Introduction to C++ Programming

Day 3 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 3 area via the *Course Materials* link on the left-hand menu. Download from this area the Zip archive `day3.zip`, saving it in the `cpp` directory that you created on Monday.
4. Enter the following commands in the terminal window:

   ```
   cd cpp
   unzip day3.zip -d day3
   cd day3
   ```

   You should now be in the `day3` directory, ready to work on today's exercises.

Exercise 1: Command Line Arguments

Take the version of `stats.cpp` that we have provided for you and modify it so that the name of the data file is specified as a command line argument.

You will need to check `argc` and terminate the program with a suitable error message if no filename has been supplied on the command line.

Exercise 2: Multi-file Programs

1. Take either your own implementation of `stats.cpp` from yesterday's exercises or the version provided for you in today's files and restructure it as three separate files, as suggested on the Slide 7 of the handout.

   You should end up with a file `data.cpp`, containing the `readData` function; a file `calc.cpp`, containing the `mean` function (and possibly the `median` function as well); and a file `stats.cpp` that now contains only the `main` function.
2. Write a **header file** called `data.hpp` that contains a prototype for the function in `data.cpp`. For example, a suitable prototype might be

   ```
   #include <iostream>
   #include <vector>

   void readData(std::istream&, std::vector<double>&);
   ```

   Note how parameter names have been omitted; they are not necessary in function prototypes.
3. Write a header file called `calc.hpp`, for the statistical calculation function(s) in `calc.cpp`. It should look like this:

   ```
   #include <vector>

   double mean(const std::vector<double>&);
   ```

   If your `calc.cpp` has a `median` function, then you should add a suitable prototype for that, too.
4. Modify `stats.cpp` so that it has `#include` directives for the two header files:

   ```
   #include "data.hpp"
   #include "calc.hpp"
   ```

5. Check that your program still compiles by entering the following command:

```
g++ stats.cpp data.cpp calc.cpp -o stats
```

   If this succeeds, run the program on the data file you created yesterday to check that its behaviour is unchanged.

6. Try recompiling without the `calc.cpp` file:

```
g++ stats.cpp data.cpp -o stats
```

   The error that you see is a **linker error** rather than a compiler error. The linker knows that `main` is calling functions called `mean` and `median`, but it cannot find any object code containing definitions of those functions.

7. Consider the commands you have just typed to compile the program. Imagine that you had made a small change to `data.cpp`. To generate a new executable, you would be compiling three files of C++ source code, even though only one of them had changed! This is seriously inefficient for large, complicated programs. A much better approach is to compile each source file *separately* into object code, and then link the resulting `.o` files together as a final step.

   Try it now, by entering the following commands, in sequence:

```
g++ -c data.cpp
g++ -c calc.cpp
g++ -c stats.cpp
g++ stats.o data.o calc.o -o stats
```

   Verify that the resulting executable still runs correctly.

   With this new approach, if one of the source files changes, that file is the only one that needs to be recompiled into object code. (We will need to repeat the final linking stage too, of course.)

Exercise 3: Makefiles

1. The drawback of the multi-file approach outlined in Exercise 1 is that it requires a lot more typing. We could avoid this by putting all of the commands in a script or batch file, but a better approach is to use a **makefile**. A makefile specifies rules for how a program is to be compiled, and a tool called `make` interprets these rules. `make` is able to figure out whether a particular source file needs to be recompiled by looking at file modification times.

   Take a look at the provided file named `Makefile`. This is a makefile suitable for building the `stats` program. Notice how it has a series of **targets** and **dependencies**. For each target, there is a command to execute if the target is identified as being out of date with respect to any of its dependencies. Each command is indented with a single tab, not with spaces. **This is a strict requirement in makefiles!**

   Notice also how it is possible to have targets that are not executables or files of object code. The `clean` target has no dependencies and simply removes the object code for the files making up the `stats` program; the `veryclean` target will remove the executable as well as the `.o` files. (It achieves the latter via a dependency on the `clean` target.)

2. To try out the makefile, enter the command `make veryclean`. Notice how this runs the `rm` command for the `clean` target first, removing the object code, before removing the executable.

3. Now enter `make` on its own, with no command line arguments. You will see each `g++` command being executed and you should end up with `.o` files for each source file, plus an executable called `stats`.

4. Enter `make` a second time and you should see a message informing you that `stats` is 'up to date'.

5. Make a trivial change to one of the `.cpp` files in a text editor. Save the file, then enter `make` again. You should see recompilation of the file that was changed, followed by linking.

Exercise 4: Creating & Using Libraries

1. Modify the makefile from Exercise 3 so that it has a target to build a library. The following addition should work:

```
libstats.a: data.o calc.o
        ar crv libstats.a data.o calc.o
        ranlib libstats.a
```

   Important: make sure that there is a single tab at the start of each command, not spaces!

2. Modify the `stats` target to look like this:

```
stats: stats.o
        g++ -L. -o stats stats.o -lstats
```

3. Do `make veryclean`, followed by `make`. The program should build as before, only this time `data.o` and `calc.o` are put into a library and `stats.o` is linked with this library to generate the finished executable.

   Take a look at the contents of the library with

```
ar t libstats.a
```

   You should see the two object code files `data.o` and `calc.o` listed.

Exercise 5: Writing a Class Definition

Write the definition (*not* the implementation) of a class `Circle`, in a header file named `circle.hpp`. Give your class definition

- An instance variable `radius`, of type `double`
- A constructor that takes the circle's radius as its only parameter
- A `getRadius` method
- Methods `area` and `circumference`, both of which take no parameters and return a `double` value

Exercise 6: Implementing Methods of a Class

1. Create a file `circle.cpp`, containing implementations of the methods of `Circle` that have been defined in `circle.hpp` in Exercise 4. Your implementation file will need two `#include` directives:

```
#include "circle.hpp"
#include <cmath>
```

   The second of these will allow you to use the constant `M_PI` to represent π.

2. Test your class by attempting to compile it like so:

```
g++ -Wall -c circle.cpp
```

Exercise 7: Enhancing the Circle Class

1. Copy `circle.hpp` to a new file, `circle2.hpp`.
2. In `circle2.hpp`, create an improved version of the `Circle` class in which
   - The class definition appears in a namespace called `geometry`
   - The constructor uses the special instance variable initialisation syntax seen in Slide 31
   - All methods are inlined
   - Accessor methods are declared as `const`
3. Test your enhancements by writing a tiny test program that simply creates a `Circle` object and calls one of its methods. Compile and run your program to make sure that everything works.

Exercise 8: Object-Oriented Temperature Tables

1. Implement a class called `TemperatureTable`, representing a Fahrenheit-to-Celsius temperature conversion table generated by the program you created on Day 1. Your class should consist of a class definition in header file `temp.hpp` and an implementation of the class methods in the file `temp.cpp`. Your class should have

   - Instance variables `start` and `end`, representing the start and end of the temperature range spanned by the table
   - A constructor that takes two parameters, representing the start and end of the temperature range
   - An overloaded `<<` operator that actually displays the table

2. Implement a small test program that contains the following lines:

   ```
   TemperatureTable table(45, 70);
   cout << table;
   ```

   Compile and run the program to make sure that it works.

Exercise 9: Object-Oriented Dataset Analysis

1. Implement a class called `Dataset` containing all the functionality of the program `stats.cpp` written for yesterday's exercises. Your class should consist of a definition in header file `dataset.hpp` and an implementation in `dataset.cpp`. Your class should have

   - A single instance variable called `data` which is a vector of `double` values
   - A constructor that takes an `istream` object as its only parameter and reads values from this input stream, pushing them onto the back of the vector `data`
   - A `const` method `mean` that has no parameters and that returns the mean of all the values stored in the vector `data`
   - A `const` method `median` that has no parameters and that returns the median of all the values stored in the vector `data`

   You can either use your own version of `stats.cpp` as the starting point, or the version provided for you as one of today's source files.

2. Implement a small test program that creates a `Dataset` object and calls its methods. Compile and run the program to make sure that it works.