

G52OBJ Lab Exercises

Steven Bagley

1. Introduction

The purpose of these exercises is to get you familiar with writing and compiling C++ programs on the school UNIX computers and then to start looking at how a date class would be implemented in C++. If you are already familiar with some of the topics covered by an exercise, then feel free to move onto the next. If you feel you want more information about C++, then I've uploaded the PRG notes from a few years back which provide a good introduction to C++.

All the files used in these exercises are available on the G52OBJ website (<http://www.eprg.org/G52OBJ/>) as a gzipped tar file, download this into your UNIX home directory (available in Windows via the H:\ drive) and extract the files from the archive. This can be done with the following commands:

```
gunzip lab-exercises1.tar.gz
tar xvf lab-exercises1.tar
```

The first command decompresses the file so that the second can extract the files from it. This will result in a directory called `g52obj` being created in the current directory containing the example files.

We shall be using the GNU C++ compiler during the course to compile our source code. However, we shall be making use of a few other UNIX tools to ease the task but we shall look at them in more detail later. For now, it is important to get to now the compiler.

2. The Compiler

The GNU C++ compiler is a UNIX command line tool that is accessed by the `g++` command like this:

```
g++ hello.cpp
```

This will compile the file `hello.cpp` into a UNIX executable called `a.out` (this is a unix compiler convention). This isn't very convenient so we tend to use the `-o` flag to specify the output filename. In the directory `helloworld`, you should find `hello.cpp` a C++ implementation of the classic hello world program. Change into this directory and compile it using the following command:

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

This will create an executable file in the directory called `helloworld` which you can execute to run the program.

3. First C++ program

Let us take a closer look at the hello world program. Open `hello.cpp` in your editor of choice. You should see the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    printf("Hello World\n");
}
```

Immediately, it should be evident that there are two sections in the file. The second section should look similar to the Java you have programmed before but the first section needs a bit of explanation. In C and C++, it is necessary to declare (or tell the compiler about) something before you use it. This goes for both classes and functions. To save us having to type the declarations in each file that uses them we generally stick them in another file (called a *header file* and given the extension `.h`) and include them into each file as necessary using the `#include` operator. Generally, inclusions fall into two categories, standard libraries such as those we are including here and our own header files (containing declarations of our classes etc.). So this section:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

includes three of the standard C library header files. `stdio.h` contains I/O routines (such as `printf` which prints things out), `stdlib.h` and `string.h` which contains string handling functions.

In C/C++, all programs begin executing with the procedure `main()` and this is defined in the second half of our example. This procedure takes two arguments an integer `argc` and an array of strings `argv` (in C/C++, strings are represented by the type `char *`). `argc` is necessary because arrays in C/C++ are unbounded and so it is necessary to pass the size of the array to code that use it. Although, we won't need to consider it during G52OBJ, `argv` is used to pass in arguments from the command line to your programs.

Looking inside the `main()` procedure, we find the code it executes. In this example, there is a solitary call to `printf()`, a very powerful library procedure that enables to print output in a formatted fashion. There are also variations that output into strings and to files.

4. Your First Class

C++'s heritage means that it can operate as both an object-oriented language and as a procedural language. Since this is an Object-Oriented course, we are going to start implementing classes and using objects. This exercise looks at implementing a class to store dates, similar to the one we saw in the second lecture.

If you look in the `dates` directory extracted earlier, you should find the skeleton code for our next exercise. This consists of four files:

```
date.cpp
date.h
test.cpp
```

These files contain the implementation of the class (`date.cpp` and `date.h`) and a small program to test it (`test.cpp`). The code for the test program is provided and if you look at it, it should be pretty evident what it does. We'll come back to it in more detail later but for now we are going to implement the code for the class itself.

It is the convention that C++ classes are split into two halves. The first half declares the class (its interface and any member variables it uses) and is stored in a header file (so that it can be included by all the other classes that utilise it). The second half contains the implementation and this is placed in the source file (with a `.cpp` extension).

4.1. Defining the class

To start implementing our date class, we need to declare the class in the header file in `date.h`. If you look in `date.h` prototype provided, you will see that it is virtually empty apart from the following:

```
#ifndef __DATE_H__
#define __DATE_H__

/* Insert class here */

#endif
```

The reason for these lines is simple. It is sometimes possible for header files to be included more than once which causes the compiler to throw an error about things being redefined. To stop this, we place lines like those above to check whether a particular compiler variable has **not** been defined. If it hasn't then the compiler variable is defined and the rest of the header file processed by the compiler. Otherwise the header file is skipped (until the `#endif` directive). You'll need to include lines like these in every header file you create. If you don't then, eventually, you'll start getting errors occurring. It needs to be unique, so using the name of the header file in capital letters with underscores around it is a good idea. If you were creating a class called `CTime` in files called `time.h` and `time.cpp`, then you would probably use `__TIME_H__` as the compiler variable to give this:

```
#ifndef __TIME_H__
#define __TIME_H__

...

#endif
```

Time to look at building this class. Open `date.h` in a text editor, and position the cursor after the comment 'Insert class here'. We define a class using the keyword `class` followed by the name of the class in this case `CDate` followed by the classes methods and variables enclosed in braces like this:

```
class CDate
{
public:
    CDate(int day, int month, int year);

    int Day();
    int Month();
    int Year();
    void SetDate(int day, int month,
                 int year);

    char* ToString();
private:
    int m_day;
    int m_month;
    int m_year;
};
```

As you can see, this is fairly similar to the example given in the second lecture so you can refer to the notes there for what the methods are intended to do. Copy the above code into the open `date.h` header file and save it to disk.

The `public` and `private` keywords define everything following them as either public or private to the class. They can occur multiple times but it is easier for users of your class if you group the public methods at the top.

4.2. Implementing the class

Now that the header file is written we can start implementing the class. The implementation goes into the `date.cpp` file. If you open this up, you will see that it is virtually blank except for the inclusion of some standard header files and the line

```
#include "date.h"
```

to include the class definition. This must be present or the file will not compile. Let's make a start by implementing the constructor. This is done in exactly the same way as we saw `main()` was implemented earlier, except for two differences. First, we need to tell the compiler that this is a method on a class and not a floating procedure. We do this by prefixing the name of the method with the name of the class joined together by two colons like this:

```
CDate::CDate(int day, int month, int year)
{
}
}
```

As in Java, the code goes between the braces. The second difference is that constructors do not have a return type specified at all. Okay, copy the above into the source file and implement the constructor. Remember that it needs to check that the values passed in are valid and then store them in the member variables we declared in the class (e.g. `m_day`).

Once you have implemented the compiler, it is probably a good idea to compile it to check it is correct. We can do this by executing the command

```
g++ -c date.cpp
```

Hopefully, it should return without giving any error messages. If it does, now would be a good idea to go and correct any errors in your code. You'll have noticed that we are using a different command to compile our file and in particular we are using the `-c` flag. This flag tells the compiler just to compile the code but not to make it into an executable file. Instead, it produces what is called an object file (not to be confused with objects in an OO-sense). We do this for every `.cpp` file, and then we link them together at the end to produce the final executable. This speeds up development since it means we only need to recompile the files that have changed. This can be automated using a tool such as `make`.

Now it is time to complete the rest of the class. With the exception of `Tostring()` most of the methods are trivial and you should be able to implement them with ease. To get you started, the `Day()` method should be implemented like this:

```
int CDate::Day()
{
    return m_day;
}
```

Remember to periodically check your code as you add to it, otherwise you'll find yourself facing hundreds of errors to correct at the end...

Finally, it is necessary to implement `Tostring()` as follows:

```
char *CDate::ToString()
{
    char *retVal = (char *)malloc(32); /* Allocate 32 bytes to store our string */

    sprintf(retVal, "%d/%d/%d", m_day, m_month, m_year);
    return retVal;
}
```

The first thing this method does is allocate some memory to store the string in using `malloc` (memory allocate). We tell it we want 32 bytes, and it returns us a pointer to that block of memory. Note that when allocating raw memory, we need to cast it to the correct type, in this case `char *`. We then use the string

version of `printf` called `sprintf` to print the date into the string. Note how `printf` formats the date for us, interpolating the numerical values of the variables into the string where `%d` is found. We can also use `%s` for strings, and `%f` for floating point numbers here, but ensure you have the variables in the right order after the formatting string or it'll crash.

That is the `CDate` class implemented, so we can now turn our attention to the test code and see if the whole system works.

4.3. Testing the class

First of all, compile `date.cpp` into an object file to ensure that there are no errors in the code. Assuming that compiles correctly, compile `datetest.cpp` in the usual manner. This should leave you with two object files in the current directory: `date.o` and `datetest.o`. To build an executable file that we can run, we need to link these two object files together along with the standard C and C++ libraries using the following command:

```
g++ -o datetest datetest.o date.o
```

The `-o` flag tells the compiler then name of the executable and we then give it the path to all the object files that need to be linked together. You can now run the `datetest` executable that should be generated, assuming you've got everything correct this should produce the following output:

```
$ ./datetest
aDate set to 23/11/1963.
aDate set to 28/2/2008.
anotherDate->Day() returns 12
anotherDate->Month() returns 12
anotherDate->Year() returns 2012
```

There you have it, your first working OO C++ program! To finish let us take a closer look at the code that is using our objects. Open `datetest.cpp` in your editor. The first thing you should notice is the inclusion of the standard libraries as we have seen before. This is followed by:

```
#include "date.h"
```

This includes the class definition for `CDate` which lets the compiler know what methods are available. Looking inside `main()`, we see the code that uses the object. This should look familiar to Java with the exception that methods are called using `->` rather than using the `.` character. This is because C++ distinguishes between two methods of creating object, either directly or via reference to an object somewhere in memory which you reference by a pointer to its location in memory. The `->` operator means dereference this pointer and call the method on the object you find there. Java only supports objects in memory and so only needs one type operator to call methods. The comments should help you follow what the code does.

The final part of the code to highlight is at the end of `main()`:

```
/* delete the objects that we have finished with */
delete aDate;
delete anotherDate;
```

In C++, we have to explicitly delete objects when we have finished with them otherwise they just lie around using up memory. This is done with the `delete` operator followed by the pointer to the object. So,

```
delete aDate;
```

deletes the object pointed to by `aDate`. A word of warning though, `delete` doesn't clear the variable holding the pointer to the object and if you attempt to call a method on the object then result is undefined...

5. Conclusion

That is the end of the first set of exercises and are basically a brief introduction to compiling C++ and implementing objects in C++. The next set of exercises will build on these examples as we create objects that implement the **Decorator** pattern for coffee shops outlined in lecture seven.