Introduction to C++ Programming

Day 2 Exercises

Getting Started

1. Boot your PC into Linux and log in using your normal username and password, then start a terminal window.
2. In a web browser, visit `http://vlebb.leeds.ac.uk` and log in to the VLE with your normal username and password, then click on the *Introduction to C++* link, which can be found under the heading 'My Organisations'.
3. Find the Day 2 area via the *Course Materials* link on the left-hand menu. Download from this area the Zip archive `day2.zip`, saving it to the `cpp` directory that you created yesterday.
4. Enter the following commands in the terminal window:

   ```
   cd cpp
   unzip day2.zip -d day2
   cd day2
   ```

   You should now be in the `day2` directory, ready to work on today's exercises.

Exercise 1: Creating a Vector

1. Use a text editor to write a program called `vector.cpp`, saving it to the `day2` directory. The program should
   - Create a vector of integers
   - Use a `for` loop to put the integers from 0 to 9 in the vector
   - Display the number of values stored in the vector
   - Display the first value stored in the vector, using `[]` or the `at` method
2. Compile and run your program to make sure that it works.

Exercise 2: Creating a List

1. Create a copy of `vector.cpp` from Exercise 1, called `list.cpp`.
2. Replace all occurrences of `vector` with `list`, then attempt to compile `list.cpp`. The command should fail because lists don't support using `[]` or the `at` method to access elements.
3. Comment out the offending line and try again. This time, the program should compile.

Exercise 3: Algorithms

1. Create a copy of `vector.cpp` from Exercise 1, called `sorting.cpp`. Remove the two lines that display the vector size and first element.
2. Add a `#include` for the `algorithm` header.
3. Add a call to the `random_shuffle` algorithm. This function requires two iterators as arguments, defining the region of the vector to be shuffled. If your vector is called `v`, then suitable iterators will be returned by `v.begin()` and `v.end()`. Check that the program still compiles before continuing.
4. Write a `for` loop that prints all of the vector's values on a single line, each preceded by two spaces. Add a `cout << endl;` statement after the loop. Compile and run the program; you should see the integers 0 to 9 displayed in a random order.
5. Add a call to the `sort` algorithm that will sort the values in the vector into ascending order. After this, copy and paste the vector display code written in the previous step. Compile and run your program; you should see another line of output in which the integers are sorted.

6. Add another call to the `sort` algorithm, this time to sort the values into descending order. Copy and paste the vector display code again. Compile the program and check that it runs correctly.

Exercise 4: Reading From a File

We can read from text-based files using file stream objects that behave in exactly the same way as the console input stream object `cin` that you have used in previous exercises.

As an illustration of this, let's take a program from yesterday's exercises and modify it to read values from a file rather than prompting for their input at the keyboard. Instead of requiring input of the special value -1 to signify no more data, we will simply stop when it is no longer possible to read a value from the file.

1. Edit the file `mean4.cpp` and add `#include <fstream>` to the list of headers at the top of the program. Then introduce a new variable called `dataFile`, defined like so:

   ```
   ifstream dataFile("test.data");
   ```

   This **file stream** will represent the file of input data in our program. If a file with the given name can be found, that file will be opened for reading.

2. Remove the `cout` statement that prompts the user to enter data, replacing it with code to test whether opening the file succeeded:

   ```
   if (! dataFile) {
     cerr << "Cannot open data file" << endl;
     return 1;
   }
   ```

3. Now modify the loop that processes input values so that it looks like this:

   ```
   while (dataFile >> value) {
     data.push_back(value);
   }
   ```

   After the loop, add code to close the input file, like so:

   ```
   dataFile.close();
   ```

4. Compile and run the program. It should output the mean of the values in the file `test.data`.

5. Try replacing the `for` loop that sums values in the array with a line of code that uses the `accumulate` algorithm.

Exercise 5: Fun With Functions

1. Copy the `temptable.cpp` program from yesterday's exercises to your Day 2 directory—e.g., with the following command in the terminal window:

   ```
   cp ../day1/temptable.cpp temptable2.cpp
   ```

2. Edit `temptable2.cpp` in a text editor, refactoring it so that it consists of

   - A function `celsiusTemperature` that takes a temperature in Fahrenheit as its sole parameter and returns the corresponding temperature in Celsius as a `double` value;
   - A function `printTable` that takes start and end temperatures in Fahrenheit as parameters and prints the corresponding temperature conversion table. This function should use `celsiusTemperature` to compute each entry in the table.
   - A function `main` that gets start and end temperatures from the user and then calls `printTable`.

3. Compile and run your program to make sure that it behaves exactly the same way as before.

Exercise 6: More Fun With Functions

The program `mean4.cpp` that you created in Exercise 4 has a `main` function that performs two distinctly different tasks: reading data and computing an average value. A cleaner, more modular design would separate these two tasks into distinct functions and then call these functions from `main`.

1. Copy `mean4.cpp` to a new file named `stats.cpp` and edit this new file.

2. Near the top of the file, after the `#include` directives but before `main` itself, create two 'empty' function definitions like this:

   ```
   void readData()
   {
   }

   double mean()
   {
     return 0.0;
   }
   ```

   Unfinished functions like this are known as **stubs**. Check that the program still compiles before proceeding further.

3. Now we need to decide how these functions will communicate with `main` and with each other. A sensible option would be pass a vector between the functions. An empty vector can be created in `main` and then be passed to `readData`, which has the job of filling it with data. The filled vector can then be passed to `mean`, which has the job of iterating over it, computing and then returning the mean value.

   Modify the stubs implemented in the previous step, like so:

   ```
   void readData(istream& input, vector<double>& data)
   {
   }

   double mean(const vector<double>& data)
   {
     return 0.0;
   }
   ```

   Notice how we use reference parameters. This is important! Notice also that the `vector` reference passed to `mean` is declared as `const`.

   Check that the program compiles before proceeding further.

4. Now implement the body of `readData`. **Do not open the file or create the vector here!** These things will happen in `main`. The parameters `input` and `data` represent the opened file and empty vector that `main` creates. All that needs to go in `readData` is the `while` loop that reads values and pushes them into the vector.

5. Implement the body of `mean`. This should simply involve moving the code that computes the mean value into the function. **Do not print the mean value here!** Instead, return the value to the caller. This approach is more flexible, ensuring that the `mean` function can be used in contexts where we need to use the value for something rather than simply printing it.

6. Finally, modify the `main` function so that it calls the `readData` and `mean` functions to do its work. If you've done this correctly, you should end up with a `main` having three parts:

   - A part that creates an `ifstream` object for the `test.data` file and checks whether the file was opened successfully
   - A part that creates an empty `vector`
   - A part that calls `readData` using the previously-created `ifstream` and `vector` objects, calls `mean` using the `vector` object and then prints the returned value

   Run your program and check that it generates the same results as `mean4.cpp`.

Exercise 7: Enhancements

If you finish early, try implementing one or more of the following enhancements to `stats.cpp`.

- If you've not already done so, modify `mean` so that it computes the sum of the vector of values using an STL algorithm. The algorithm you need is called `accumulate`, and it expects a pair of iterators as its first two arguments, plus an initial value on which to base the sum (use 0.0 for this).

  Note: you will need to add `#include <numeric>` to your program's list of headers.

- Add a function to compute the median. This is defined to be the middle value of a sorted list of values if there is an odd number of values, or the average of the two values spanning the midpoint if there is an even number of values. For example, the median of a sorted five-element vector `v` would be `v[2]`, whereas the median of a sorted six-element vector `v` would be `(v[2]+v[3])/2.0`.

  Don't forget that you can use an STL algorithm to do the sorting very easily!

- Add a function to compute the standard deviation, σ. This is defined mathematically as

  $$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)}$$

  where $x_1, x_2, \ldots x_N$ are the values and μ is their mean. In effect, standard deviation is the square-root of the average of the squares of the differences between each value in the vector and the mean of all the values.