
}
```

```
Thing1<int> list1;
Thing2<int> list2;

list1.pushback(5);
list1.pushback(7);
list1.pushback(2);
list1.pushback(10);
list1.pushback(3);
list1.pushback(15);

cout << list1.size() << endl;

list1.printforward();
cout << endl;

list1.printbackward();
cout << endl;

cout << list1.popback() << endl;

list1.printforward();
cout << endl;

cout << list1.popfront() << endl;

list1.printforward();
cout << endl;
cout << list1.size() << endl;

cout << list1.find(10) << " " << list1.findpos(10) << endl;
cout << list1.find(15) << " " << list1.findpos(15) << endl;

cout << list1 << endl;

cout << list1.getValue(0) << endl;
cout << list1.getValue(2) << endl;
cout << list1.getValue(5) << endl;

list1.setValue(2, 123);
cout << list1 << endl;

cout << list1.isempty() << endl;
list1.clear();
cout << list1.isempty() << endl;

for (int i = 1; i < 10; i++)
    list2.pushback(rand() % 100);

cout << list2 << endl;

list2.resetIteratorToFront();
cout << list2.get() << endl;
list2++;
cout << list2.get() << endl;

list2.resetIteratorToBack();
cout << list2.get() << endl;
list2--;
cout << list2.get() << endl;
list2--;
cout << list2.get() << endl;
++list2;
cout << list2.get() << endl;
list2 -= 3;
cout << list2.get() << endl;
list2 += 2;
cout << list2.get() << endl;
list2 += 12345;
cout << list2.get() << endl;

list2.insertval(123);
```

```
cout << list2 << endl;
list2 -= 3;
list2.insertval(-25);
cout << list2 << endl;

list2.resetIteratorToFront();
list2 += 5;
cout << list2.get() << endl;

list2.deleteval();
cout << list2 << endl;
cout << list2.get() << endl;

return 0;
}
```

Output: Of course, your numbers may vary since the population was randomized.

```
6
5 7 2 10 3 15
15 3 10 2 7 5
15
5 7 2 10 3
5
7 2 10 3
4
1 2
0 -1
7 2 10 3
7
10
0
7 2 123 3
0
1
54 85 73 17 48 42 92 85 18
54
85
18
85
92
85
48
92
18
54 85 73 17 48 42 92 85 18 123
54 85 73 17 48 42 -25 92 85 18 123
42
54 85 73 17 48 -25 92 85 18 123
-25
```