Introduction to C++ Programming

Day 1 Exercises

Getting Started

1. **You will need to boot the PC into Linux for this course.** You must do this by watching the boot sequence and entering `2` when offered the choice of Windows or Linux.
2. Log in to Linux using your normal username and password. Let the instructor know if this doesn't work.
3. Start up a **terminal window**. You will find the Terminal application listed on the System menu, which you can access by clicking on the button at the bottom-left of the screen.
4. In the terminal window, create a directory to hold the materials for this course by entering the following command:

   ```
   mkdir cpp
   ```

5. In a web browser, visit `http://vlebb.leeds.ac.uk` and log in with your normal ISS username and password, then click on the *Introduction to C++* link, which should be visible under the heading 'My Organisations'.
6. Find the Day 1 area via the *Course Materials* link on the left-hand menu. Right-click on the link to the Zip archive `day1.zip`, choose *Save As* and save the archive to the `cpp` directory that you created earlier.
7. In a terminal window, enter the following commands, one after the other:

   ```
   cd cpp
   unzip day1.zip -d day1
   cd day1
   ```

   You should now be in a directory called `day1`. Enter `ls` to list the files in this directory. You should see a number of files with names ending in `.cpp` listed.

Exercise 1: Editing a Program

1. Start up a text editor—for example, the *Kate* editor. You can find this or other editors in the application menus that you saw earlier.
2. In the text editor window, enter the source code for the "Hello World" program:

```
#include <iostream>

using namespace std;

int main()
{
  cout << "Hello World!" << endl;

  return 0;
}
```

   Save this code to a file `hello.cpp` in your `day1` directory.

Exercise 2: Compiling & Running a Program

1. In the terminal window, compile your program from Exercise 1 with

   ```
   g++ hello.cpp -o hello
   ```

2. Execute the program in the terminal window by entering

   ```
   ./hello
   ```

   Congratulations: you've just written and run your first C++ program!

Exercise 3: Data Representations

1. Using commands equivalent to those of Exercise 2, compile the program `ranges.cpp` and run it.
2. Do the same for the program `sizes.cpp`.
3. Based on the results of running these programs, does the computer you are using have a 32-bit CPU or a 64-bit CPU?

Exercise 4: Integer Division

1. Examine `division.cpp` in a text editor, then compile and run the program. Try inputs of 10 and 2, then 10 and 3, then 1 and 2. Notice how the fractional part of the result is discarded, leaving only the integer part. Newcomers to C++ (or other languages that behave similarly) often find this surprising.
2. Run the program again, this time using inputs of 1 and 0. What happens?

Exercise 5: Infinite Loops

1. Take a look at `loop.cpp` in a text editor. The `while` loop in this program never terminates, because the expression that it evaluates to decide whether the loop should continue is always true.
2. Compile `loop.cpp` and run the program. The word "Help" will scroll endlessly up the screen. You can stop this (or any other) infinite loop by pressing `Ctrl+C` (The `Ctrl` key, together with the `C` key) in the terminal window that is running the program.

Exercise 6: Compiler Errors & Warnings

1. Take a look at the program `mean.cpp` in a text editor. Compile and run it to see how it behaves.
2. Try removing a semi-colon from the end of one of the program statements inside the `main` function. Try to compile the program and see if you can relate the compiler error message to the error that you introduced.

   Pay particular attention to line numbers in compiler error messages. You should start looking for the error on this line and then work your way backwards through the code until you find the problem.
3. Comment out line 19 (the one with the variable definition `double value`) by putting two forward slashes (`//`) at the start of that line. Save the file and try to compile it. Once again, see if you can relate the error message to the error.
4. Add the variable definition `int x;` somewhere in the code—e.g., on line 22. Try compiling the program without the `-Wall` option, then again with the `-Wall` option.

Exercise 7: Debugging a Program

The program in `mean2.cpp` is similar to that in `mean.cpp`, except that it allows you to input any number of non-negative values. As soon as you enter a negative value, the program computes the mean of all the numbers input so far (excluding the negative value). There is a problem with the program, and we are going to identify what is wrong using GDB, the GNU Debugger.

1. Compile the program using the command

   ```
   g++ -Wall -g mean2.cpp -o mean2
   ```

   Then run it to see how it behaves.
2. Load the program into the debugger by entering `gdb mean2` at the Cygwin terminal prompt. When the `(gdb)` prompt appears, enter the command `run`. The program should behave just as it did outside the debugger.

3. We need to step through the program a line at a time to figure out what is happening. Enter the `list` command to view a section of the program source code, then enter `list` a second time. (To save on typing, you can recall the previous command with the up arrow, or abbreviate it to `l`.) You can see that interesting things don't start to happen until line 25, just inside the program's `while` loop. Set a **breakpoint** here by entering the command `break 25`.

4. Enter the `run` command once more. This time, execution halts at the breakpoint that you've just specified. You can list the currently active breakpoints at any time using `info breakpoints`. You can remove a breakpoint at a particular line of the program by entering `clear`, followed by the program line number.

   Try adding a second breakpoint outside the loop, at line 34. Then try listing your breakpoints with `info breakpoints`.

5. You can resume execution with `continue`, or **single-step** through the program using `next`. While the program is paused, you can examine the value of any variable using the `print` command. These commands can be abbreviated to `c`, `n` and `p`, respectively.

   Enter `n` to execute the statement `cin >> value;` on line 25. The program is now waiting for input, so enter the value 7.5 and the next statement will be displayed. You can check whether the input worked by entering the command `p value`.

   Now enter `c` and the program will loop back to the breakpoint on line 25 again. This time, enter `n`, input a value of `-1` and enter `c`. The program will leave the loop and halt at the second breakpoint, line 34—the statement that computes the mean. Enter `list` (or `l` for short) to view the surrounding context for this line.

6. Now let us examine the values of the program's variables. Instead of using `print` repeatedly, use the `info locals` command to display the current values of *all* local variables. Notice how the `numValues` variable is zero, despite the fact that we have successfully entered a value. Can you spot the error in the program now? Enter `quit` to leave the debugger, then go back to the editor and see if you can fix the problem.

Exercise 8: Using a Vector

You saw in `mean2.cpp` how a mean value could be computed by a loop that reads values from the console and accumulates a total. Another approach would be to separate the reading of the data from the computation—i.e., to read in all the data first and then compute the mean. We can do this if we store the data in a vector.

1. Examine `mean3.cpp`. This is meant to be a vector-based version of the program seen previously in `mean2.cpp`. The program is missing three essential lines relating to the vector, at the locations indicated by the TO DO comments.

   Add the missing code, then compile the program in the usual way and run it, comparing its behaviour on a set of input values with that of `mean2`.

2. You will hopefully have noticed that `mean3` is not giving the correct result. There is a subtle bug in the code that computes the mean of the values in the vector. Find and fix the bug, then recompile and test your program once again.

Exercise 9: Writing a Program

The formula for converting a temperature in Fahrenheit, T_F, to a temperature in Celsius, T_C, is

$$T_C = \frac{5(T_F - 32)}{9}$$

The task here is to write a C++ program that creates a Fahrenheit-to-Celsius conversion table for a range of temperatures specified by the user. The program should accept two integers from the user, representing the endpoints of the desired temperature range; it should verify that these temperatures define a valid range (i.e., that the second is greater than the first) and terminate the program with a suitable error message if there is a problem; finally, it should iterate over the specified range in five-degree steps, performing the calculation for each temperature and writing the results as a neat table on the console.

Before beginning, you might like to look again at `mean.cpp`. If you can read this program and understand what is happening on each line, you should have no problems writing the temperature conversion program described above.

1. Use a text editor to create a file named `temptable.cpp` and put at the beginning a **comment block** that starts with `/*` and finishes with `*/` (see `mean.cpp` for an example). Between these delimiters, add some text that explains the purpose of the program. Also include your name and a 'last revised' date.

2. Add to `temptable.cpp` a skeletal `main` function, containing only the statement `return 0;`. Save the file and check that your program compiles before proceeding further.

   Tip: programming in a new and unfamiliar language is *much* less painful if you take small steps, compiling and testing after each step!

3. Copy the `#include` and `using namespace std;` lines from `mean.cpp` to the new program, putting them between the comment block and the `main` function.

4. Add two variable definitions to the start of `main`, to represent the start and end of the temperature range. Then add a `cout` statement to prompt the user for data entry, followed by a `cin` statement that reads values from the console into the two variables. Compile and test your program before proceeding. (You could add a temporary `cout` statement to print out variable values if you want, or you could compile with `-g` and run using the debugger.)

5. Add an `if` statement to do data validation. If the second temperature is not greater than the first, you should print a suitable error message and terminate the program. A suitable way of doing the latter in `main` is another `return` statement, returning a non-zero value. (By convention, C++ programs return 0 to the invoking environment if execution was successful, or a non-zero value if the program failed in some way.)

   Compile your program and test it by trying to input valid and invalid temperature ranges. Make sure it is behaving as required before proceeding.

6. Finally, write a `for` loop that iterates over the specified range of Fahrenheit temperatures. The body of this `for` loop will need to contain an arithmetic expression to compute a temperature in Celsius, using the loop variable as the Fahrenheit temperature. (Watch out for possible loss of precision here ...) You should introduce a new variable of type `double` into the program, to hold the result of the calculation. You will also need to add a `cout` statement that prints the Fahrenheit and Celsius temperatures.

7. Modify the `cout` statement in the loop and add further `cout` statements before and after the loop, so as to create the effect of a table like this:

```
+--------------+-------------+
|  Fahrenheit  |   Celsius   |
+--------------+-------------+
|       45     |       7.2   |
|       50     |      10.0   |
|       55     |      12.8   |
|       60     |      15.6   |
|       65     |      18.3   |
|       70     |      21.1   |
+--------------+-------------+
```

   In order to format the numbers in the two columns correctly, you will need to use **stream manipulators**. To access these, add `#include <iomanip>` near the top of your program.

   The manipulators you need are `setw`, to set the 'column width' in characters; `fixed`, to specify use of a fixed number of decimal places; and `setprecision`, to specify how many decimal places are required. The following examples show how you could write an integer `n` in a three-character column and a floating-point value `x` in a five-character column using two decimal places:

```
cout << setw(3) << n;
cout << setw(5) << fixed << setprecision(2) << x;
```