PH30056 Comp Phys B – Monday 1 February 2021
Introduction to C++, Visual Studio 2017, and openGL

Visual Studio 2017

Here is an alternative, for example when using your own computer with an internet connection:

To run visual studio on university servers remotely via UniDesk: Go to https://www.bath.ac.uk/guides/working-from-any-location-with-your-own-device/ and follow the instructions for connecting to uniapps, you should get a window with a list of applications. One of them is Visual Studio. If you click on that, it should open a remote desktop that looks much like the PC lab desktop. This allows desktop applications to be run remotely, it works on any PC (even off-campus). Remember to save copies of all files also on your own hard-drive!

On UniDesk, you should find a program called Visual Studio 2017 under UniApps in the start menu. This will connect you to a well tested version of Visual Studio on a central university server. When you open it for the first time, it will ask you to set a default environment, I suggest you choose Visual C++ Development Settings, or something similar to that. It may need time for some housekeeping when you start, please be patient.

If you would like to work on your own computers, you have two additional options:

(1) If you are running Windows, you can download Visual Studio Community Edition for free from Microsoft “https://www.visualstudio.com/downloads/” and run it on your machine. When installing, Visual Studio will ask which Workloads to install. Here, I chose "Desktop development with C++". Note that you’ll need few GB of hard drive space. Normally, you can get a free 30 day license, but by adding your University of Bath account (username@bath.ac.uk) you can use the university license to Visual Studio. There is also something called Visual Studio Code which runs on windows/mac/linux, I haven’t tried it but you might want to give it a go.

(2) If you are using a mac or linux, it is possible to run C++ programs from the Terminal. This is similar to what was done in the 2nd year C-programming course but the compiler is called g++ (and not gcc). However, if you want to run the programs in this course, you will also need to install Xquartz (for graphics) and you will need to access some libraries to make the graphics work properly. See Moodle page for detailed instructions on how to make the code run on the server linux.bath.ac.uk or a mac.

To display windows while running any program on linux.bath.ac.uk you will need to use a terminal that support X-windows forwarding. For ssh from a command line, you can run “ssh -X” to enable X-windows forwarding. If you would like to run from a terminal on a windows PC, I use MobaXterm https://mobaxterm.mobatek.net/ but there are other excellent options, including Xming. Once you install MobaXterm, click “Session” and “SSH,” enter “linux.bath.ac.uk” for remote host (and you can enter your username here). Also make sure X11-forwarding is enabled under “Advanced SSH settings.” To save and transfer files on linux.bath.ac.uk, you can either download directly from Moodle onto your H-Drive (the command “firefox &” opens a browser window). Alternatively, you can set up a separate SFTP session (or use “scp” from a command line on Mac/Linux) in much the same way, which will let you transfer files from your computer through drag-and-drop.

If you download the command-line code for linux/mac, you navigate (“cd”) to the folder with the coursework code and run the “make” command. Provided you have the correct libraries installed (they are installed on linux.bath.ac.uk), this should compile the code and you can run the executable ./run file.

To compile the file main.cpp (replace file name as appropriate) on linux, use the command

```
g++ main.cpp -o ./run -Wall -Wextra -g -O0 -I/usr/local/include
-I/usr/include -L/usr/local/lib -lm -lGL -lGLU -lglut
```

To compile the file main.cpp on mac, use the command

```
clang++ main.cpp -o ./run -Wall -Wextra -g -O0 -I/usr/local/include
-I/usr/include -L/usr/local/lib -lm -framework OpenGL -framework GLUT
```

1. New C++ Project

We start by reassuring ourselves that C++ is similar to C, and we check out the main differences. Below are click-by-click instructions for MS Visual Studio, but running the g++ compiler on the same files you yield the same results.

Once Visual Studio 2017 (VS) is open, click *File* menu create a *New Project*. It will ask for the project type – select *Empty Project Visual C++*. You will need to select a sensible name for the project (perhaps *myProject01*) and a location in your user area where you want to store it. This information goes in boxes near the bottom of the window. (In the end of the lab, remember to save a copy of your files also to somewhere else than the lab PC that you are using!)

After a bit of patience, you should see a big empty window with several sub-windows (panes). Now, go to the *Project* menu and click *Add New Item*, and select *C++ File (.cpp) Visual C++*. (You should delete the existing code if there is any). Paste the contents of the file *hello1.cpp* from Moodle. Note this file is just a simple C program, there is no C++ yet. (Another note, if you cannot run the file based on the instructions below, you may need to add `#include "stdafx.h"` in the beginning of the file).

Now, go to the *Build* menu and select *Build Solution*. If all is well, VS should say “Build succeeded” or something similar, in the lower part of the window. If you have errors then one way to see them is to click on *Error list* (near the bottom; this may open automatically). You will get a list of compiler problems in a pane at the bottom of the VS window. This is updated in real time, you may find it useful.

Once you have compiled (built) the program, go to the *Debug* menu and select either *Start Debugging* or *Start without Debugging*. A window should pop up (the *console*), it will say *Hello world*, since this is what your program asked it to write. The program is waiting for you to press a key (and maybe hit enter). When you do that, the program will end and the console will disappear.

Note that if you don’t end your program by asking for some input then the console may appear and disappear so quickly that you never see it.

A nice feature of VS is that instead of building and running your code from the menus, you can just click the button with a *Play* symbol in the toolbar at the top of the screen. (It may be labelled “Local Windows Debugger” but it basically just rebuilds (compiles) your code and then runs it.)

2. “Hello World” in C++

Now go to moodle and find the file *hello2.cpp*. Delete your current code from VS and replace it with the contents of this new file. Compile and run by using the *Start* button.

You should find it does almost the same as your previous program, even if the code looks different.

Some comments on *hello2.cpp*.

- `using namespace std;`
  is a way of enabling some functionality to do with input/output, we will include this in all our programs.
- `cout << "text" << endl;`
  This will print *text* to the console. The `endl` means that the line should end after the text (like `\n` in C). You should read this as “send the string *text* and the end line character to the console output (cout)”. So, in some sense, `<<` passes the thing on the right to the thing on the left.
- `cin.get()`
  This is a function that reads a character from the console input (cin). In the program, the output of this function is stored in a variable called `readChar`.
- Before the line that includes the command `cin.get()` try adding a line `int x=4;` and then
  `cout << "x is " << x << endl;`
  When you run this, observe that it uses multiple `<<` symbols to pass multiple things to `cout`. This is a bit like `printf` in C although the syntax is different. It is possible to control formatting (eg number of decimal places) but we don’t discuss this for now.
- Note `#include <iostream.h>` does a similar job in C++ as `#include <stdio.h>` in C.

3. Reminder of `Planet` structure in C

Find the file *planetStruct.cpp* on moodle. Delete your old code and replace it with this program. Compile and run it. Remind yourself that

- `Planet` is a type
- `mass` and `radius` are member variables for that type.
- `earth` is an instance of that type
- `earth.mass` refers to the member `mass` of the instance `earth`

4. A class called `Planet`

The big difference between C++ and C is that C++ is *object-oriented*. Our idea is that `earth` will be an *object* within our program. In practice, this means that we create a `class` called `Planet`, which replaces our original `struct`.

Now you can choose: you may either write your own C++ `Planet` class based on the lecture notes or you may try the files that are ready on Moodle (in the second case, see the next paragraph). If you want to write your own class it should have two member variables, `mass` and `radius`, and two member functions that calculate the volume and density. First, delete your old code. Write both the class definitions and the main code in the same file: start with the preprocessor directives, then the class definition and finally the main. Alternatively, you can download the corresponding file from Moodle and follow the instructions below.

Find the file *planetClass.cpp* on moodle. Delete your old code and replace it with this program. Compile and run it. You should see that its results are similar to *planetStruct.cpp* but the code is a bit different. The main points:

- Instead of defining a Planet struct using `typedef struct`, we define a Planet class using `class Planet`. The class definition begins `class Planet {` and ends with `};`. (Note this is one of the few times that a closing brace '}' must be followed by a semicolon!)
- The class definition starts with `public:` See the lecture notes for further details on `public:` and `private:` (users and developers).
- The class definition includes declarations of member variables, as in the old struct.
- The class definition includes definitions of **member functions**, which are called `Volume` and `Density`. Within the `main` function, these functions are called by commands like `double density = earth.Density();`

This last point is the most important. The idea is that every instance of the Planet class can "calculate its own density" using a function call of the form *instanceName.functionName()*. This should remind you of *instanceName.memberName* (as in `earth.mass`) but now the member is a function and not a variable.

In order to calculate its density, the Planet must "know" its mass and its radius, but this is fine because these are member variables. You can see how this works from the function definitions. Note the `Density` function uses a call to the `Volume` function. Also note that in the function definitions, all members of the class can be accessed using their names alone (eg, inside the `Volume` function, we write *radius* and not *instance.radius*).

Two refinements for the `Planet` class

Remember in C, you were encouraged to put function declarations at the top of your programs, and function definitions at the bottom. In C++ there is a similar philosophy, part of which says that we should declare member functions inside the class definition, but define them outside.

Take a look at the file *planetClass2.cpp* on moodle, which does this. You can see that the function definitions are at the bottom.

Note the `::` syntax says which class the function belongs to. So `Planet::Density` refers to the Density function for the Planet class. In principle there might be a completely different object (eg `RubberDuck`) that also includes a member function called Density. If we need to specify which Density function we mean, we use the `::` syntax which is sometimes called a

"scope resolution operator". (Of course if we write `earth.Density()` then there is no ambiguity since `earth` is already declared to be a `Planet`, so we do not use a `::`)

Now take a look at *planetClass3.cpp* on moodle. Two new functions have appeared within the `Planet` class. Both the functions are called `Planet`, which is the same name as the class(!). These are special functions called **constructors** which get called automatically whenever a new `Planet` is declared (or created). These functions have the same name but they are different from each other – one of them is defined so that it does not take any arguments (inputs) but the other takes two inputs (both are of type `double`).

When the Planet `earth` is declared in the main function, the constructor function without any arguments is called. When the Planet `jupiter` is declared, you can see that the name of the new instance is followed by brackets containing two inputs. These are passed to the input of the relevant constructor, and this constructor sets the mass and radius of the planet to be equal to these numbers.

See more details on the **constructors** in the lectures notes. Moreover, note that in the lectures we saw that the class declarations and definitions can also be written in separate files. There will be an example of this in the next section.

5. Enabling graphics

In this course we will deal with programs that draw pictures on the screen. You can imagine this requires a bit of technology. Within Visual Studio, we need to import some libraries which are related to a standard way of doing this called openGL. We import the libraries as follows

- go to: *Project* $\rightarrow$ *Manage nuget packages.* a window will open.
- where it says "search" type "nupengl" (this is the graphics package, do not include the quotes)
- click on nupengl.core in the left panel of the window
- check that the description in the right panel says "this library adds the various libraries to your project..."
- click the Install button in the right panel

Now find the file called *glSimple.cpp* on moodle. Delete your old program and replace it with this one. It is a little bit longer than anything we saw so far but don't panic!

Try compiling and running it. As well as the Console window we saw already, a new window should appear with some text in it. This is the window where we will be able to manipulate graphics.

Here are some notes to help us understand the program

- `namespace drawFuncs` and `namespace colours`
  Don't worry about these for now, we'll come back to them.
- `class Window`
  Here we define a class that will be associated with the new window that we have created. You can see that the window has members for its size (in horizontal and

vertical directions) and its position on the screen, and the window has a title, which is an instance of a `string` class. The `string` class is a standard part of C++.
You can see that the Window class has three member functions, one of which is a constructor (since it has the same name as the class itself). If you are interested in them, these functions are defined near the end of the code.

- `Window *win`
  After the definition of the window class, there is a declaration of a **global variable** called `win`. This variable is a pointer to an instance of the `Window` class. Note that this declaration does not create a `Window`, it just creates a pointer that will store the adress of a `Window`.
  In general it is a good idea to **use the minimal possible number of global variables**. Here we use one such variable, we might be able to avoid this but it turns out to be useful because of the way that openGL works.

- `main` function
  This function is short but it may be a bit confusing.
  The `glutInit` function sets up some openGL machinery (we do this just once, at the beginning).
  We create an array for the window size and a string for the window title.
  then...

- `win = new Window(...)`
  This is a new C++ concept. It creates a new `Window`, executes the constructor using the inputs provided in (...), and stores the address of this new window in the global pointer `win`. This is an example of *dynamical allocation*: we create a new object without declaring a new variable.
  Note: if we create a new object in this way, we should usually be careful to delete it later. Here we don't do this.

- `drawFuncs::introMessage();`
  This line just executes a function which was declared in the `drawFuncs` namespace above and is defined below the `main` function. The namespace just says that that this function is defined as part of a set of functions. The name of the set is `drawFuncs`, and the `::` says that we refer to a function inside this set.

- `glutDisplayFunc(drawFuncs::display);`
  `glutKeyboardFunc(drawFuncs::handleKeypress);`
  These two lines call functions associated with openGL. They tell the program what it should do when the window gets updated, and what it should do when the user presses a key. The inputs to these functions are themselves functions(!), which are defined inside the `drawFuncs` namespace. This will be discussed more later...

- `glutMainLoop();`
  At this point, we pass control of the program over to openGL. The idea is that we have told openGL how to draw the screen and what to do if the user presses a key so now we can walk away and let it do the work.

- Function definitions are given after `main`. It might be worth looking at `drawFuncs::handleKeypress` which is fairly simple, as long as you realise that `switch` is a way of taking action according to the value of a particular variable. In this case `key` is an input to the function that is provided by openGL, according to which key is pressed by the user. Basically this function does nothing unless the user pressed q or e, in which case the program ends.

Using multiple files to make programs more readable

You can see that our code is getting a bit complicated now. One nice thing about object-oriented programming (OOP) is that we can split our code into different files. The usual guideline is that for each new class we create two new files. One file contains the class definition – if the class is called `myClass` then the filename is usually *myClass.h*. (The 'h' is short for 'header'.) The other contains function definitions for the class – if the class is called `myClass` then the filename is usually *myClass.cpp* although people sometimes use *myClass.cc*.

If a class is very simple then we can include all functions in the class definition (this is called an "inline class") and uses just one file *myClass.h*. We can also break the rules and define more than one class in a single file. This is bad style but there may be reasons to do it.

We are going to restructure our program to use multiple files. This is slightly tricky so here are instructions:

- Within the main C++ program, turn the lines from "class Window " to the matching ";" into comments, (do this by highlighting them and doing Edit $\rightarrow$ Advanced $\rightarrow$ Comment Selection.)
- At the right hand side of the screen, there is a window called "solution explorer". Click on the main source file in this window (which is called ProjectName.cpp where ProjectName is your project name).
- Go to Project $\rightarrow$ Add Class. A window will appear. Select C++ class and hit the Add button. In the "class name" box enter "Window" (do not include the quotes). Hit Finish. *Note if you try to do this without first commenting-out the class definition in the main program then Visual Studio will complain.*
- Within the Solution explorer you will see two new items: Window.h and Window.cpp. The idea is that the class declaration goes into Window.h and the function definitions in Window.cpp The stuff to go into these files is provided as on moodle as *Window.h* and *Window.cpp* so copy-paste this material into the relevant places.
- Remove the class definition from the main program. Remove from the main program all function definitions that start with `Window::`
- Add a line
  `#include "Window.h"` after the other #include lines at the top of the main program
- Recompile and run. It should work.

If you followed these instructions and things do not work, note that the contents of the main program file should match the provided file *glMulti.cpp* on moodle. Also *Window.cc* and *Window.h* should match the files on moodle.

The point is that the main program is now easier to read (compared with *glSimple*) because the machinery for the `Window` is now hidden away in another file. Note however there are a lot of `#include` statements scattered around the place now. These are needed to ensure that the material from the different files can be connected up properly – it is nearly always the header files that get included. Also note that header files typically start with `#pragma once`. This is a bit of C++ magic that means that even if you include the same file twice, it only gets noticed once. This means that if you have *too many* `#include` directives then it doesn't usually cause a problem.

If you got to here then well done! You have now seen the main ingredients that we will need to make progress.

6. A few more things to try

The next thing is to try to make small changes to what the program does. Here are a couple of things that you might want to attempt, to test your understanding.

- Can you change the program so that it prints a different string to the graphics window (instead of “a string”)?
- Can you change the program so that it prints some output (eg “hello”) to the console whenever the user presses the `p` key (while inside the graphics window)? (Remember that the `handleKeypress` gets called whenever the user presses a key.)
- The next steps (harder) are (i) to make the graphics that appears in the main window change in response to user input and (ii) to make the graphics update themselves, as animations. These will be dealt with in the next session. . .

Version history:
AS, 1 Feb 2021