CS264 Laboratory Session 3

John McDonald

13^{th} October 2015

Deadline: All solutions to be submitted by 5pm Tuesday 20^{th} October 2015.

1 Lab objectives

In this lab you will continue on from last week by solving (slightly) more complex problems in C++. In particular, you will be required to write programs that make use of the features of C++ that we covered since the last lab.

Learning Outcomes
Having completed this lab you should have a thorough understanding of the dynamic memory management facilities provided by C++. In particular you will have experience of writing programs that take advantage of the following operators:`new`,`new[]`,`delete`, and `delete[]`.

2 Questions:

For each of the problems given below write a C++ program that provides a solution. Each box provides a filename to use (or in certain cases multiple filenames). Please ensure that you use those filenames. Failure to do so can result in loss of marks.

Step 0.1: Note that now that you may be working on multiple cloned versions of your repository (e.g. in the lab and at home or on your laptop), it is important to ensure that you keep each in sync. Remember that in order to push changes from your working directory to the repository you must (i) execute `git add` on each changed or added file to add the changes to the *staging area*, (ii) commit them to the local repository using `git commit -m '<commit log message>'`, and (iii) push the changes to the remote repository using `git push`.

In order to bring a local repository up to date with your remote repository you should use `git pull`.

For this week's exercises you should create a sub-directory off of the top level directory called `Lab3`. All your work should go into that directory. Be sure to commit at least after each exercise with appropriate commit messages.

Remember:

- **Add source files to your git repository and commit changes regularly.**
- **All commits should be accompanied by messages that would allow a lecturer or demonstrator to understand the purpose of the commit.**

- **Comment your code.**
- **Use proper indentation for function and control structures.**

Exercise 1.1: Write a program which allows the user to input an sequence of doubles and then prints that sequence in reverse. The length of the sequence should be specified by the user *at run-time*. Hence, initially the program should ask the user to input how many numbers the sequence will contain.

You should save the source in a file called `exercise1.cpp`.

Exercise 1.2: Now write a program that extends the functionality of the above program such that it keeps reading the sequence until it reads -1. That is, the program should not intially ask for the length of the sequence.

Hint: In designing a solution to this problem you will need a list that can keep growing arbitrarily. This can be achieved by allocating an array of a fixed length and then reading numbers until the array becomes full. At this point a new array can be allocated which has a size of the original array plus some fixed amount. Once allocated, the contents of the first array should be copied to the second array (at which point the first array should be freed).To copy the contents of one array to the other you can either implement your own routine or use `memcpy` (full detail of which can be found on cplusplus.com).

You should save the source in a file called `exercise2.cpp`.

Exercise 1.3: Write a program, using struct's, that reads in the name and age of a list of people, and again, prints them out in reverse. The program should store the above details using an array of structs. Initially the program should ask the user for the number of people that will be input.

You should save the source in a file called `exercise3.cpp`.

Exercise 1.4: At the end of the lecture slides on struct's, there is an example of creating a static link list that stores three peoples' names. Copy this program to a file called `exercise4.cpp` and compile and run it to verify it works.

Exercise 1.5: Now take a look at the last slide from the section of the notes on functions. In that section you will see how it is possible to separate out particular sets of routines into a separate implentation and header file which can be compiled independently to the main program. Following this approach separate out the example struct's program such that the program source consists of a file called `customers.h`, `customers.cpp`, and `exercise_1_5.cpp`.

Each of the files should contain the following:
`customers.h`: should contain the definition of the customer structure and the *declaration* of `print_customers`.
`customers.cpp`: should contain the implementation (or *definition*) for `print_customers`.
`exercise5.cpp`: should contain an include of `customers.h` and the main program.

Exercise 1.6: Finally, using the above set of files as a base, extend functionality of the customers api such that it provides all of the functionality necessary to construct a linked list of people to input by the user. To do this you should add the following routines:

```
customer *create_list(string name);
void insert_name(customer* head, string name);
int list_length(customer* head);
void print_customers(customer *head);
```

Demonstrate the functionality by writing a program which allows the user to input an aribtrary number of customers. The program should store these customers in a linked list. The program should keep inputting customers until the user inputs a user with the name `end` at which point all the user details should be printed to the screen.

You should save the source in a file called `exercise6.cpp`.