Jason Witherell's Guide to C++
Updated: 1/8/2020

1. Overview

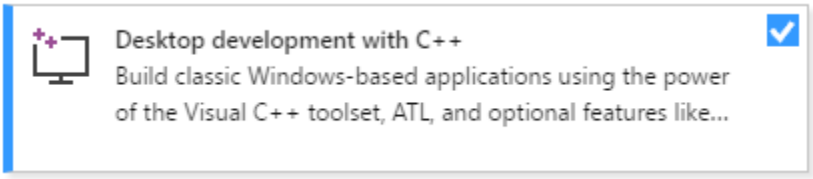
This guide is aimed at my ETEC2101 class, but my ETGG380x students might benefit from it as well. It's not meant to compete with other (much better) references out there. I just wanted a place I could post common C++ operations and then reference it in slides / labs / etc. If you find something missing, ask and I can add a section for it.

1a. Getting Started

I'm going to use Visual C++ 2019 as my "supported" compiler. There are many other compilers out there (CLion, NetBeans, Code::Blocks, etc.) I think Visual Studio has an excellent debugger and broad industry adoption. First off, get a copy! I'd recommend the **"Community" edition** (it's free and does everything we need in my classes)

https://www.visualstudio.com/downloads/

Import note: C++ is no longer a "standard" part of Visual Studio. To enable it, you'll need to enable the "Desktop Development with C++" feature:



1b. Creating a project

A project is a collection of settings and *links* to files that will be compiled into an executable. In linux, a project takes the form of a makefile like this:

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

hellomake: hellomake.o hellofunc.o
	gcc -o hellomake hellomake.o hellofunc.o -I.
```

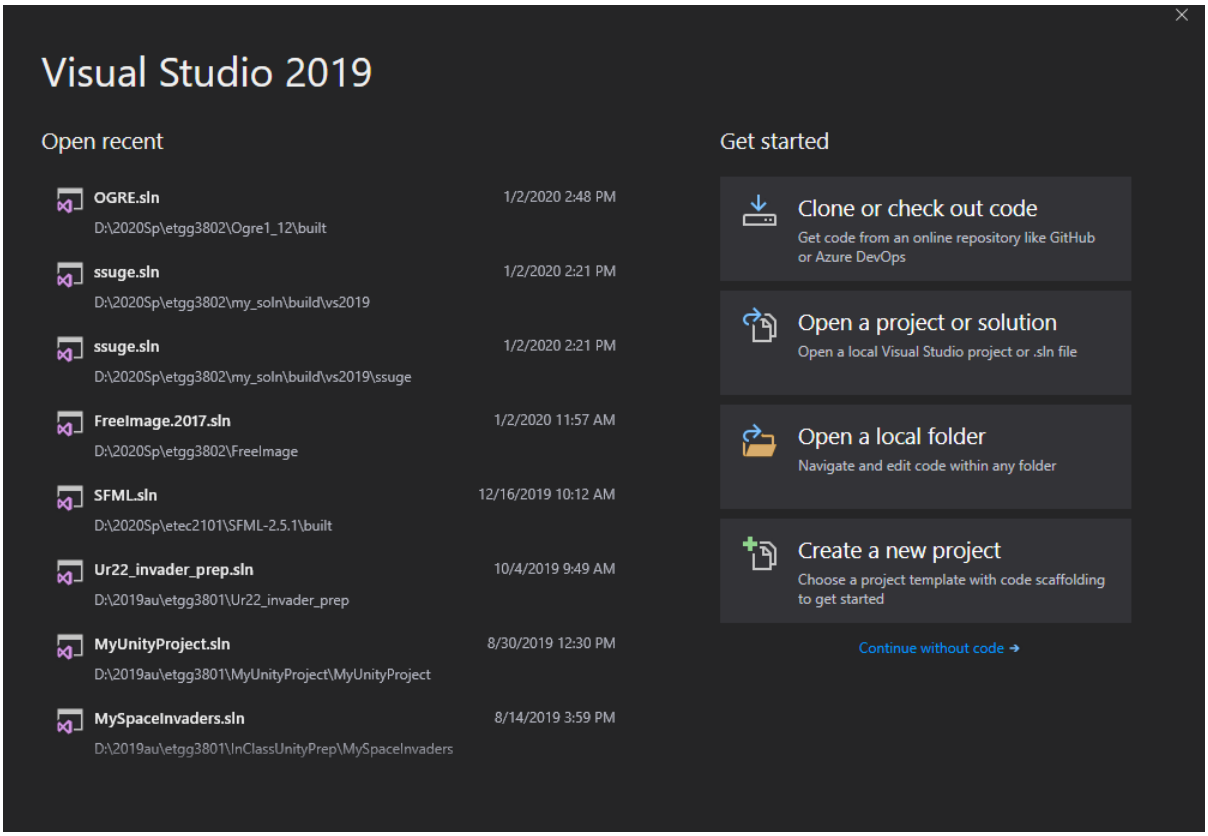
Windows, unfortunately, doesn't have such a tool. Instead, each IDE (Integrated Development Environment) provides its own unique project structure. In visual studio, this consists of several files and folders:

- **my_project.sln**: an xml file with *links* to one or more project files

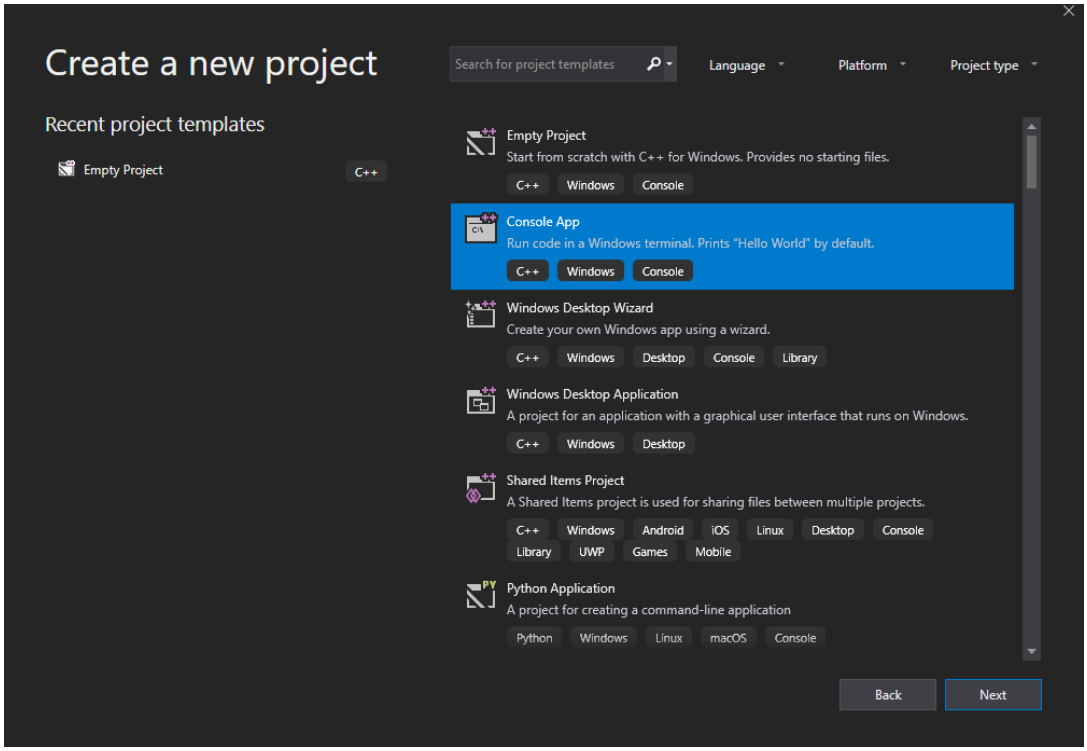
- **my_project.vcxproj**: an xml file with project settings and *links* to project files (.h, .cpp, etc.).
- **my_project.vcxproj.filters**: helps visual studio organize your files visually (not really needed – can usually be deleted [it will be re-created, though])
- … (there will be additional files if you don't select "Empty Project" in the dialog below)

Here's the process needed to make a simple *console* application.

a. Start Visual Studio
b. The splash screen should look like this. Click "Create a New Project"



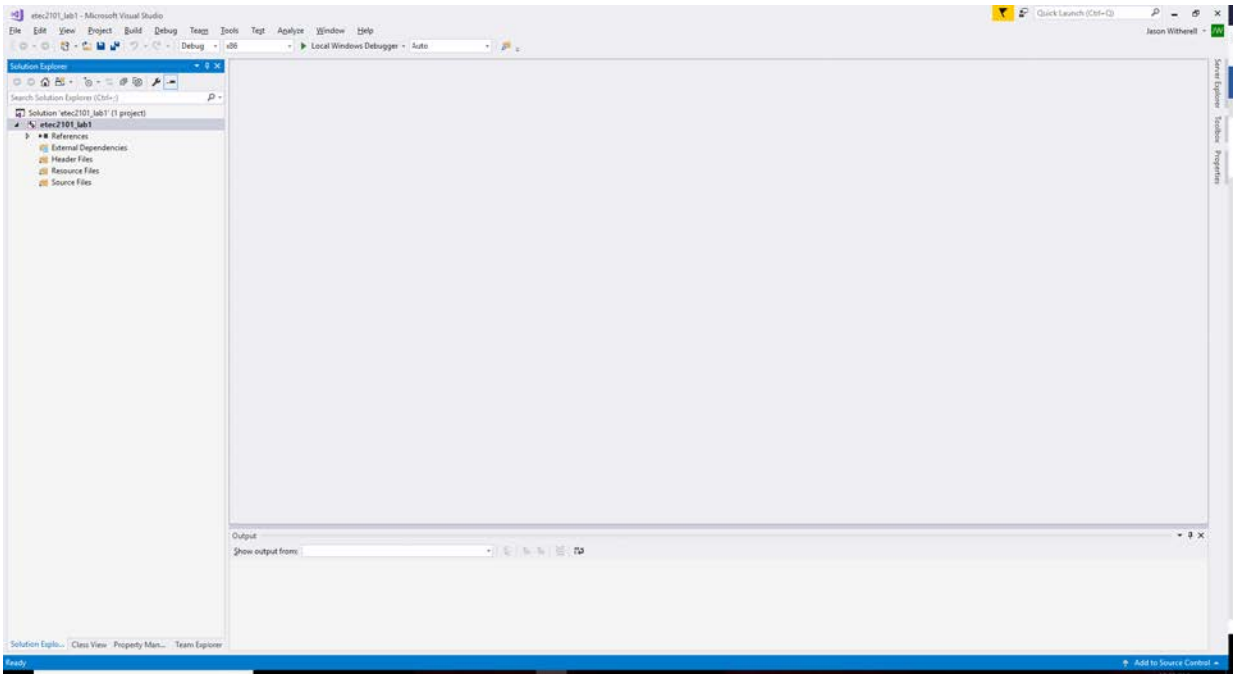
c. For ETEC2101, I usually use a "Console App". For ETGG380x, I usually select "Empty Project". If neither shows up, make sure you installed C++ when setting up Visual Studio.



d. Enter a name for the project (don't leave it at the default value of "ConsoleApplication1"!) and a location. I usually check the "Place solution and Project in the same directory" button. Note, if you select c:\Foo as the Location and "test" as the project name, a new c:\Foo\temp folder will be created.

1c. Visual Studio layout

We'll use 3 panes in Visual Studio. When debugging, you'll use a few more. Here are the main panes:



- Solution Explorer (left): Shows the current project (mine is called "etec2101_lab1") in bold. Right-click on this to modify project settings.
- Output (bottom): When building a project, you'll see warnings and errors displayed here.
- Main panel (middle): Shows the current files (with tabs for other open files). You won't see anything here until you add some files to your project.
- …

1d. Adding things to a project

Remember: a project just contains links to resource files (.h and .cpp for now; later .lib, etc.). To add a new .cpp to the project, Right-click on the project in the Solution Explorer and select "Add New Item...". In the dialog box, select cpp file (I'm calling mine "main.cpp") Here are the contents of my main.cpp:

```
#include <iostream>
using namespace std;

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;

    return 0;
}
```

Adding header files uses the same process. Although, as noted later, header files (.h) don't *need* to be in the project – it's only a convenience. Cpp files on the other hand, have to be part of the project (not just open in the editor) to actually become part of the executable.

1e. Building your project.

Click Build => Build Solution (F7 is the default hotkey). Note, Visual Studio creates four groups of project settings by default: Debug (x86 and x64) and Release (x86 and x64). Whichever is active in the Drop-down is the executable that is being built.

If there is an error, double-click on the error line and you'll be taken to that spot in your project. Tip: *Most* of the time, Visual Studio's "IntelliSense" (code-completion and error-checking) will flag an error with red squiggles as you type. Sometimes, though it gets out of date or lags behind a little. Sometimes it's better to attempt a build and see what errors are produced there.

If there are no errors, you should get a message in the Output window saying "1 succeeded" (one project). If this happens, you can optionally run the program through the IDE by clicking the "play" button (green arrow, middle of the top menu) or Debug => Start Debugging (F5 is the hotkey). Note: this runs the program – unless you have something that keeps it open (like a game loop), the window will shut down immediately upon completion. You can also use Debug => Start without Debugger (Ctrl + F5). This runs the program externally (no Debugger support) and pauses after it completes.

1f. Modifying project settings

For simple projects, you may not have to mess with this, but for large projects (or when using external libraries), you'll be forced to delve into project settings. **Important**: as noted above, your project actually has 4 different collections of settings (Debug | x86, Debug | x64, Release | x86, and Release | x64). It's important here because you can optionally make a setting apply to all configurations or just one – make sure you're always aware of which project is active when setting project settings.

To get to settings, right-click on the project name in the Solution Explorer and choose "Properties". In the top of this dialog, you can select Debug, Release, or All Configurations, and similarly select the Platform (or All Platforms). Most settings we'll need are in the C/C++ or Linker sections.

Note: If you don't plan on using a Platform or Configuration, it's probably best to remove it. To do this, click "Configuration Manager..." At the top of the properties page. Then under Active Solution Configuration, select "Edit...". Then Remove any unneeded Configuration. Do the same thing for the Platform.

1g. (Optional) Re-organize your project directory.

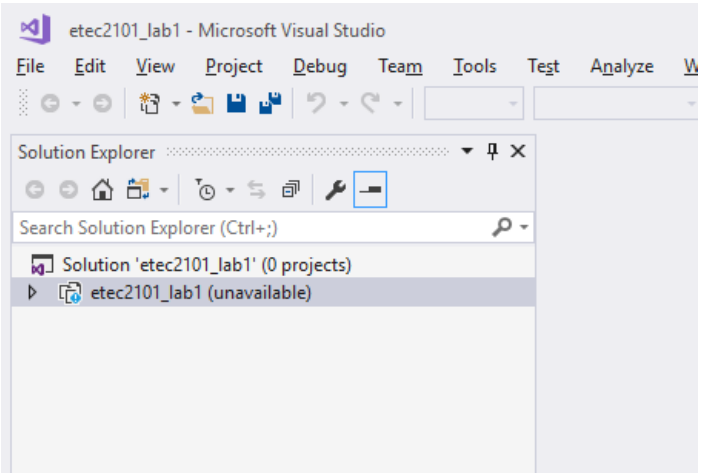
In my opinion, visual studio's default organization (basically dump everything into one folder) isn't good for large projects. My recommendation is to create the project as normal, save the project, then close it. Do all your re-organization, then re-open the project.

Each developer (or studio) has their own preferred style. Here are my recommended folders (each of these is sub-folder within the main project folder) and what to put in them.

- **build**: contains all project files. If using different compilers (but the same source files), you might want sub-folders. Put these files in this folder:
  - my_project.sln
  - my_project.vcxproj

    - my_project.vcxproj.filters
    - my_project.vcprojj.users
    - (there's also a .vs folder that gets auto-created. You can delete it, but it will be re-created each time you load VS)
- **bin**: contains everything needed to run the project (exe, dll's, scripts, etc.) Broken up into sub-folders (Debug, Debug64, …) for each project type
- **tmp**: contains temporary files used during the build process. Broken up into sub-folders (Debug, Debug64, etc.).
- **include**: Includes *our* .h files created in the project. Note: Visual Studio puts all new files in the same folder as the vcprojx file – you'll need to manually change the path when adding a new .h file. If you forget this step, remove (not delete!) the file in Solution Explorer and then right-click on the project and select "Add Existing..." To re-add it.
- **src**: similar to include, but for *our* .cpp files.
- **dependencies**: if using external dependencies (SDL, SFML, Ogre, etc.), I recommend you put all .h and .lib files pertaining to that project in a sub-folder here (e.g. ogre1_12). That way, if your project folder moves to another computer, you don't have to assume the user has things set up as they are on your computer (e.g. c:\sdks\ogre1_12)

After making these changes, re-open the .sln file. Visual Studio might complain that one or more of your project files are unloadable. To fix this, right click on the "broken" project in the Solution Explorer...



...and select "Remove". Next, right-click on the solution and select Add => Existing Project... Find your .vcxproj file (which should also be in build). If you had any files in the project, those links are also likely broken as well. To fix this, right-click on the file in the Solution Explorer and click "Remove" (make sure to select "Remove" rather than "Delete"). Right-click on the Project, and select Add => Existing Item... and re-add your .h and / or .cpp files (which should now be in either the src or include folders)

The last step is to modify the project settings so new temporary and executable files get placed in the appropriate direction. To do this, go to Project Settings (see Section 1f). Make sure to select "All Configurations" and maybe "All Platforms". Note: all paths are relative to the vcxproj files's folder.

- Change Output Directory to **..\bin\$(Configuration)**
- Change Intermediate Director to **..\tmp\$(Configuration)**

Note: The $(Configuration) "variable" will be replaced with either "Debug" or "Release", whichever happens to be getting built. The ".." means go out one folder.

You may optionally want to change the Working directory to match your output director. By default, Visual studio treats the project directory (that contains your vcprojx file) as the working. But often, we'll want it to be near our executable. To do this, go to the Project Properties, Debugging section, and copy/paste the Output directory (to ..\bin\$(Configuration)\).

The next step is to make our include folder a "standard" location. Go to the property window and navigate to C/C++ => General (make sure you're changing this for all configurations). Under "Additional Include Directories", add `..\include\` to whatever else is already there. Now, when writing C++ code, it is preferable to use:

```
#include <my_dependency.h>
```

(rather than "my_dependency.h", which generally only works if that .h is in the same folder as your project file)

To double-check that these settings were entered correctly, build your project. You should have no new files (or directories) being created in your build folder. You *should* have an exe (and maybe a few other files) inside bin\Debug or bin\Release (whichever was built) and a few files in tmp\Debug or tmp\Release.

Since this is kind of tedious, I'll usually get one project set up the way I like it and use that as a "template" for new projects.

2. The build process and including external dependencies

A lot goes on when you hit the "build" button. There are roughly 4 stages a C/C++ application developer needs to be aware of when using an IDE:

2a. Pre-processor

In this stage, any pre-processor macros are "expanded". For example, #include gets replaced by the contents of that header file. #if … gets evaluated – if it evaluates to true, everything in that "block" becomes visible to the next stage (else it's ignored). The programmer never sees this happening, but it is important to be aware of it nonetheless.

2b. Compiler

At this stage, there are no longer any pre-processor macros (and header files are irrelevant). The compiler converts each .c or .cpp file *that is in the project* into a machine-code object file. This file is a collection of assembly-level instructions and is not generally human-readable. These object files are temporary (Visual Studio caches them so the whole project need not be rebuilt if nothing has changed in a file).

2c. Linker

At this stage, a different tool collects all object files into one executable. Any static libraries (also collections of machine code, but from an external source – not from one of our .cpp files) contents are also combined with the executable.

2d. Run-time

When the user double-clicks on your executable, the operating system runs it. The only thing we need to be aware of here is if the project includes a dynamic link library (dll). If so, it is important that that dll be "discoverable" by the executable. The easiest way to do this is to put it in the same folder as our executable.

2e. Creating a project that uses all of the above

This seems to be one of the harder things in C++ for my students. But it doesn't need to be! It's actually pretty simple once you done it a few times.

> **Step One**: get the dependency. If you're lucky, the developer of the dependency will provide a "binary" build for your compiler of choice. If not, you'll have to build the dependency from source (which varies from almost trivial to super-tedious). The dependency I'm showing here is called "SFML". You can find this dependency at http://www.sfml-dev.org. Go to download , pick the version and select Visual Studio 2017 32-bit (or 64-bit – your choice – just make sure to use the "x64" platform if doing this)
>
> **Step Two**: Assuming you did the optional folder re-organization (section 1g), you should have an empty dependency folder. Create a "sfml2.5" (or similar) folder and put everything you downloaded there. You can optionally get rid of the "fluff" (in SFML's case, everything but the bin, include, and lib folders)
>
> **Step Three**: Move the dll's from the bin folder to your executable's folder. In SFML's case, they provide different debug and release dll's (this is a good thing!). Put the dll's that end with "-d" into our bin/Debug and the other dll's into our bin/Release folder. If the dependency doesn't provide different versions for debug / release (like openal32.lib in SFML's case), just put a copy of the same file in both bin/Debug and bin/Release. After this, delete the bin folder from within dependencies.
>
> **Step Four**: Go to the Project Properties page. Reminder: be sure you're aware of what Configuration / Platform is selected! We need to change a setting in the C/C++ properties and two settings in the Linker section:
>
> > **C/C++ property changes**: Go to General. Under "Additional Include Directories", we need to specify where that dependcy stores its header files. Note that all paths are relative to the folder containing the vcxproj file, so for SFML, we want to add a line similar to this: "..\dependencies\SFML-2.5.0\include" (make sure to keep the include directories you already have!) You can tell this step is working if you attempt to include a header from the dependency in one of your .cpp files and don't get an error. Note: some dependencies are designed so that you include the “include” folder, which has nested folders. In SFML’s case, the include folder has a SFML folder within. In this case, you’d include as:

```
#include <SFML/whatever.h>
```

Linker property changes: We need to do a similar step, but Visual Studio asks that you specify *where* the .lib files exist and also *what* .lib files you want to link with. To change the former, go to Linker => General => Additional Library Directories. To change the latter (you'll usually have to do this step individually for debug and release because they usually have different names), go to Linker => Input and type each .lib file you want to link with in the editor. In our case, we'll likely use:

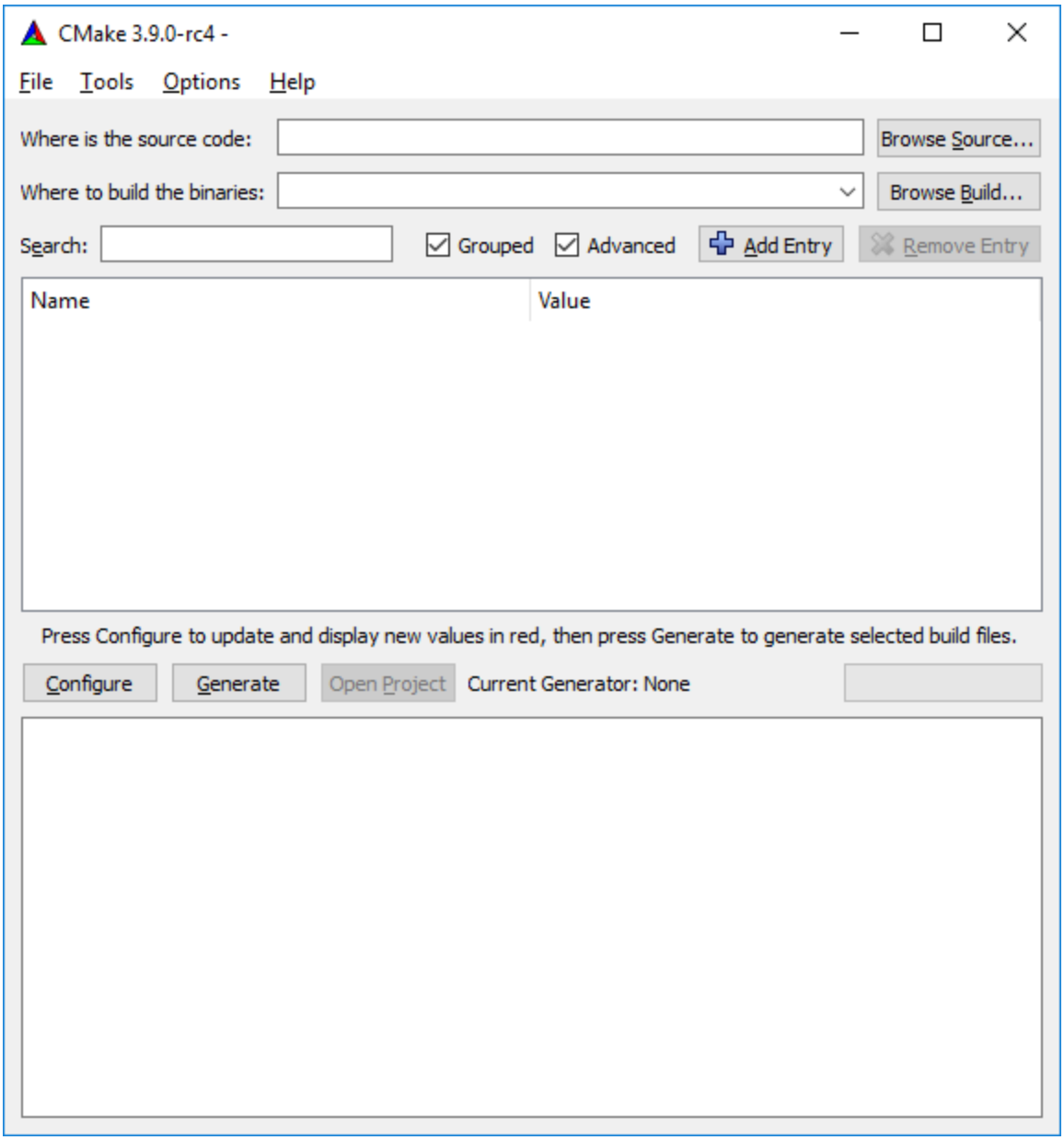
- sfml-window.lib
- sfml-system.lib
- Sfml-graphics.lib
- Sfml-main.lib

2f. Building an external dependency from source

Unfortunately, not all external dependencies come in a nice neat binary package. This problem is made even worse on windows, which has no built-in make utility.

Once common tool among windows developers is CMake. If you download a project from source and see a CMakeLists.txt file in the top-level folder, this project is targeted for cmake.

The first step is to download CMake (a free, open-source project) from www.cmake.org. When you start up cmake (GUI), you'll see this:



Enter the path of the top-level folder of the project (e.g. c:\SDK\Ogre1.11) – the one that has the CMakeLists.txt file in it. I usually make the "build the binaries path" the same as the first box, **but** appending /built to the end.

Important: After you've generated the project (in the next step), if you want to make any changes, just delete the entire built sub-folder.

Click configure. You should be given a list of all compilers available on your system. If using Visual Studio 2017, you want Visual Studio 15 (Microsoft's naming convention is a little weird). Then you'll see a list of options for *how* you want to build the project. You can often change many options. Make sure to read through this carefully. Sometimes the project will describe

the option in detail (online or with a tooltip) – other times, you'll have to experiment to see the effect. Once you've changed all options you wish to change, click Generate.

The generation process should build a .sln (solution) file in your built folder. You want to open this in visual studio, and build the solution (often there is an ALL_BUILD that builds everything project). Wait for a while (hours sometime!) and you should now have a binary for your executable and compiler of choice.

3. The simple stuff (I.e. the things you should've learned already in ETEC2110…)

3a. General Program structure

For console applications, you only have to have one cpp (or c if using a c-compiler) file with a function named main (the entry point). You can change that, and sometimes have to for other types of projects (dll, lib, windows-application, etc.), but generally for console applications, I'd leave it as main. In strict C, the main program must have this "signature"

```
int main(int argc, const char ** argv)
{
      // do stuff.
}
```

Note: you might've seen argv as "const char* argv[]" rather than "const char **" – these are equivalent. In both cases, it means an array of c-strings.

The parameters to this depend how your program was invoked. Normally, if you run the program through your IDE, argc will be 1 and argv will be the name of your executable. If the program is run from the command line *and* the user types things after the executable name (for example "my_program.exe bob 15", argv will have additional strings (in the example case it would contain "my_program.exe" at the first slot, "bob" at the second, and "15" at the 3rd.

In more complicated projects, we typically have multiple cpp files (along with h files). Each cpp file is compiled individually (into machine code files, often with an .obj extension). The linker then compiles the machine code in all obj files (and any linked lib files) into one executable.

3b. Variable declarations / use

C (and C++) is a statically-typed (with a few loopholes) language that uses explicit variable declarations. This basically means, you have to know the type of all variables at compile-time. For example:

```
int x = 15;
float y;
//…
y = 23.15f;
// …
y = 23.16f;
```

Unlike dynamically-typed languages (like Python), you can't change the type of a variable. For example, if we attempted to put this code in the same scope as the above code, we'd get an error (something about the variable already being declared)

```
char * x = "abc";
```

3c. If variants

C and C++ use curly-braces to denote code-blocks. With if statements (and loop statements), if you have only one line of code, it's optional to include the curly braces (but OK if you prefer to add them and be consistent). C/C++ have the usual if structures that other languages have

```
if (x < 15)
{
     // do stuff
}
else if (x > 32)
     // do a single line of code here
else
{

}
```

In a structure like this, we're guaranteed that exactly one of those three choices will be executed. If the else were removed, 0 or 1 block would be executed.

A handy (and, depending on the compiler, possibly more efficient) way to do long if/if-else's is a switch statement. A switch statement is only useful if you're checking for discreet values of a single variable. For example, if we want to print out "apple", "banana", "carrot" if the char variable 'val' holds an 'a', 'b', or 'c' respectively (or print "not recognized" in all other cases), we could use this switch statement:

```
switch (val)
{
     case 'a':
          printf("apple\n");
          break;          // Very important – we'd "fall
                          //   through" without this
     case 'b':
          print("banana\n");
          break;
     case 'c':
          print("carrot\b");
          break;
     default:
          print("not recognized\n");
          break;
}
```

3d. Looping structures

C has 3 main looping structures: while, do-while, and for loops (for loops have a "for-each" version in C++, when working with STL containers)

```
while (x != 42)
{
      // Do something
}

do
{
      // Do something
} while (x != 42);
```

The difference between these is subtle. The first one might not execute at all (if x did hold a 42 when we get to the loop). The second is guaranteed to execute at least once – the condition is checked after we're done with an iteration.

The c-style for loop is made up of 3 parts separated by a semi-colon:

- the (one-time) initialization step
- the condition that is checked at the end of each iteration
- the update code that is run at the end of each iteration

A very common use of for loops is to make a counter variable go from some start value up to but not including an end value. For example, if we want to make i go from 10 to 19, we would use:

```
for (int i = 10; i < 20; i++)
{
      // Do something
}
```

This is completely equivalent to this while loop (although in the for loop case, the variable would go out of scope when the loop is done – in this version, it would still exist)

```
int i = 10;
while (i < 20)
{
      // Do something
      i++;
}
```

As a preview, if we have a C++ vector of integers, we can traverse it in C++ using an alternate form of the for loop:

```
vector<string> my_data;
// Add data to my_data
for (string cur_elem : my_data)
{
      // cur_elem is the value of the current element,
      // not the index.  It is auto-updated.
}
```

This is equivalent (if you dis-regard the scope different) to:

```
for (int i = 0; i < my_data.size(); i++)
{
      string cur_elem = my_data[i];
      // use cur_elem as above.
}
```

3e. Arrays

Arrays are a fundamental data structure in C. They are homogeneous (in that all elements have to have the same type) and are of a fixed size (which needs to be set at *compile* time [dynamic allocation is a work-around if we don't know the size at compile time)

A simple example:

```
int vals[3];
vals[0] = 42;
vals[1] = 50;
vals[2] = 99;
```

You can also initialize the values at the time of declaration (this can *only* appear as part of a declaration). Additionally, we can let the compiler deduce the size (although it is valid to specify the size [4 in this case] inside the square brackets.

```
float fvals[] = {1.1f, 2.2f, 3.3f, 4.4f};
```

There's no way to adjust the size of the array after it's been created (again, dynamic allocation is the work-around). Also, there's no way to determine the size of an array – you have to store this in your own variable.

To the compiler, the names vals and fvals are just memory locations of the *start* of this chunk of data. All elements of an array are in consecutive memory locations.

3f. Structures

C allows you to create compound structures (multiple named variables). This is a very basic form of OOP (Object-oriented programming), without a lot of the features necessary to be called true OOP (inheritance, polymorphism, data + method pairing, member privacy, etc.). Once you've been exposed to classes, think of structs as a class with no methods, and that everything is public. In strict C, you must use a lot of typedef's and some convoluted syntax. If using a modern C++ compiler, you can omit some steps:

```
struct Foo
{
      int x;
      char name[32];
};    // Don't forget that semi-colon!
```

The above code usually goes at the top of a cpp file (or in a .h file if it will be used multiple places)

Now, you can create a variable of type Foo, which has two members:

```
Foo test = {42, “Bob”};          // Initializes the data members,
                                 // in the order they appear in
                                 // the struct.
// In strict c, you’d need: struct Foo test = {42, “Bob”};
Foo test2;
test2.x = 15;
strcpy(test2.name, “Sue”);
```

3g. Pointers

This is one of the most powerful, dangerous, and mis-understood aspects of C/C++. A pointer is simply an integer that holds an address in memory.

Take a simple example:

```
int x = 42;
int * ptr = &x;
```

These two lines of code allocate (on the stack) two integers (the first is obviously an integer; the second, ptr, is a special type of integer). With a debugger, you can see the value that was assigned to ptr, but normally this isn’t important. What is important is that the ampersand operator on the second line got the memory location of x and is storing that value in ptr.

Once you have a pointer, you can do interesting things with it. One of the most basic is to de-reference, or access the thing pointed to by the pointer. For example,

```
*ptr = 52;
```

This actually changes x to hold a 52. Ptr itself is unchanged (it still points to the address of the value stored by x).

This is the dangerous part (don’t do this at home, kids!)

```
ptr++;
```

Now, ptr holds the address of *something* which is 4 bytes after the memory location of x. What’s there? Who knows! It’s 4 bytes because an integer is typically 4 bytes, and ptr was declared as pointing to the address of an integer. Bad things could happen if you access it – and sometimes nothing will happen.

There’s a special relationship between pointers and arrays (and is a big reason that the dangerous bit of code from above is a thing). Say we’ve statically declared an array:

```
float data[5] = {2.3f, 1.9f, 0.1f, 2.1f, 9.9f};
```

The name “data” is really just a pointer to the beginning of the chunk of 20 bytes (floats are typically 4 bytes each). In fact, we can do this:

```
float * fptr = data;
```

After this, we can do some interesting things:

```
*fptr = 8.7f;          // as if we had done data[0] = 8.7f;
fptr++;                // fptr now points to the second element
fptr[0] = 6.4f;        // equivalent to data[1] = 6.4f;
*(fptr + 1) = 7.3f;    // equivalent to data[2] = 7.3f;
```

In C, the way we represent strings is with character arrays. Often, though, we can do some shortcuts using pointers:

```
char my_str[16];
strcpy(my_str, "Bob");      // Puts B, o, b in the first 3
                            // characters, and a 0 in the 4th
                            //   (the other 12 are probably
                            //   just garbage)

const char * another_str = "Sue";     // Similar to above, but
                            // auto-allocates storage
                            // on the stack (the const
                            // is so it can't be changed)
```

We can also use pointers when working with structs (or classes). Here's a simple example (suppose Foo is a struct with an int field called x and a char array called name. We can do this:

```
Foo my_struct = {15, "Bob"};
Foo* sptr = &my_struct;

my_struct.x = 16;
(*sptr).x = 16;             // Same effect as line before
sptr->x = 16;               // "Syntactic sugar" for the line
above.
```

3h. Dynamic memory allocation

As mentioned above, arrays are of a fixed size. If we don't know the size at compile-time, we need to use dynamic allocation. You can either use the C-function (memcpy) or the C++ new operator. It is very important that you free up this memory (otherwise, you have a memory leak) and that you don't mix C-style and C++-style allocation methods. If you allocated memory using malloc, use free to release it. If you used the new operator to allocate, use the delete operator to release it.

```
// Suppose the integer size has been given a value earlier
float * data = (float*)malloc(sizeof(float) * size);
// Use data just as a regular array
data[0] = 15.2f;
// …
// When done with the memory…
free(data);
data = NULL;          // to avoid a dangling pointer
```

Here's the same code, but using C++ style allocation

```
float * data = new float[size];
data[0] = 15.2f;
delete[] data;
data = NULL;
```

You can also allocate a single element using both C and C++ style allocation. This might seem kind of pointless (why not just make a normal instance). The advantage is the pointer can be re-seated (made to point to different things) at run-time. Also, when using classes (and constructors), you often don't know the initial parameters needed to instantiate a variable at run-time.

C-style

```
int * single_int = (int*)malloc(sizeof(int));
*single_int = 42;
Foo * single = (Foo*)malloc(sizeof(Foo));   // doesn't trigger a
constructor call!
single->x = 42;
strcpy(single->name, "Carol");
// Later
free(single_int);          single_int = NULL;
free(single);         single = NULL;
```

C++-style

```
int * single int = new int;
*single_int = 42;
Foo * single = new Foo(42, "Carol");        // Calls the
constructor
delete single_int;    single_int = NULL;    // Note: no [] in the
delete call
delete single;        single = NULL;
```

3i. Functions

x

3j. Pre-processor macros

x

4. Classes
5. References and Pointers (and const)
6. C++-style I/O (Input / Output)
7. Templates
8. xx