# Software Engineering Basics 1

## What happens when you press Build in Visual Studio?

For every .cpp file in the solution, an obj file is produced. You can see these in the Debug and Release folders within the Project directory. This is a two step process.

First the **PreProcessor** runs over the .cpp file. This is the simpler of the two steps. It processes all the # commands line by line, such as

* #include – it will take a copy of the file referenced and paste it into its output buffer line by line, processing it as it goes, including its own includes
* #define – tokens defined earlier in a #define are replaced at this stage.
* #ifdef – screens out pieces of code in any #if type statement that evaluate to false at compile time.

The result is a massive .cpp file with all included and processed code. This will include Windows code, openGL code, and anything touched via the recursive including of header files. You can’t see this file on disc.

This file is then passed to the **Compiler** which does the much more cryptic task of turning C++ into machine code. The result is the .obj file, a binary file that lives on disc. The obj file will contain calls to external functions, ie functions whose body exists in another cpp file. It reference these by a “mangled name” or “decorated name” which is usually semi readable and contains the class name and method name.

Once this has been done for every cpp file, the **Linker** is called, which plugs together all these function references in the various obj files and lib files. (Think of Lib files as a collection of precompiled obj files squashed together into a big block of code. That is what they are pretty much).

The final result is an .exe file.

The .exe file may still have some function calls referring to an outside object. This is what a .dll is. The exe, when launched, will attempt to find the dll and get the functions from it at runtime. (Dynamic Linking)

**In brief**

**FOR EACH CPP FILE**

**{**

**PREPROCESS**

**COMPILE to produce OBJ file**

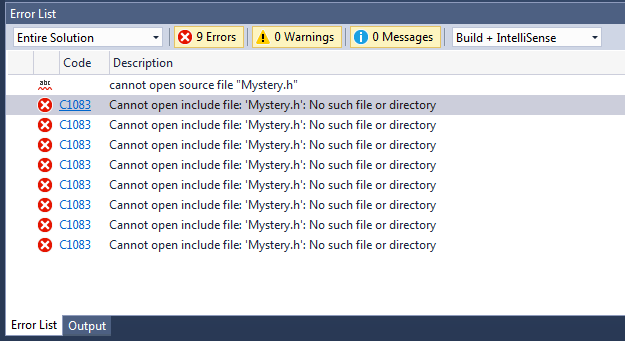
**}**

**LINK ALL OBJ FILES and LIBS into EXE**

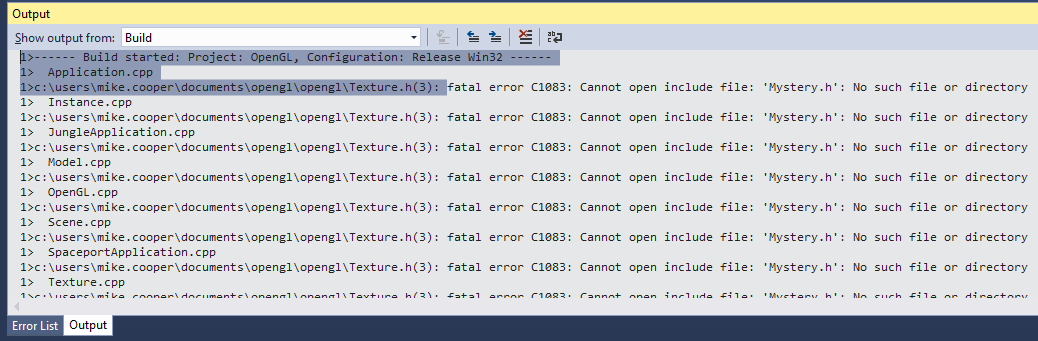
## What can go wrong?

This mental model is very useful for interpreting errors from the compiler (or linker) when they happen.

Visual Studio shows you the Error List pane after attempting to build (I’ve added an include to an imaginary file here in one of my .h files.)



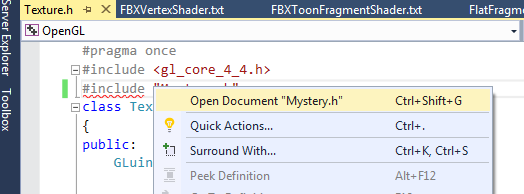
The output pane can provide more info, in that it shows you which file was being compiled when the error occurred. I recommend working with this view when resolving build errors. In either, you can double click on the error to go to its source, if it’s a pre-processor or compiler error.



## Preprocessor errors

“Cannot find include file.”is the most common one.

You can right click on any include file and select **Open Document** to open it. This will tell you if the file isn’t on the Include Path, or take you to the one its opening in the case of ambiguities. (eg two files with the same name exist in different folders on the include path.)



## Circular dependencies

Consider the following case, where two classes need to know about each other.

Scene.h

#pragma once

#include “Instance.h”

class Scene

{

vector<Instance> m\_instances;

}

Instance.h

#pragma once

#include “Scene.h”

class Instance

{

void Draw(Scene\* scene);  
}

What will happen when we try to compile this?

The answer is nothing, because you can’t compile a .h file! You need a .cpp file like so

Scene.cpp

#include <OpenGl.h> // system header

#include “Scene.h”

Let’s pretend we’re the Preprocessor now.

Our output file will start with the expanded contents of OpenGL.h first (and all our other system headers)

We then include Scene.h, which starts with a pragma followed by the include for Instance.h

[Contents of OpenGL.h]

//#pragama once for scene.h has been hit now, note this in our internal memory

#include “Instance.h”

We include Instance.h and it looks like this now…

[Contents of OpenGL.h]

//#pragama once for scene.h has been hit now, note this in our internal memory

//#pragama once for insatnce.h has been hit now, note this in our internal memory

#include “Scene.h”

The pragma once for Scene.h stops us from including this file again, we just step over that line and go straight into the body of Instance.h, and then copy the body of Scene.h in. Our final preprocessed file looks like this.

[Contents of OpenGL.h]

//#pragama once for scene.h has been hit now, note this in our internal memory

//#pragama once for insatnce.h has been hit now, note this in our internal memory

class Instance

{

void Draw(Scene\* scene);  
}

class Scene

{

vector<Instance> m\_instances;

}

At the highlighted line, it will fail to compile because it doesn’t know what a Scene is! This error can be mystifying, because we’ve included Scene.h, but the recursive includes make this happen.

## Forward Declarations

We can fix this by making one of the includes a forward declaration. A forward declaration means we have specified that a type exists without saying how big it is or what its members are, and that we will fill that information in later.

We can only do this when we are dealing with pointers and references. If we have a full member variable or a pass-by-value function argument, the compiler needs to know the size of the type. If we access a function or member variable, the compiler needs details. In the above example, we can forward declare Scene in Instance.h because its just a pointer.

Instance.h

#pragma once

class Scene;

class Instance

{

void Draw(Scene\* scene);  
}

In .cpp files where you need to use a member variable or method of Scene, or declare a Scene on the stack, you will need to include “Scene.h”

If you don’t you get an error about using an *Incomplete type* ie a class you’ve forward-declared without a full declaration.

**Use forward declarations wherever possible. It can make your code compile faster for huge projects, and is a good practice to get into.**

## Compile single files

**If you’re drowning in errors, try compiling one cpp file at a time.** Open the file and press CTRL+F7, or right click on the file in the Solution Explorer.

## Include order

**Always include system files first before including files in your own project.**

Your classes may rely on system includes, eg have a glm::vec3 member, but the third party libraries definitely don’t rely on your code to run!

System files should always have angle bracket includes. This means that the current location of the cpp file is not on the include path, only the include directories specified in the Project Settings.

## Linker errors: Unresolved Externals

Read the description of what the Linker does again. It takes all the obj files, each of which refer to functions outside of themselves, and wires them all together.

Scene.obj will reference Instance::Draw.

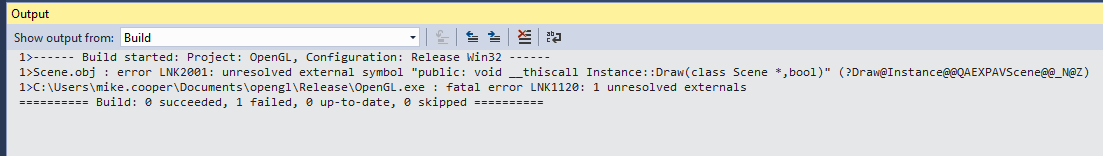
Instance.obj will contain the body code for Instance::Draw

So these two obj files need a link between them as they are rolled into the exe.

If the compiler cannot find the body for an external function, it reports an Unresolved External. You don’t get a line number for this, but it will tell you which obj file it was trying to link, and hence you can work out which cpp file is at error.

You then have to figure out where the function body should have come from.

If I comment out the body for Instance::Draw I get this error.



You can see the mangled/decorated name that the linker uses internally

(Draw@Instance@@QAEXPAVScene@@\_N@Z) and also the function name as used in cpp code.

(public: void \_\_thiscall Instance::Draw(class Scene \*,bool)) – with “\_thiscall” meaning we pass a this pointer, it’s a member function.

When this error occurs it’s up to you to find where the function body should be, and make sure its compiled. (Here we’d look in Instance.cpp as the obvious place)

The most common causes of unresolved externals, in order are:

1. A .cpp file hasn’t been included in the project. Check your project tree
2. A .lib file hasn’t been included in the project. Check the Linker->Input tab in project properties.
3. A function hasn’t been declared. Declare its body, possibly inline in the H file if its an empty default constructor or something simple.
4. Your function body has been #ifdef’d out, most common when you’re using platform specific code.