Building C++ Programs on Windows, Mac OS X, and linux

This document attempts to explain in brief the process by which C++ source code is turned into executable programs, and (some of) the various tools and *integrated development environments* (IDEs) used in this process.

revision 0.1 · 8/26/2014

Table of Contents

The Build Sequence

C++ source code is typically stored in plain text files with a special extension to signify that the file contains C++ code. While C code traditionally used the ".c" extension, unfortunately C++ has no one convention; thus you will see ".cpp", ".cc", ".cxx", or others. These are generically called *source* files.

Additional files such as *headers* and template description files may use ".h" or other extensions, and also contain code. However, typically the contents of these other files are inserted automatically into the code in "source" files using `#include` preprocessor directives. It is thus primarily source files with which we are concerned, although we cannot completely ignore the existence of these other files.

Building programs from C++ source files is a complex task that requires several steps, many of which are typically hidden from the user. The three primary tasks are known as *preprocessing*, *compilation*, and *linking*, which are described in more detail below. At a high level, compilation is the task of taking individual source files and turning them into *object* files containing binary machine language instructions. Object files, which typically have a ".o" or ".obj" extension, are then joined together by the *linker* to form the final executable program. (In Windows, these are files with a ".exe" extension; under linux and Mac OS X there is typically no extension for executables.)

One other file type that will be important to our discussion is the *library*. The simplest explanation of a library is that it is a collection of object code files into one file. Libraries are much easier to share and use than collections of object files. Libraries typically come in two flavors; static libraries (".a", ".lib") or dynamic (".so", ".dll", or ".dylib"). The distinction here is that dynamic libraries are shared by all programs using them; only one copy of the code in a dynamic library is needed, but that copy needs to be available to all programs which need it. The code in a static library is copied into each program that needs it, increasing the program size and the memory required, but the resulting program can be self-contained.

Depending on the operating system, compiler, and other factors, these different file types may be joined by a host of others which serve various purposes. Visual Studio in particular produces a lot of files beyond the ones mentioned so far, but you shouldn't need to know or understand them for most purposes.

Preprocessing

C++ compilers typically perform preprocessing as an initial step, thus you don't need to do this as a separate step. The C++ preprocessor defines a simple *macro* language which is used to modify the C++ source in various ways. For this document we only really need to examine the preprocessor `#include` directive.

When the preprocessor encounters a statement like

```
#include <foo>
```

or

```
#include "foo.h"
```

it strips out the statement and replaces it with the contents of the file `foo.h`. The first syntax above signifies that you want to include a file which is part of the standard C++ *libraries*. For instance,

```
#include <cmath>
```

provides you with definitions for various math functions you can use on variables of type `double`. These files live in a location that is already known to the compiler, and you don't need to tell the compiler where to find them.

The second syntax implies that you are supplying a header file that is not a part of the standard library. This may be a file you created, or perhaps a file from some third party library that you are using. By default, you can use this notation on any file which is in the same directory (or folder) as the file being compiled. Sometimes, particularly when using third-party libraries, you will want to use header files which live in another location. In this case you may have to inform the compiler where to look for header files. The mechanism for doing this varies; we'll discuss it more when we look at specific compilers and IDEs.

Compiling

At the simplest level, a *compiler* translates source code (text) into machine language instructions (binary) recognized by some specific machine architecture (e.g., your computer). The intricacies of how a compiler works are beyond the scope of this document (we offer a whole course on the topic!) so we will concern ourselves with a slightly higher level of abstraction.

Typically, the compiler uses one source code file to produce one *object code* file. These files contain information about the different *objects* (functions and global variables) defined in the file, and information about external objects referenced in the file. (Note that the use of the term object here is different from its use in the term “object-oriented programming”.) The information concerning a function, for instance, might include the function's name, return type, and parameter types, as well as the machine language translation of the function code.

The syntax for invoking the compiler on one or many source files varies, and will be discussed in more detail later.

Linking

The *linker* is responsible for stitching together the different bits of object code from object files and libraries to form a complete program. The most important responsibility of the linker is to resolve all dependencies and assign fixed memory addresses for the various objects so that they can be found by the objects using them. If you use a function in one module (object file or library) which is actually defined in some other module, you must supply the defining module to the linker or the linker will complain about an unresolved dependency.

Typically you invoke the linker with all of your object files as part of the input. When you need something from the standard C++ libraries, you usually don't need to specify which libraries you need – your linker will automatically include the standard libraries. However, you must tell the linker which non-standard libraries you wish to include in your build; depending on the library, you may also need to tell the linker where to look for libraries.

As with the previous two tasks, the syntax for invoking the linker varies, and will be discussed later.

Tools Overview

The three tasks described above are not necessarily in one-to-one correspondence with the programs that must be invoked to perform them. As already described, the preprocessor is typically never invoked by itself, but is invoked by the compiler on your behalf. Depending on the system you are using, you may be able to use one program to invoke the compiler and linker as well; this program is typically called the "compiler" even though it is doing more than just compilation.

At a minimum, then, you must interact in some fashion with the compiler, and possibly other tools. How you interact with the compiler is somewhat up to you; this section provides a quick overview of some of the options.

Command Line

One option for building your software is to invoke the necessary tools from the command line (i.e., in a terminal on linux or Mac OS X or from a Command Prompt in Windows). For small programs, this can be the fastest and simplest way to go. For instance, on a linux system where your program is entirely specified by the single source file hello.cpp, you can build the program `hello` using the command

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

Scripting

When projects get more complex, especially when you need to include files from multiple directories, use multiple libraries, or invoke the compiler with many options, using the command line can become tedious and error-prone. One way to make the process more efficient and safe is to use a tool such as `make`. This program and its many variants and successors (e.g., Ant, Maven, etc.) provide a syntax for scripting complex builds with relatively few commands.

The advantage of a build script is that you can run one simple command and reliably perform the build the same way every time. The disadvantage is that you must learn the syntax of your scripting system, which may be arcane to say the least. The best way to get started with a tool like `make` is to find an example of a basic `make` script (called a Makefile) and modify it for your own purposes. A good starting Makefile is provided in the section on `make` below.

IDE

Another popular option for building software is to use an integrated development environment, or IDE. An IDE typically incorporates many different tools used in the development process. At a minimum, you can expect an IDE to include a dedicated code editor (likely with special features to make coding easier, such as syntax highlighting and text completion) and a build system, all driven from a graphical user interface (GUI). Some examples of IDEs (which are explored more in depth below) are Visual Studio, Eclipse, Code::Blocks, and Xcode.

As with `make`, the advantage of using an IDE's build system is that you get to run a simple command (e.g., by clicking a button on a GUI) to build your system repeatedly and reliably. As with `make`, the disadvantage is that you have to learn how to use a potentially complex piece of software. For example, the way in which you specify things like the locations of library files is different for every

IDE. However, if you are more comfortable finding your way around a GUI than the command line, an IDE may be your best option.

Obtaining Tools

Visual Studio 2013

Windows

Microsoft Visual Studio 2013 is our preferred environment for building C++ software under Windows. It should be installed on all Windows lab machines administered by CCIT; if you find a lab where it is not installed, please contact your instructor or CCIT directly.

You can obtain Visual Studio 2013 free through the school's DreamSpark account:

http://ccit.mines.edu/CCIT-dreamspark

If you are running Windows 7, you may need to use the "Visual Studio Express 2013 for Windows Desktop" version (the other versions only work on Windows 8 and above). The Express version is a free product, which you can also obtain directly from Microsoft:

http://www.microsoft.com/en-us/download/details.aspx?id=40787

Visual Studio is a full-fledged IDE built around Microsoft's C++ compiler. It is the recommended way to use Microsoft's compiler, as the command line options for the compiler can be rather complex. However, it is possible to compile on the command line using the Microsoft compiler, and Microsoft also has a `make` tool called `nmake`.

Mac OS X and Linux

Visual Studio is only available for Windows. If you wish to use Visual Studio on your personal machine and currently have only a linux or Mac OS X installation, you have a couple of options. Either requires you to have a copy of Windows, which you should be able to obtain a free copy of through the DreamSpark program.

The first option (recommended if you have a reasonably fast computer with 4GB or more of memory) is to install VirtualBox (https://www.virtualbox.org/) and then install Windows on top of it. VirtualBox creates a "virtual" PC running on Mac OS X or linux, so you can run Windows in its own window and still have access to all of your normal software. The main disadvantage of this approach is that Windows will run slightly to a lot slower depending on the speed of your hardware. I recommend you make your virtual hard drive at least 50GB in size if you have sufficient hard drive space.

The second option is to install Windows side-by-side with your other operating system in a "dual-boot" configuration. In this setup, you can decide at boot time whether you want to run Windows or your primary operating system; when in one, you won't be able to use the software installed in the other.

g++/clang

`g++` is the GNU project's C++ compiler, with a long and rich history (see http://www.gnu.org/ for more about GNU). It is free and open source software (FOSS), which is defined differently by different people, so I'm not going to attempt a definition here. The important thing is that `g++` is freely available on just about any platform and operating system you can name. A new competitor, `clang` is the C/C++ front-end for the LLVM compiler (http://clang.llvm.org/). It is largely drop-in compatible with `g++`; for the programs we will be doing in class, the differences between these compilers should not matter,

and you can run either program.

These are the preferred compilers for linux and Mac OS X users. You may use them from the command line, or via `make`, the Code::Blocks IDE, the Eclipse IDE, or (Mac OS X only) Xcode.

Mac OS X

You will first need to download and install Xcode from the Apple App Store (free). It is strongly recommended that you update to the latest version of Mac OS X, too. At this point you should have access to `g++` from the command line. In the latest versions, `g++` is actually an alias for `clang`.

Linux

Depending on your distribution of linux, you may or may not already have `g++` installed. You can check by trying to run `g++` from the command line of a terminal window. If you don't have it, then you should be able to install it from your distribution's software repository.

Windows

If you prefer the linux-style command line, you may choose to use `g++` on Windows. This compiler and associated tools are available in two different packages, Cygwin (https://www.cygwin.com/) and MinGW (http://www.mingw.org/). You must use one of these packages if you wish to use Eclipse or Code::Blocks on Windows (see below). **Note: support from your instructors for using these compilers under Windows is limited at best – you will largely be on your own if you choose this option. Therefore, this option is not recommended.**

A better option if you wish to use `g++` on Windows may be to run linux under VirtualBox (https://www.virtualbox.org/). It is recommended that you have a reasonably fast computer and at least 4GB of memory if you want to try this option. VirtualBox creates a “virtual” PC running on Windows, so you can run linux in its own window and still have access to all of your Windows software. The main disadvantage of this approach is that linux will run slightly to a lot slower depending on the speed of your hardware.

make

The `make` tool should automatically be installed with g++ or clang (see above for installation of these tools).

Eclipse CDT

Eclipse is a free, powerful, full-featured IDE written in Java, originally created for Java programmers. It has been extended for use with many programming languages and tools, including C++ (see http://www.eclipse.org/cdt/). You will need to first install a Java runtime (the latest from http://www.java.com/ is best). You will also need an installation of `g++` or `clang` (see above for how to obtain).

You can download the Eclipse+CDT bundle from http://www.eclipse.org/downloads/ (download the “Eclipse IDE for C/C++ developers”). There isn't really an installer for Eclipse; it comes as a compressed archive (.zip or .tar.gz) containing a single folder named “eclipse”. Extract the folder to wherever you want Eclipse to live on your hard drive. Inside the folder is an executable called `eclipse`. Run `eclipse` from that folder and you are good to go.

For Windows users, see the note above about support for `g++` under Windows – this option is not recommended for Windows users.

Code::Blocks

Code::Blocks (http://www.codeblocks.org/) is a full-featured IDE for C++ with all that that implies. Code::Blocks is freely available from http://www.codeblocks.org/downloads/26 for all operating systems; on linux you should also be able to install it from your software repository. You will first need an installation of `g++` or `clang` (see above for how to obtain). Note that the website states that the Mac OS X version is not entirely stable, so this may not be the best choice for Mac users. Also, see the note above about support for `g++` under Windows – this option is not recommended for Windows users.

Xcode

Xcode is an IDE for Mac OS X only, free from the Apple App Store. When you install it, it also installs the `clang` compiler. More than many IDEs, Xcode is a complex piece of software that may be difficult to learn for beginners.

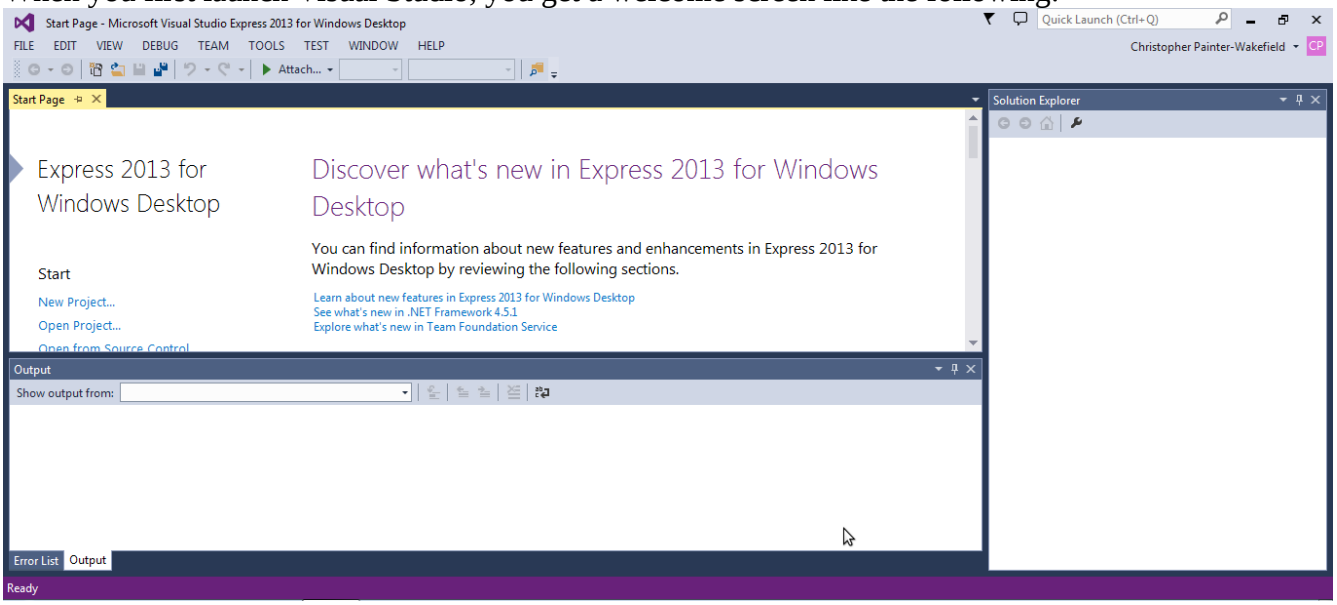
Editors

If you are working with command line tools only or with `make`, you will need some way to edit your code. (If you are using an IDE, then use the code editor provided by the IDE.) The choice of editors is nearly limitless, and as contentious as the choice of programming languages, operating systems, or political parties. In general, any text editor will do the job, but some editors provide more support for programming – anything from color syntax highlighting to code completion and indenting support. The easiest way to get started is to do a web search for “code editor” followed by your operating system name.

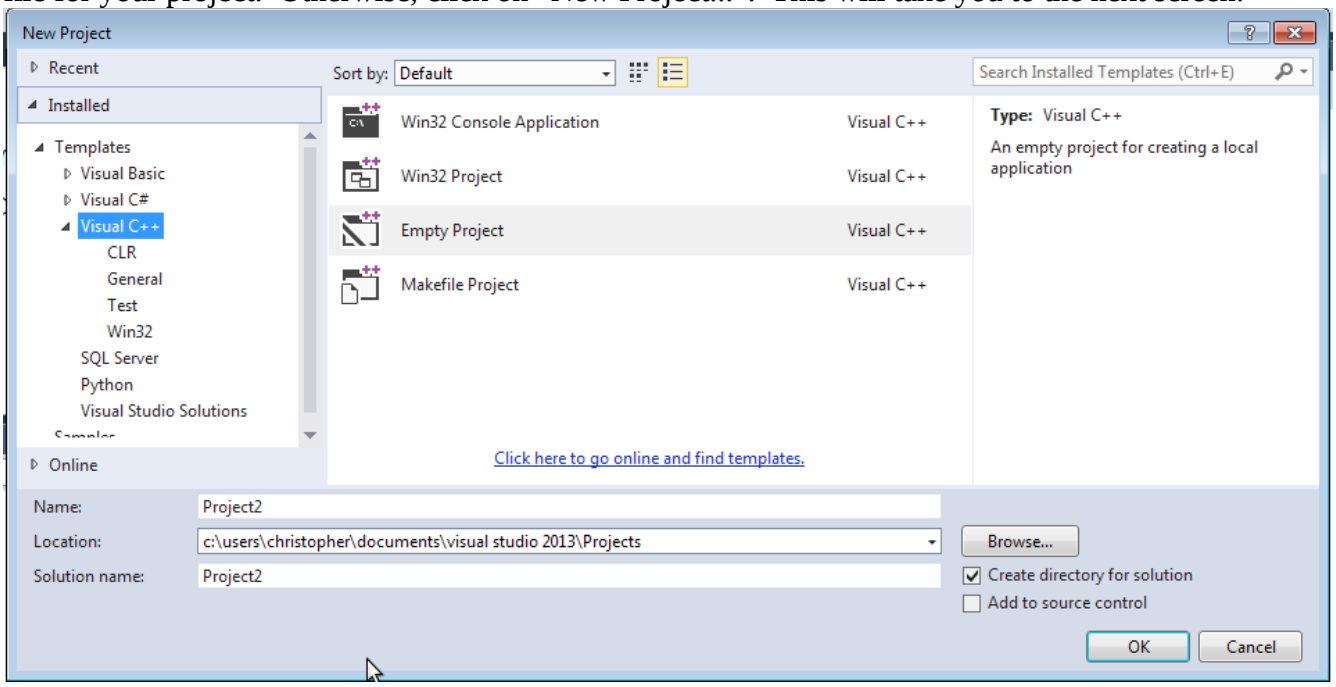
Using Visual Studio 2013

Getting Started

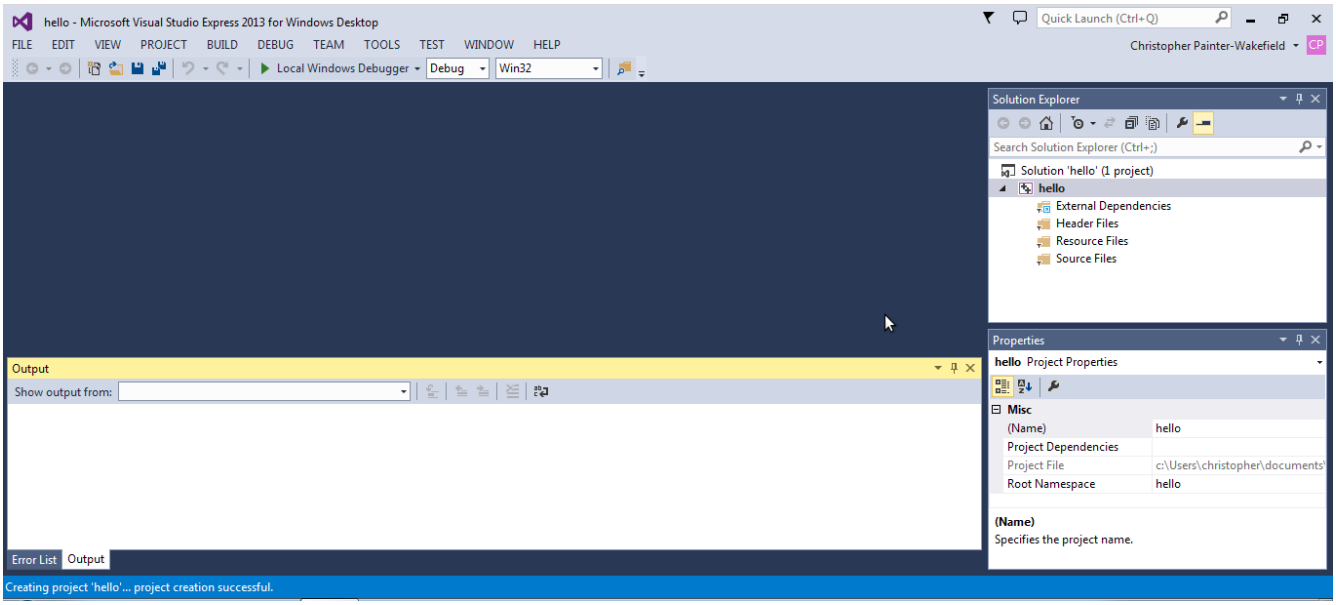
When you first launch Visual Studio, you get a welcome screen like the following:



If you previously worked on a project, you can click on “Open Project...” and browse to the solution file for your project. Otherwise, click on “New Project...”. This will take you to the next screen:

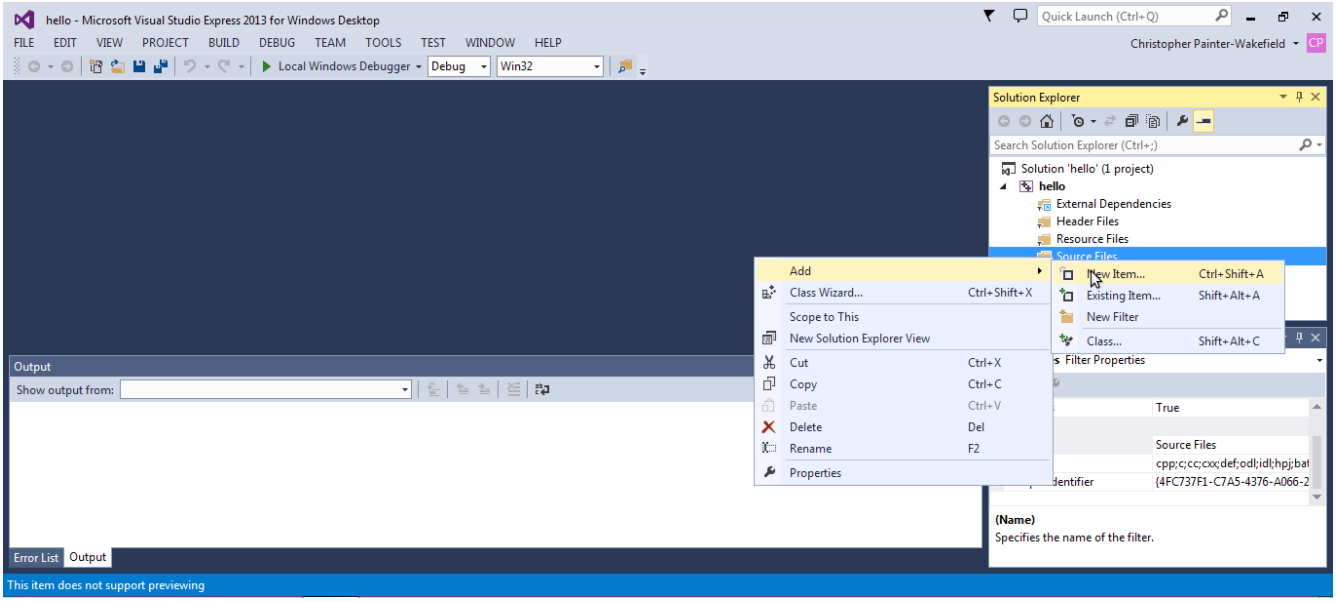


Be sure that “Visual C++” is selected in the left-hand pane. Choose “Empty Project” in the main pane. Give your project a name, then click on “OK” to get to the next screen:

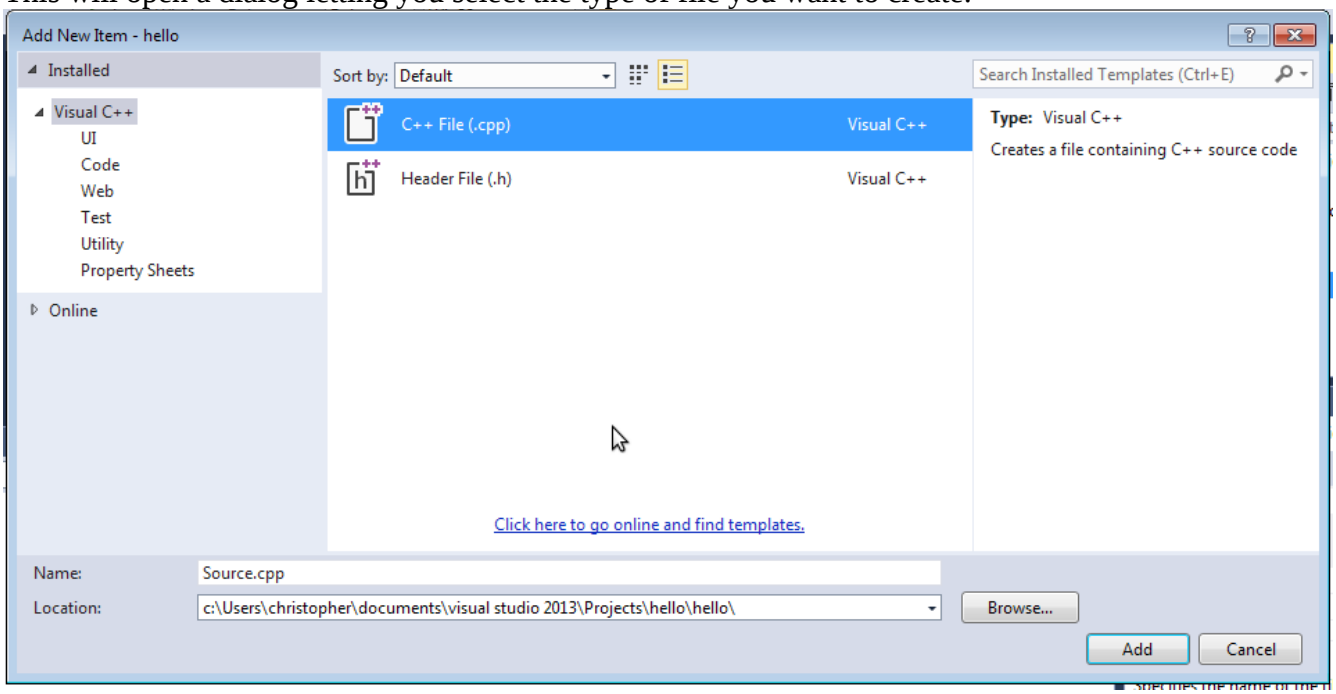


Yours may look different. The project here is named “hello”, as you can see in the “Solution Explorer” pane on the right. The Solution Explorer is where you will manage the files in your project.

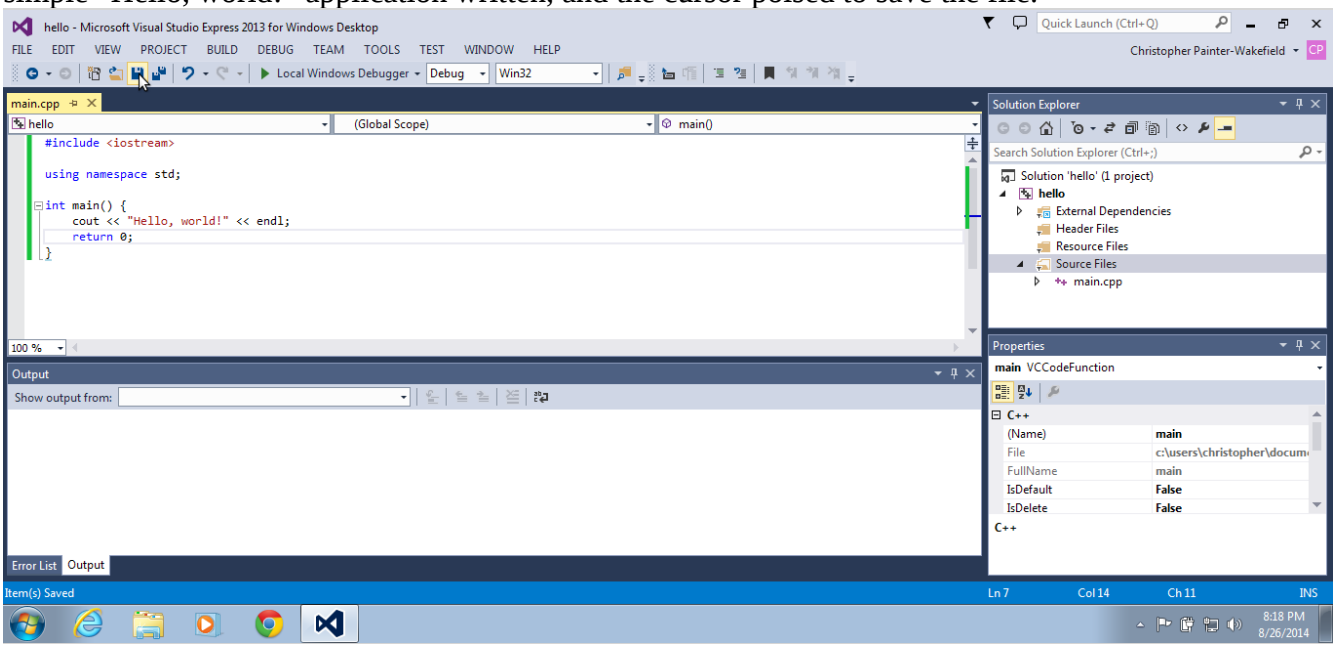
To get started programming, you'll want to create a source file to work with. Using your mouse, right-click on “Source Files” in the Solution Explorer. Choose “Add” in the drop down menu, then “New Item...”.



This will open a dialog letting you select the type of file you want to create:



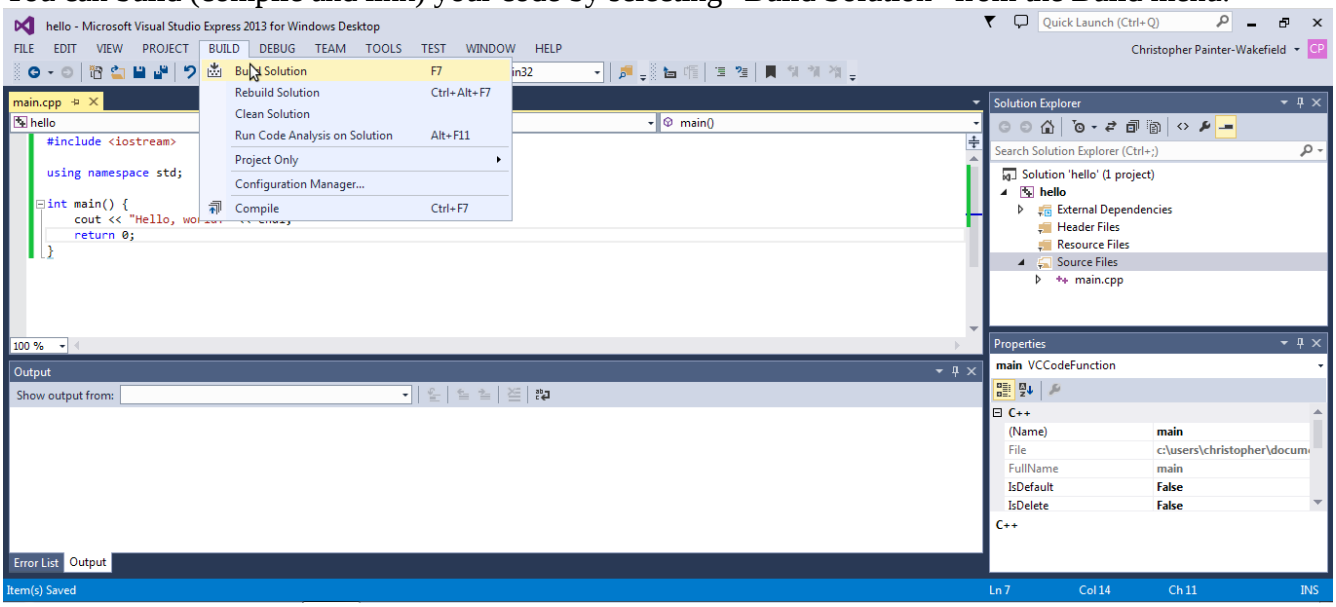
Give your file a name, then click “Add” to get back to the main screen. You should now have your new file open in the main pane, and can start writing code. The image below shows the window with a simple “Hello, world!” application written, and the cursor poised to save the file.



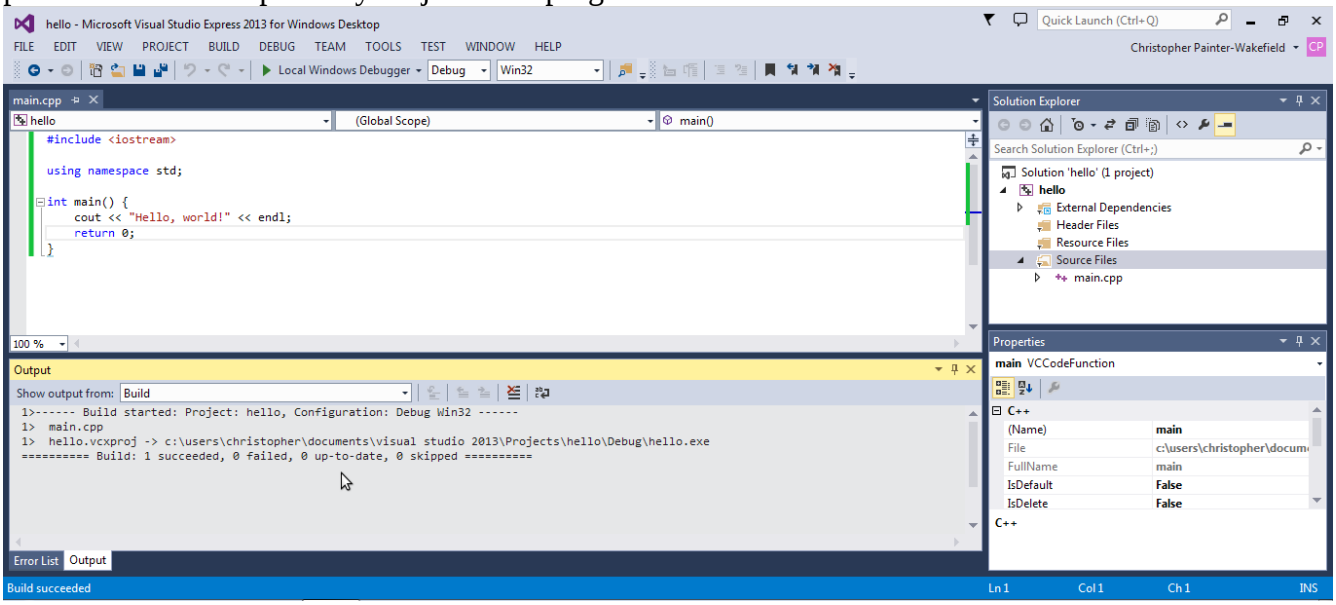
You can add as many source files under “Source Files” as you wish, and your own header files under “Header Files”. Visual Studio will compile everything that is listed as part of your project in the Solution Explorer.

Building and Running Your Code

You can build (compile and link) your code by selecting “Build Solution” from the Build menu:



Depending on whether or not your build is successful, you will see the output pane display information like in the image below, or you might see a list of errors. Note that, among other things, the output pane shows the full path to your just-built program.

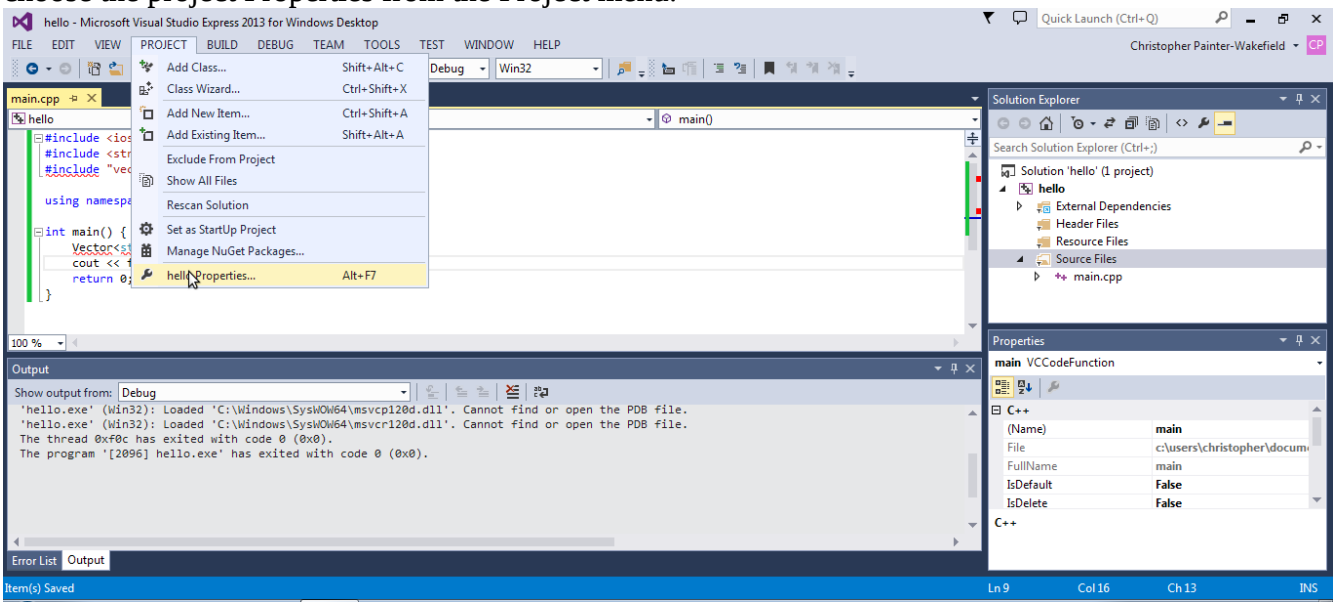


You can run your code by clicking on the green “Play” icon in the menu bar, or by choosing “Start Debugging” or “Start Without Debugging” from the Debug menu. If your program has not been built, or if you have modified your code since your last build, this option should also prompt you to build your code.

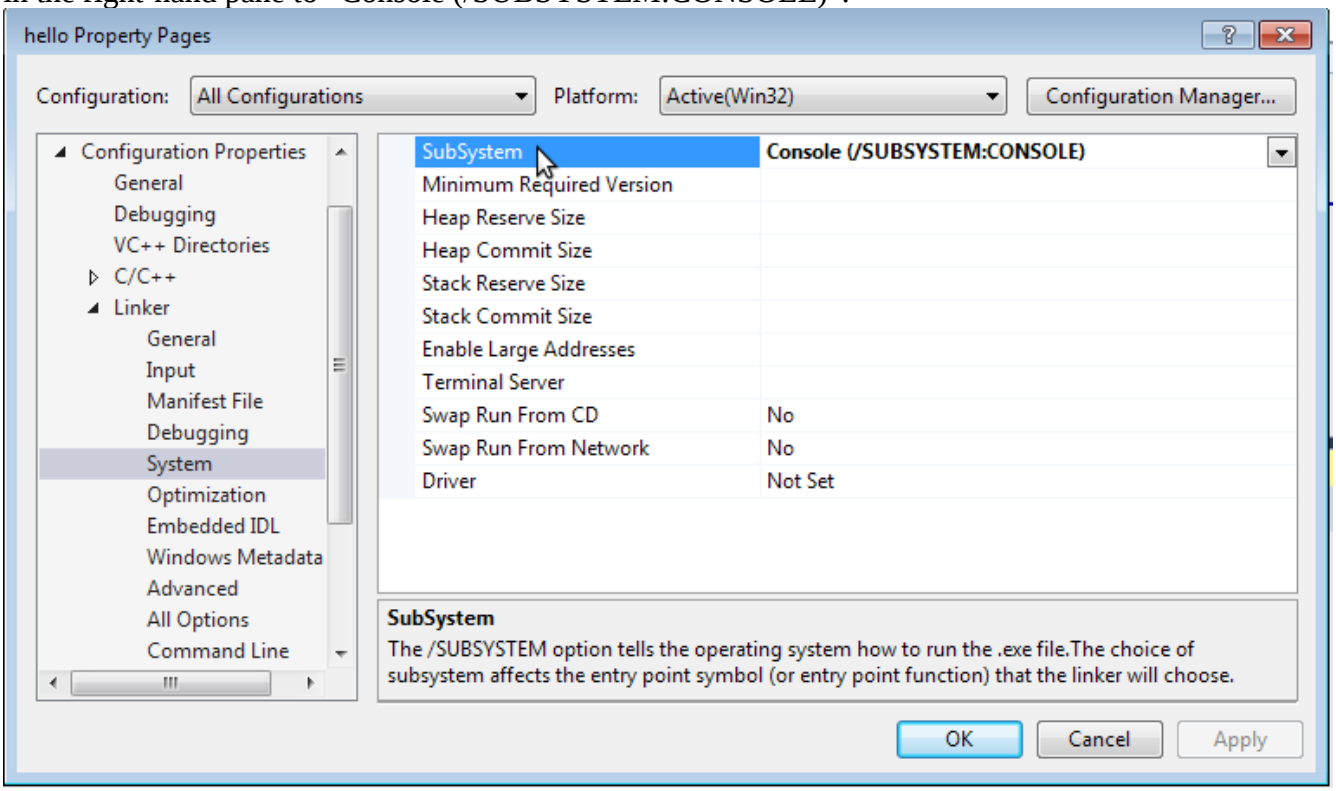
Unless you do some extra setup, you will see your program flash up briefly in a Command Prompt window, then disappear. One way to get around this is simply to launch a Command Prompt (from the Windows Start Menu → Accessories), navigate your way to the folder containing your program, and

run it manually. Since you opened the command prompt yourself, it won't disappear when your program exits, like it does when Visual Studio launches it.

The other approach is more work up front, but may save you time and hassle in the long run. First, choose the project Properties from the Project menu:



You will get a dialog filled with lots of different settings you can apply to your project. First choose "All Configurations" from the Configuration drop down at the top of the dialog. Then, in the left-hand pane, click on "Configuration Properties", "Linker", and "System". Change the "SubSystem" setting in the right-hand pane to "Console (/SUBSYSTEM:CONSOLE)".



Click on “OK” to close the dialog and save your preference for this project. Now to run your program without having it disappear immediately, use the “Start Without Debugging” option (under the Debug menu) to run your program.

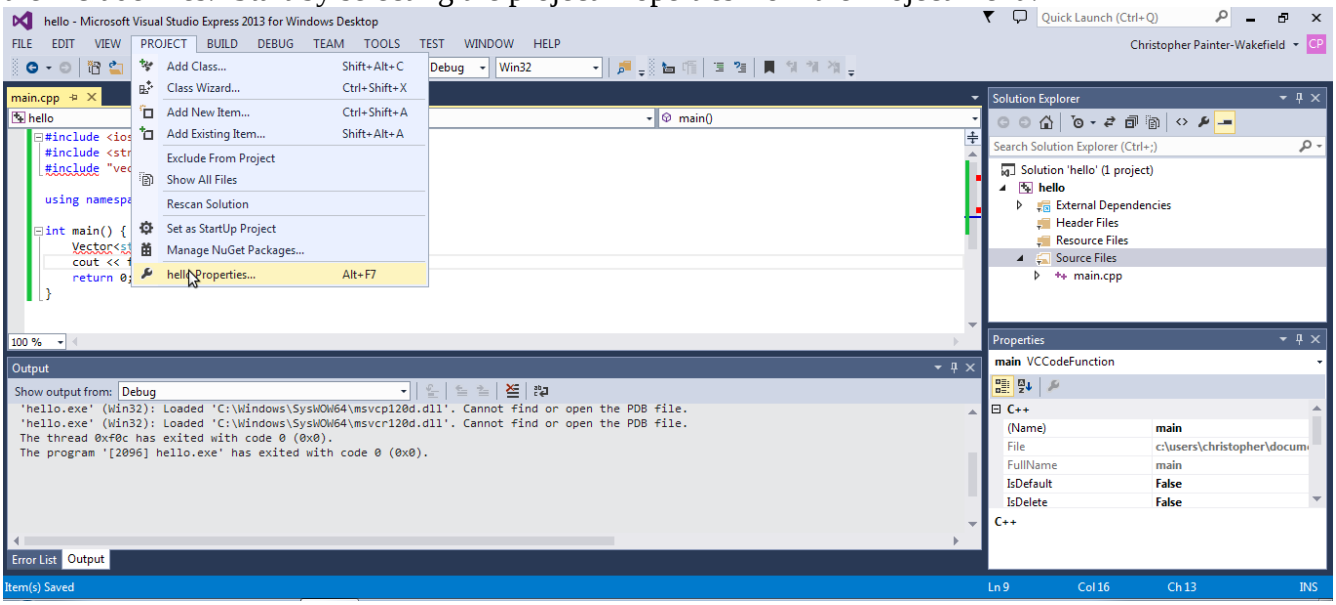
Note that you will need to do these steps for each new project you start, but only once for each project.

Using Third-Party Libraries

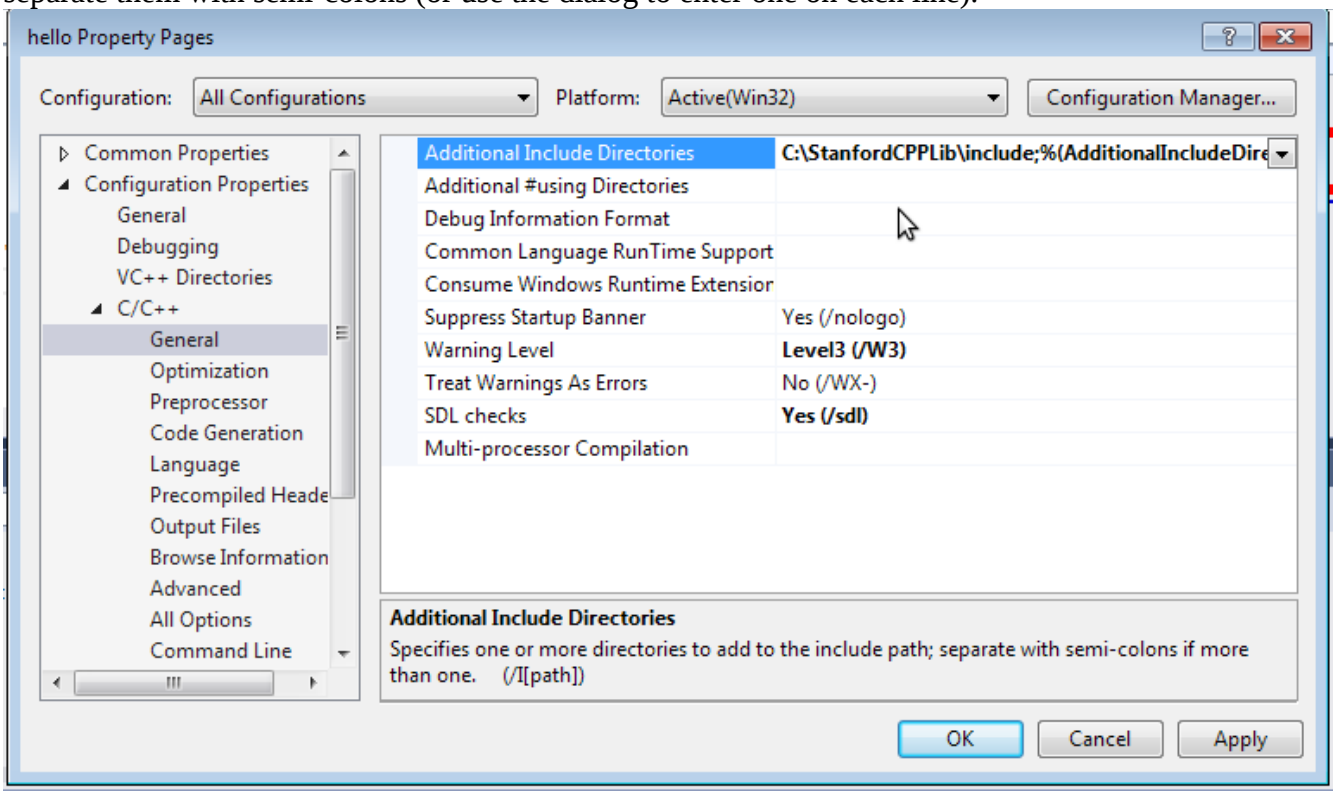
This section provides basic information on how to use third-party libraries with your project, including how to tell Visual Studio where to find your header files and how to link with static or dynamic libraries.

Include Paths

If you need to include files from another directory, you will need to tell Visual Studio where to look for the include files. Start by selecting the project Properties from the Project menu:



In the dialog box that comes up, first choose “All Configurations” from the Configuration drop down at the top of the dialog. Then, in the left-hand pane, click on “Configuration Properties”, “C/C++”, and “General”. The property you want to modify is called “Additional Include Directories”. Clicking next to this option in the right-hand pane will let you either enter a path directly, or open up another dialog box where you can browse and find the directory you want to add. You can enter multiple paths if you separate them with semi-colons (or use the dialog to enter one on each line).



Click on “OK” when done editing.

Linking with Libraries

When you need to use a static library (with a .lib extension), the simplest and easiest way to incorporate it into your project is to add it as an item to Source Files in the Solution Explorer. Right-click on “Source Files”, choose “Add” and then “Existing Item” to get a file dialog which you can use to find and select the library file.

If you need to work with third-party dynamic libraries (.dll extension), you must have the associated development libraries, which also have .lib extensions. These can be pulled into your project the same way as for static libraries, and your project should build just fine. To run your finished project, you will need to copy the dynamic library (.dll) into the folder where your program runs from. You do not need the .lib file to run.

Finally, you can add libraries of either type using the project properties. After bringing up the project properties dialog (Project menu → Properties), in the left-hand pane select “Configuration Properties”, “Linker”, and “Input”; then modify the “Additional Dependencies” property by adding whichever libraries you want (use the .lib filename, not the .dll filename). You will also need to specify a location for finding your libraries (again, the .lib files, not the .dll files); this is done under “Linker”, “General”, with property “Additional Library Directories”.

Using g++ or clang on the Command Line

Basic Usage

Using g++ (or clang) from the command line is a fairly straightforward activity. Everything is controlled by the command line switches or *flags* that you supply. By default, if you issue the command

```
g++ file1.cpp file2.cpp
```

the compiler will try to compile each file1.cpp and file2.cpp and then link the resulting objects together to create a program (replace g++ with clang if using clang). The output program will be called a.out.

To get the output program named something else, use the -o flag:

```
g++ -o myprogram file1.cpp file2.cpp
```

Note that you never explicitly put header files into the command line; these are included implicitly by the #include directives in your source (.cpp) files.

C++ 11 Support

Depending on the version of g++ you have, it may default to using the C++ 11 standard or an older standard. To enable C++ 11 features in your source code, use the -std=c++11 switch:

```
g++ -std=c++11 -o myprogram file1.cpp file2.cpp
```

You should use this flag whenever you are compiling source code; you do not need it if you are just linking (as described below under “Multifile Programs”).

Debugging and Optimization

If you want to be able to use a debugger on your program, you will need to tell the compiler to include debugging information in its output. You do this with the -g flag, which can be combined with any of the other flags. It does not generally hurt anything to include this flag.

Another option that is commonly used is the -O option. This tells the compiler to do extensive *optimization* while compiling your code. Optimization can speed up code in certain cases, and is generally used only when building software that has been thoroughly tested and debugged. In general, you do not want to use optimization with debugging, as the optimization can sometimes effectively rewrite your code in significant ways, making it difficult to debug.

Multifile Programs

When you are compiling programs that have many different source files, you may wish to compile and link in separate steps. The advantage is that you can compile each of your source files independently, thereby seeing only compiler errors for one file at a time.

To compile just one source file into an object file, use the -c flag:

```
g++ -c file1.cpp
```

You do not need to specify an output name in this case, as the output will automatically be named “file1.o”.

Once all of your source files have been compiled into object files, you can use `g++` to link them into a program:

```
g++ -o myprogram file1.o file2.o
```

This approach is particularly helpful when using `make`, as detailed in the section titled "Using make".

Using Third-Party Libraries

Include Paths

If you are using a third-party library, you may need to tell the compiler where to find include files. This is accomplished using one or more -I flags. Each -I is followed (without spaces) by the relative or absolute path of the directory or folder containing include files. E.g.,

```
g++ -I../myincludes -I/usr/local/include -o myprogram file1.cpp
```

Linking with Libraries

If you need to use a static library (typically one with a .a extension), generally the easiest way to make it part of your program is to simply link it in. Supply it as one of the files on the command line and the compiler will automatically recognize it as a library by its extension. You can give a relative or absolute path to the library:

```
g++ -o myprogram file1.cpp file2.cpp ../mylib/mylib.a
```

If you need to link with dynamic libraries (.so or .dylib), you need to use the -l and -L flags. The -l flags tell the linker which libraries to link with, while the -L flags tell the linker where to look for libraries. If you are linking with a library that is generally included with the system but not linked with by default (e.g., OpenGL), then you only need to use -l.

The syntax for using -L is identical to the syntax when using -I for include files. The -l flag, on the other hand, has its own unique twist. All dynamic libraries on linux or Mac OS X will be named starting with the prefix "lib". For instance, on linux, the OpenGL dynamic library is named "libGL.so". When you use the -l flag, you omit the "lib" part of the name as well as the extension. Another wrinkle is that the shared library flags need to come *after* any source or object files which depend on them. So to link with libraries named "libfoo.so" and "libbar.so" living in /home/christopher/mylib, you would do something like

```
g++ -o myprogram -L/home/christopher/mylib file1.cpp -lfoo -lbar
```

As a final note, when you try to *run* a program that is linked against dynamic libraries, the operating system will not know where to find the libraries if they are not in a standard location. So if you are linking against third-party libraries that are not installed into one of the standard locations (e.g., /usr/lib, /usr/local/lib), you will need to tell the operating system where to find your dynamic libraries. On Mac OS X and linux on the command line, you do this by setting an environment variable; for instance, if your libraries are located in /home/christopher/mylib, and your program is called myprogram, you would do the following commands to run myprogram:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/christopher/mylib
./myprogram
```

Using `make`

Using `make` to build your software is one step up from using the command line. Essentially `make` provides a scripting environment geared specifically to managing software builds. In addition to providing various shortcuts for managing multifile builds, it avoids unnecessary compiles by checking the timestamps of produced files against the files on which they depend. However, the user must specify the dependencies properly in the `make` script (the Makefile) for this to work. The syntax used in Makefiles for really large projects can be extensive and arcane, and in fact, such Makefiles are often generated using yet another tool.

For most purposes, a simple Makefile suffices to simplify builds. Essentially a Makefile contains *rules* which specify what is to be built, what *dependencies* must be satisfied before building, and a *recipe* for building. A simple rule might look something like the following:

```
hello: hello.cpp hello.h
	g++ -o hello hello.cpp
```

This rule specifies that the thing being built is the file “hello”, that hello depends on the files “hello.cpp” and “hello.h”, and the recipe (which is identified by starting it with a tab) consists of the single command “g++ -o hello hello.cpp”.

Once you have a Makefile containing a rule, you can invoke `make` on the rule using “make target”, e.g.,

```
make hello
```

The `make` tool will first test to see if the target exists, and if so, if any of the dependencies are newer than the target. If the target exists and is newer than any of its dependencies, then make does nothing; otherwise, it runs whatever commands are in the rule's recipe.

If you call `make` without any arguments, it tries to build the first target in your Makefile. The real power of `make` is this; if there are rules in the Makefile which have the dependencies of a target as targets themselves, then `make` first checks to see if those dependencies need to be made recursively. This means you can have multiple rules that “chain” together to create a build.

For instance, you might have the following rules:

```
hello: hello.o other.o
	g++ -o hello hello.o other.o

hello.o: hello.cpp hello.h
	g++ -c hello.cpp

other.o: other.cpp
	g++ -c other.cpp
```

If you call `make` with this Makefile in the current directory then it will first check to see if hello is newer than the two object files hello.o and other.o. If not, it checks to see if it needs to build the object files first, then it builds hello.

There are a few standard targets which Makefiles traditionally include. The two most common are

“all” and “clean”. The “all” rule is usually the first rule in the Makefile and has only dependencies; its dependencies are any programs or other products you are trying to ultimately build. Following the recursive build approach of `make`, when you try to make “all”, it will first see if the dependencies need to be made. The target “clean” is used to “reset” your directory to a state where all automatically built items (object files, executables, etc.) are deleted. This lets you perform a complete rebuild of everything in your project by issuing `make` twice, the first time with the target “clean”:

```
make clean
make
```

In addition to rules, Makefiles can (and usually do) contain variables that make it easier to customize the build. For instance, a Makefile will typically start with a line like

```
CC=g++
```

and then use the variable reference `$(CC)` instead of `g++` in all of its recipes. This makes it easy to change compilers when moving to a different platform or compiler.

Makefile Example

The Makefile listing below is a good general purpose Makefile that is easy to modify for various projects. To customize for your project, you need to modify the variable definitions at the top of the file. For instance, the “CFLAGS” variable specifies the flags which are given to the compiler when building object files from source files. The “LDFLAGS” will be given to the compiler in the linking step; “LIBS” specifies the -l flags used to include libraries in the linking step. Most importantly, “SOURCES” lists all of the source files in your project, while “TARGET” holds the name of the program you want to build.

This Makefile assumes that you are building a program called “myprogram” from source files “file1.cpp” and “file2.cpp”, linking in two dynamic libraries “libfoo” and “libbar” which live in /home/christopher/mylibs.

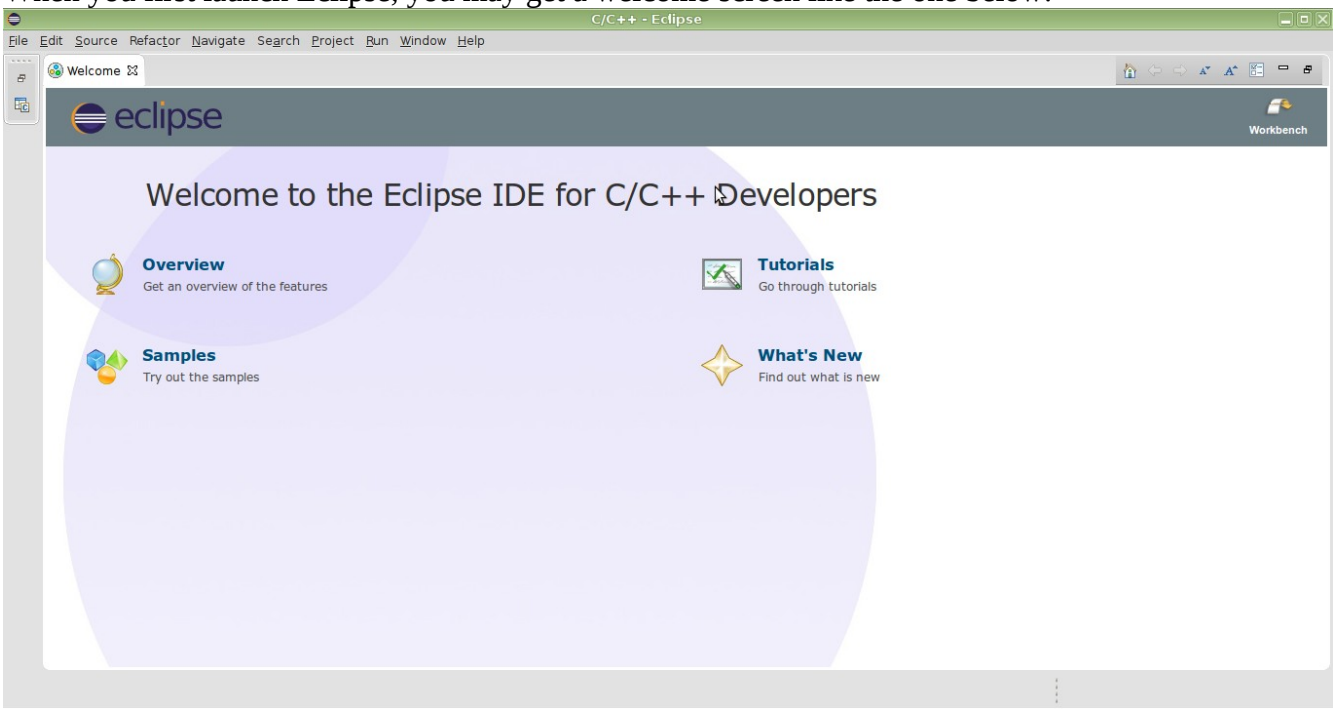
```
CC=g++

CFLAGS=-c -g -std=c++11
LDFLAGS=-L/home/christopher/mylibs

LIBS=-lfoo -lbar
SOURCES=file1.cpp file2.cpp
OBJECTS=$(SOURCES:.cpp=.o)
TARGET=myprogram

all: $(TARGET)

$(TARGET): $(OBJECTS)
     $(CC) -o $(TARGET) $(LDFLAGS) $(OBJECTS) $(LIBS)

.cpp.o:
     $(CC) $(CFLAGS) $<

clean:
     rm $(TARGET) $(OBJECTS)
```

If you want to understand this Makefile better or want to improve it, check out the documentation available at http://www.gnu.org/software/make/manual/make.html. This is documentation for the GNU version of `make`, which is what most linux systems and Mac OS X use. There are also several good tutorials on `make` on the web.

Using Eclipse CDT

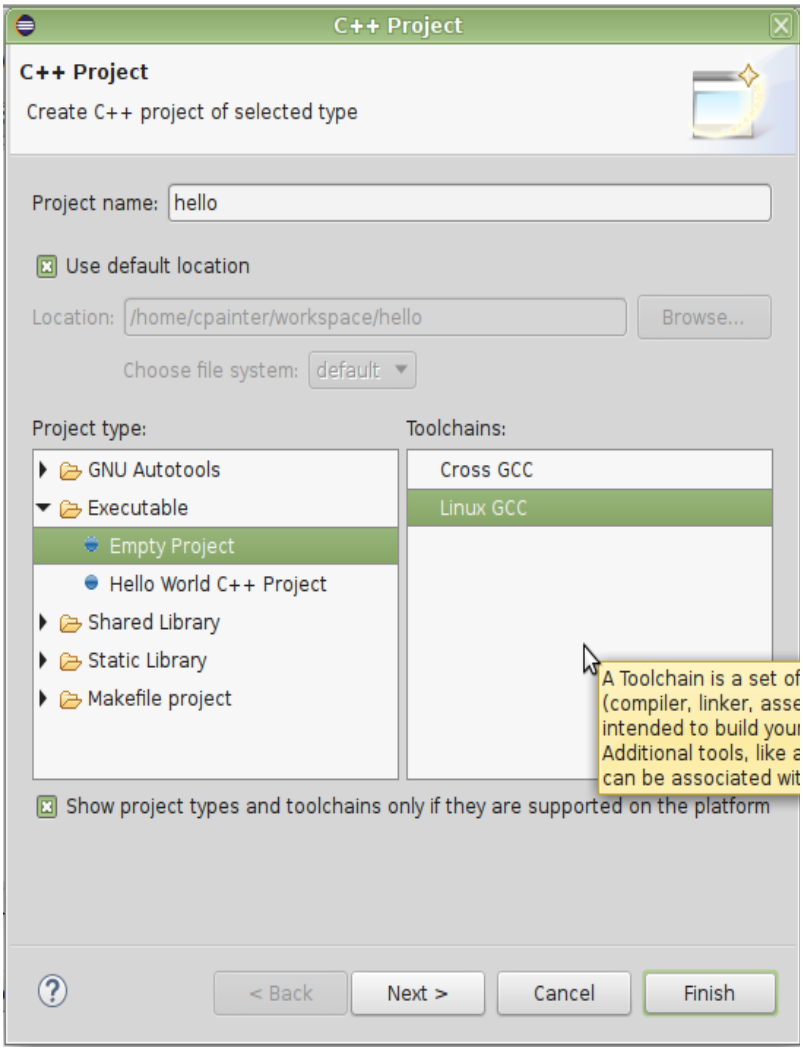
Getting Started

When you first launch Eclipse, you may get a welcome screen like the one below:



Click on “Workbench” in the upper right corner to get started. You may also be prompted to select a location for your workspace, which is a folder where Eclipse will put all of your project files.

When you get to the workbench, start a new project by choose New → C++ Project from the File menu. This will give you a new project dialog like the following:



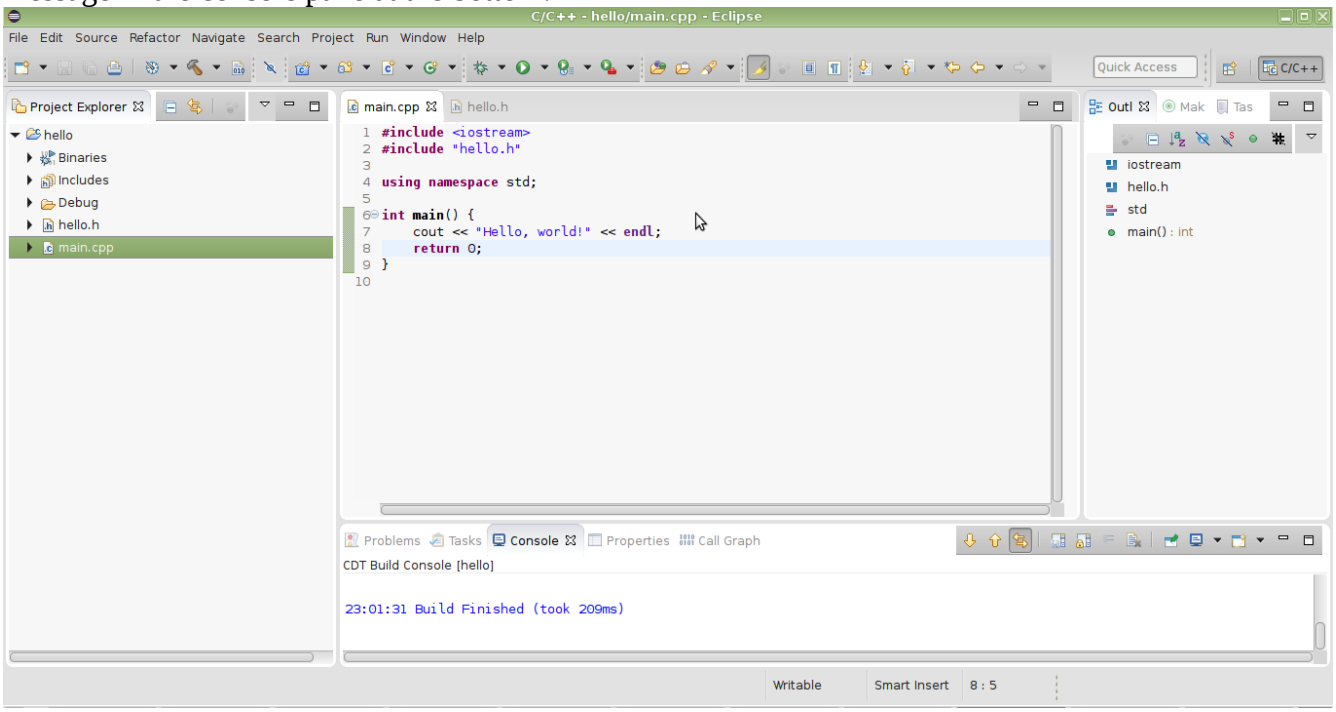
Be sure to select the correct “toolchain” in the right-hand pane, if you have a choice. Avoid “Cross GCC”. Give your project a name, and click on “Finish”.

Once your project has been created, you can start adding files to it, either by using the File menu, or by right-clicking on your project name under the Project Explorer in the left-hand pane.

Building and Running Your Code

If you want to just build your code, make sure you save it first – Eclipse doesn't seem to want to prompt you to save when you just choose to build. (For some reason, it prompts you to save and build if you click on the run icon, though.)

You can build your code by clicking on the hammer icon on the toolbar, or by choose “Build All” from the Project menu. Your code should be built, and you will get compiler output and a “Build finished” message in the console pane at the bottom:



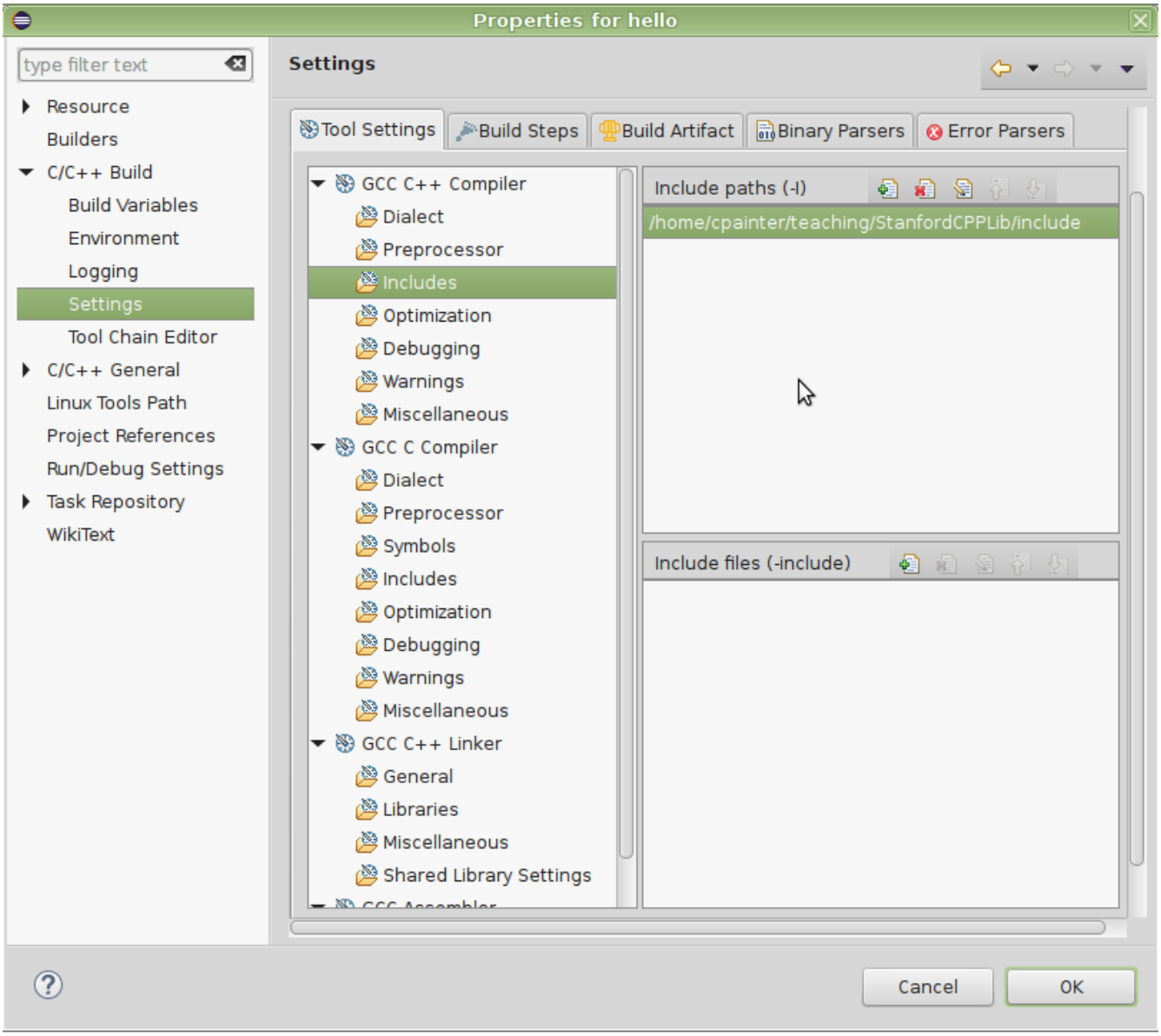
You can run your code using the green play icon in the toolbar, or by choosing “Run” from the Run menu. Any console output from the program will appear in the console pane at the bottom.

Using Third-Party Libraries

Be sure to read the section on “Using g++ or clang on the Command Line”. Eclipse is a somewhat think wrapper around the compiler, so knowing what the command line options are will help you figure out what Eclipse needs from you.

Include Paths

To tell Eclipse to add include paths using the -I compiler flag, go to Project → Properties. In the dialog that opens up (see below), choose “C++ Build”, then “Settings” in the left-hand pane. In the left-hand panel of the main pane, select “GCC C++ Compiler”, then “Includes”. You can add include paths in the right-hand top panel. Each include path goes on a separate line.



Linking with Libraries

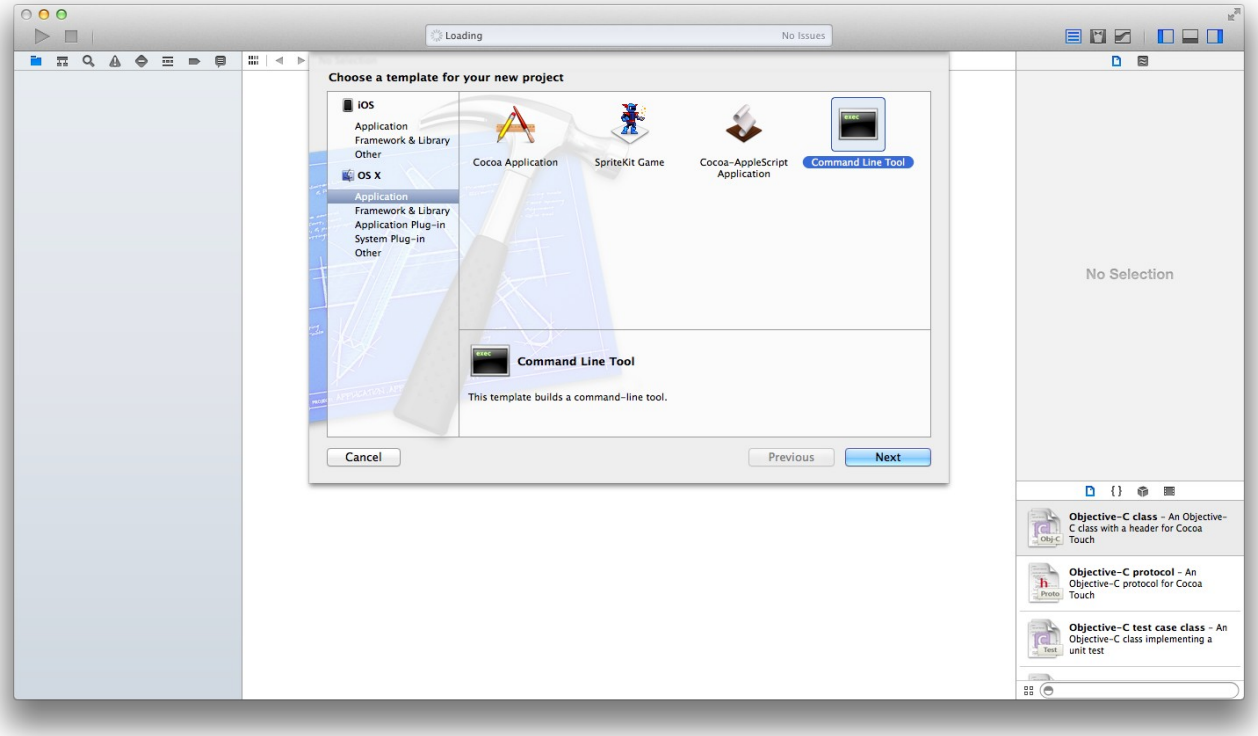
Adding libraries and library paths is essentially identical to adding include paths. Open up the project properties as above, selecting “C++ Build”, then “Settings” in the left-hand pane. In the left-hand panel of the main pane, choose “GCC C++ Linker” and “Libraries”. Now you can add libraries and library paths in the right-hand panel. Note for libraries that you add only the part that would come after the -l on the command line.

Using Xcode

Getting Started

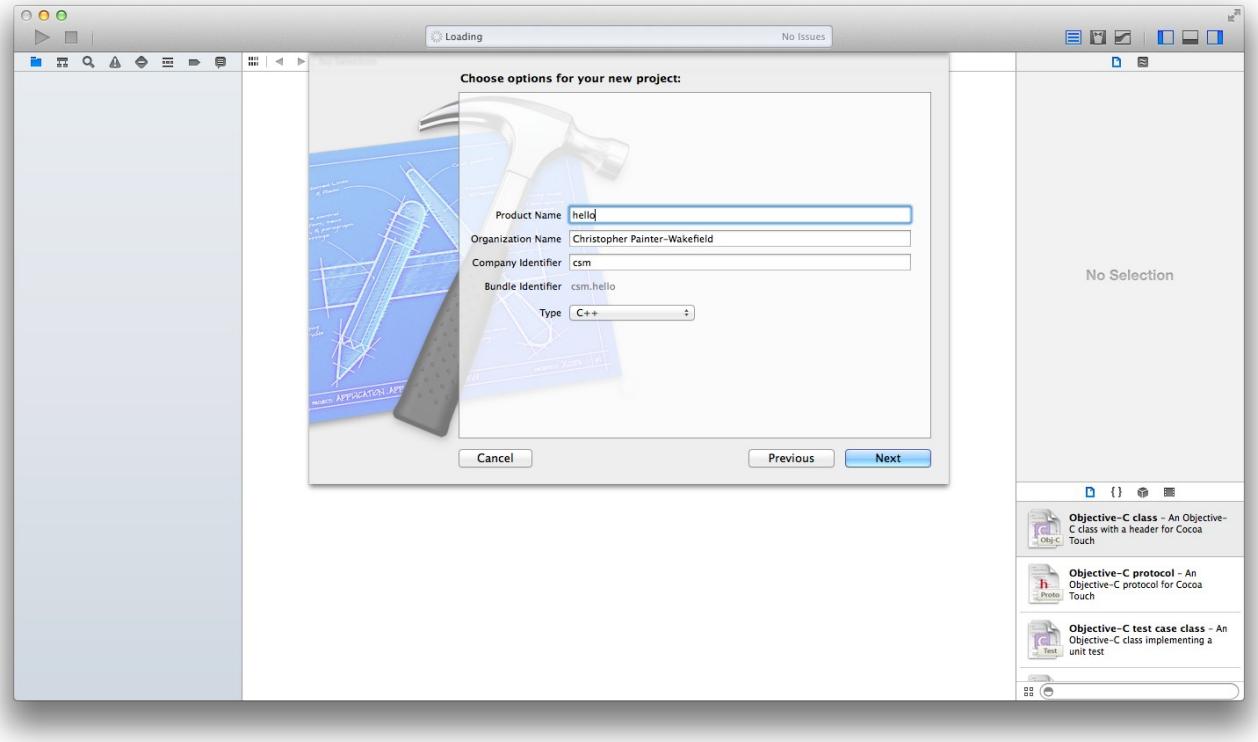
When you first launch Xcode, you get a welcome screen like the following:

If you previously worked on a project, it should be listed in the right-hand pane, and you can open it by double-clicking. Otherwise, click on “Create a new Xcode project”. This will take you to the next screen:

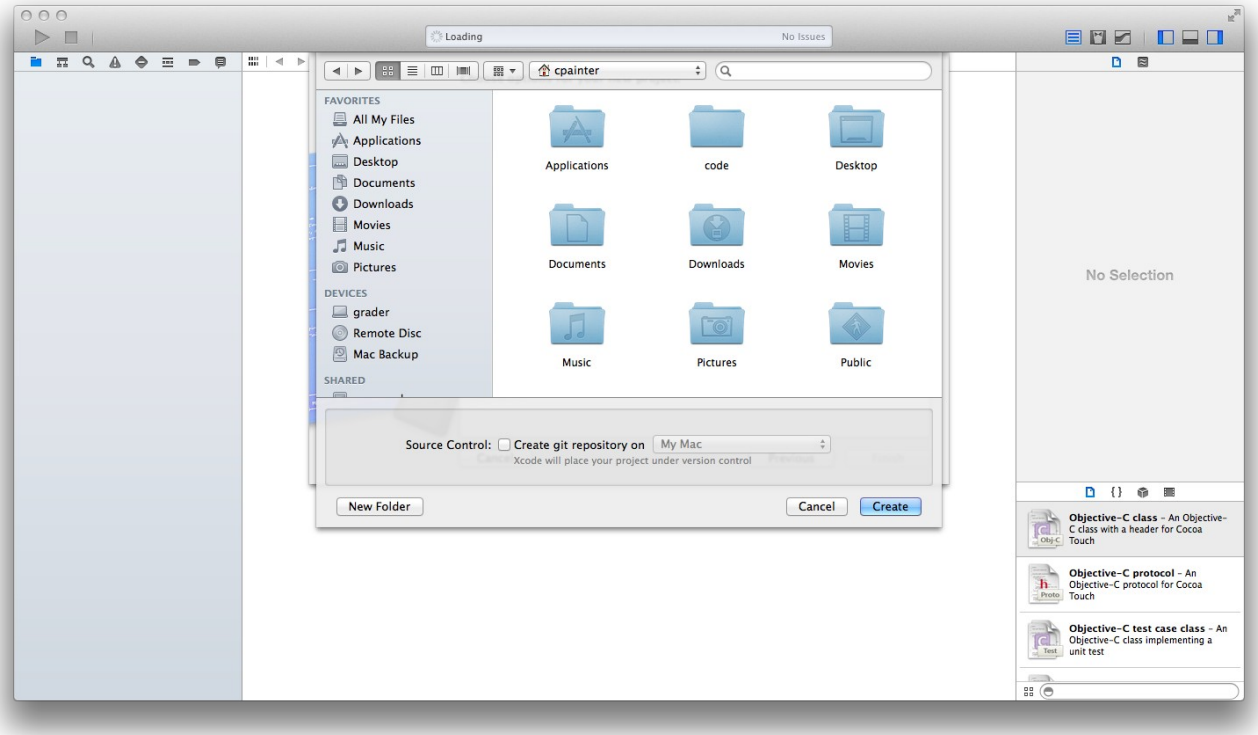


Your screen may look a little different. In the left-hand pane of the dialog box, click on “Application”

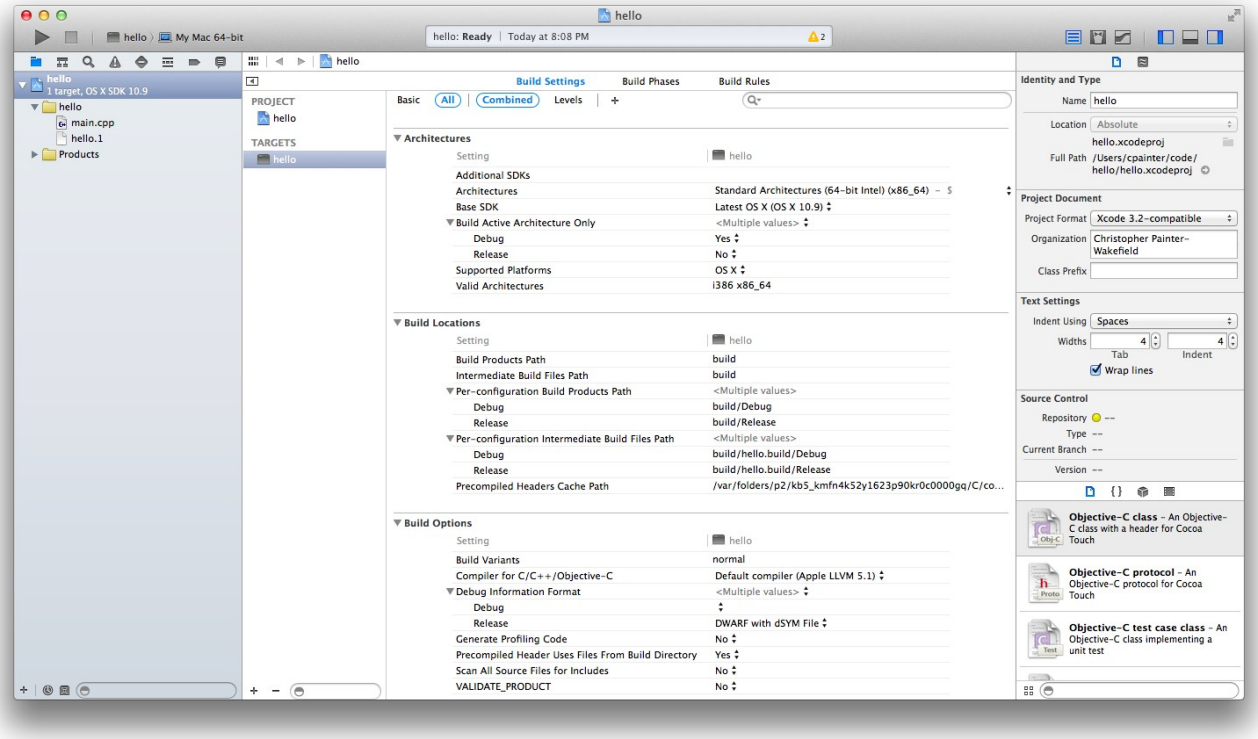
under “OS X”. Then you can select “Command Line Tool” in the right-hand pane. Click on “Next” to get to the next screen:



Give your “product” a name – a good choice here is the name of the program that you want to create. Here I've chosen “hello”. Be sure that “C++” is selected for the Type. Click “Next” to see:



This dialog lets you choose where your project files will live. Select or create an appropriate folder and click “Create”. Your project will be created along with a simple main.cpp file implementing the “Hello, world!” program. You will now be in the main application window (don't worry if yours looks different):



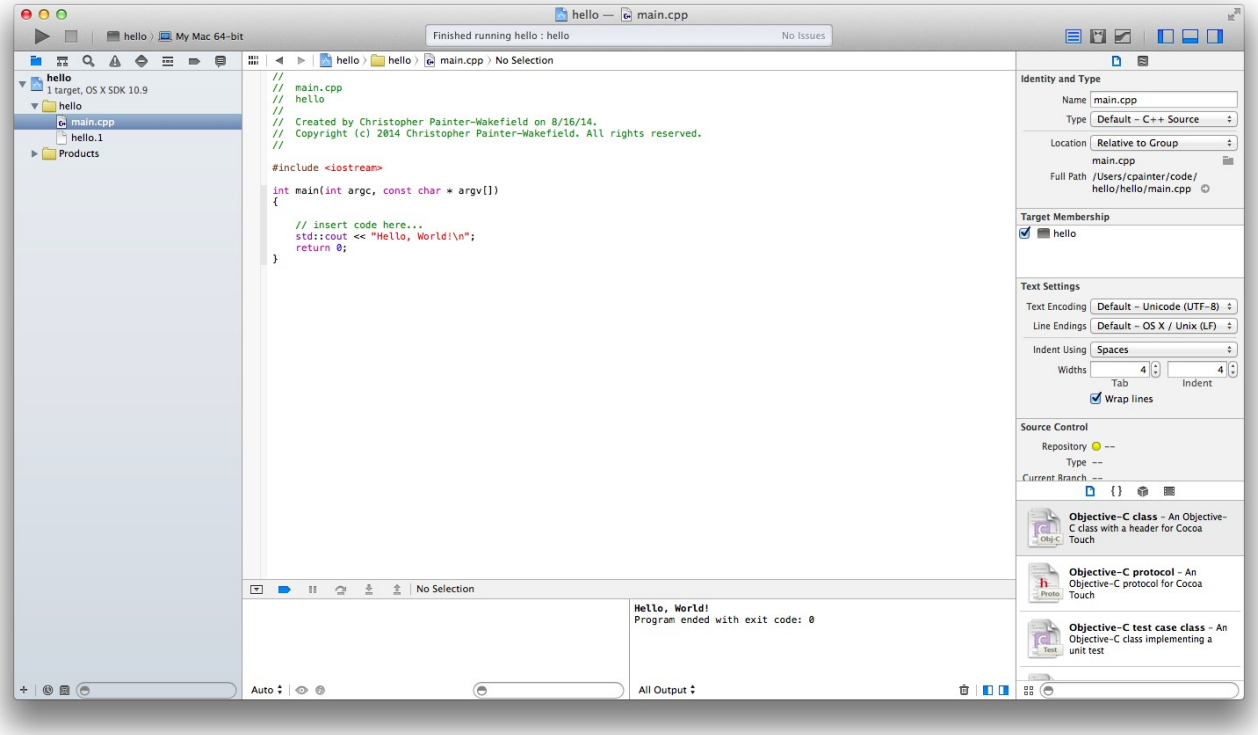
This is a pretty complicated piece of software. It may take you a little time to find your way around. Take some time to explore the interface. You navigate mainly by clicking on things in the left-hand pane.

If you click on the folder icon in the upper left corner of the left-hand pane, you get an overview of the parts of your project. Notice that you have a main.cpp file listed; clicking on that will take you to a code editor view on main.cpp.

You've been provided with a valid C++ program that you can now edit and extend. If you want, you can immediately build and run the provided “Hello, World!” program by clicking the triangular Play icon near the left edge of the title bar.

Building and Running Your Code

You can build and run your code in one step by clicking on the “Play” icon at the left of the main window title bar. If you do this on the provided “Hello, World!” program, for instance, you get a window that looks something like the following:



Note that the program output, along with the return code from the program, are displayed in the lower right of the central pane. If you want to just build or just run an already built program, these options are available from the Product menu in the top menu bar.

You can also run your program from a file browser window by double-clicking, or from the command line. First, you need to know where the executable file has been put. If you go to the left-hand pane and click on the arrow next to “Products” you should see your program (e.g., “hello”) listed. Click on your program name under Products. The upper right pane of the window, where it says “Identity and Type”, will show you the full path to the file. There is also an arrow which will bring up a browser window at that location. (You can change the location that the program is put in; see the section on “Build Path” below.)

If you double click on a simple program like the hello program, you may briefly see a terminal window pop-up, and then disappear. This is your program running; it launches a terminal window for the output of the program, but this window closes as soon as the program completes.

A better way to see your program output is to bring up a terminal window (via the Terminal application), navigate to the directory (folder) containing the program, and run the program from the command line (see “Quick Command Line Tutorial” for help with this).

Setting Up Your Project

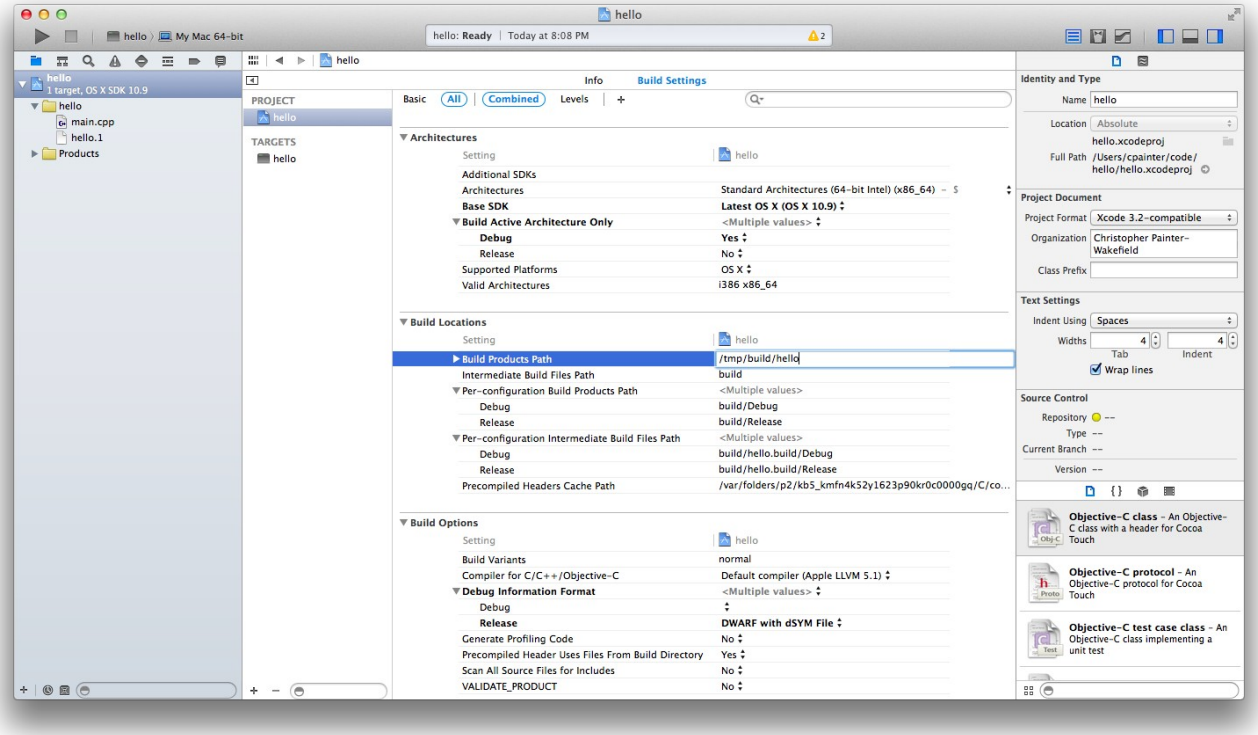
If you are only creating a simple command line application, with no dependency on third-party libraries or include files, then you probably don't need to do anything in this section except possibly set your build path. If you need to use third-party libraries or the like, then this section explains how to tell Xcode where to find the files you need.

The first step for all of these settings is to click on the project name at the top of the left-hand pane. This will bring up the project settings page in the center pane.

Build Path

By default, Xcode buries your compiled executable deep in an obscure location where you probably don't want to go find it (for instance, to run on the command line). To fix this, bring up the project settings page. The center pane will look something like the picture below. In the left portion of the center pane you can select either the project or a target; select the project.

Then select “Build Settings” at the top of the pane. Below “Build Settings”, make sure both “All” and “Combined” are selected. Now you can scroll down through myriad settings for the project until you find a section titled “Build Locations”. Click on the space to the right of “Build Products Path” and wait until it lets you enter a new location. You can now set the location to something sensible, like “/tmp/build/project_name,” making it much easier to find your program.



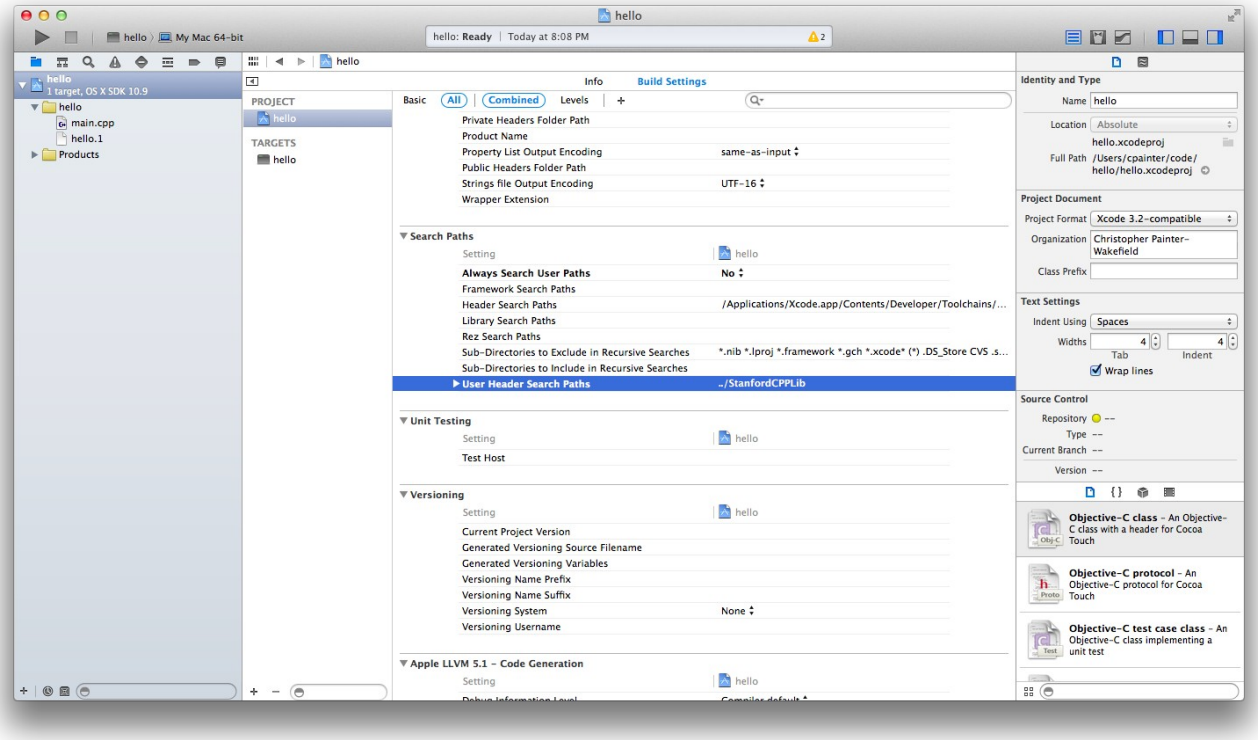
Include Paths

If you need to include files from another directory, you will need to tell Xcode where to look for the include files. Start by bringing up the project settings page. The center pane will look something like

the picture below. In the left portion of the center pane you can select either the project or a target; select the project.

Then select “Build Settings” at the top of the pane. Below “Build Settings”, make sure both “All” and “Combined” are selected. Now scroll the center pane down until you find a section titled “Search Paths”. Click on the space to the right of “User Header Search Paths” and wait until it lets you enter a new location. You can now enter a location where the compiler should look for header files in addition to the directory where your source code is. You can enter multiple locations if you separate them by a space.

You can use relative or absolute path names. So, for instance, you might enter something like “../myincludes /usr/local/include”.

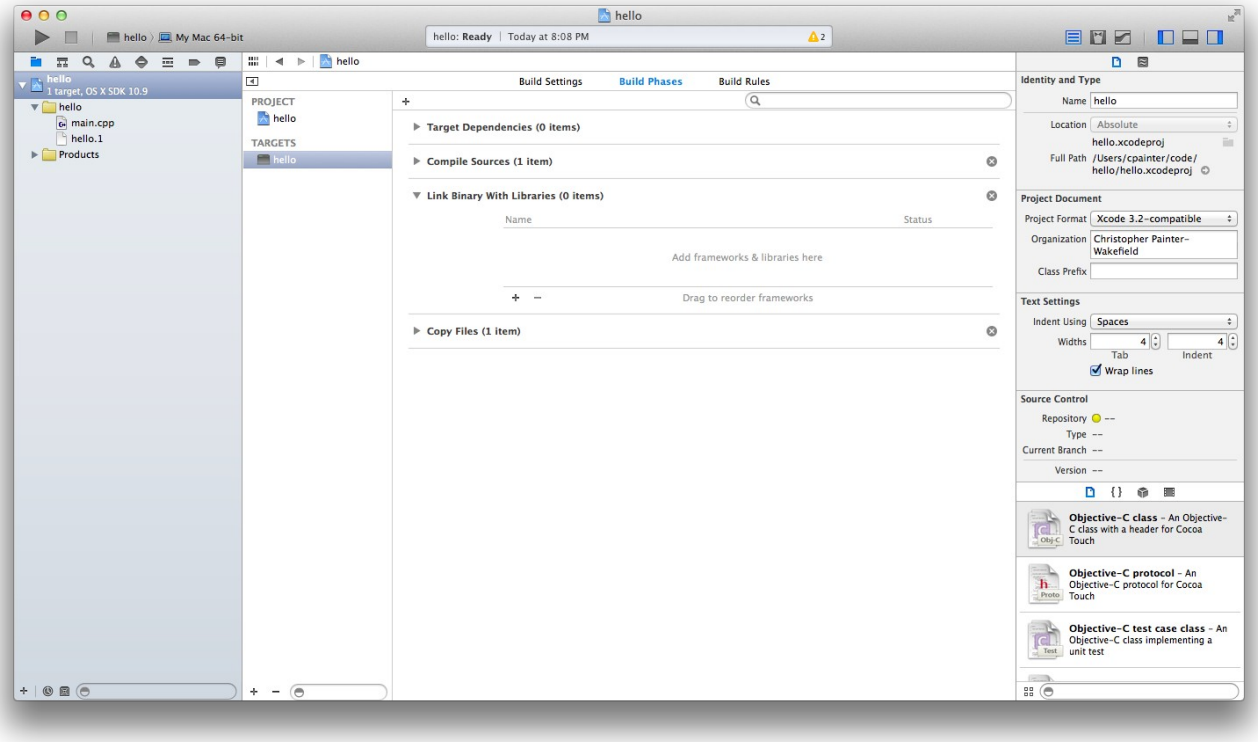


Linking with Libraries

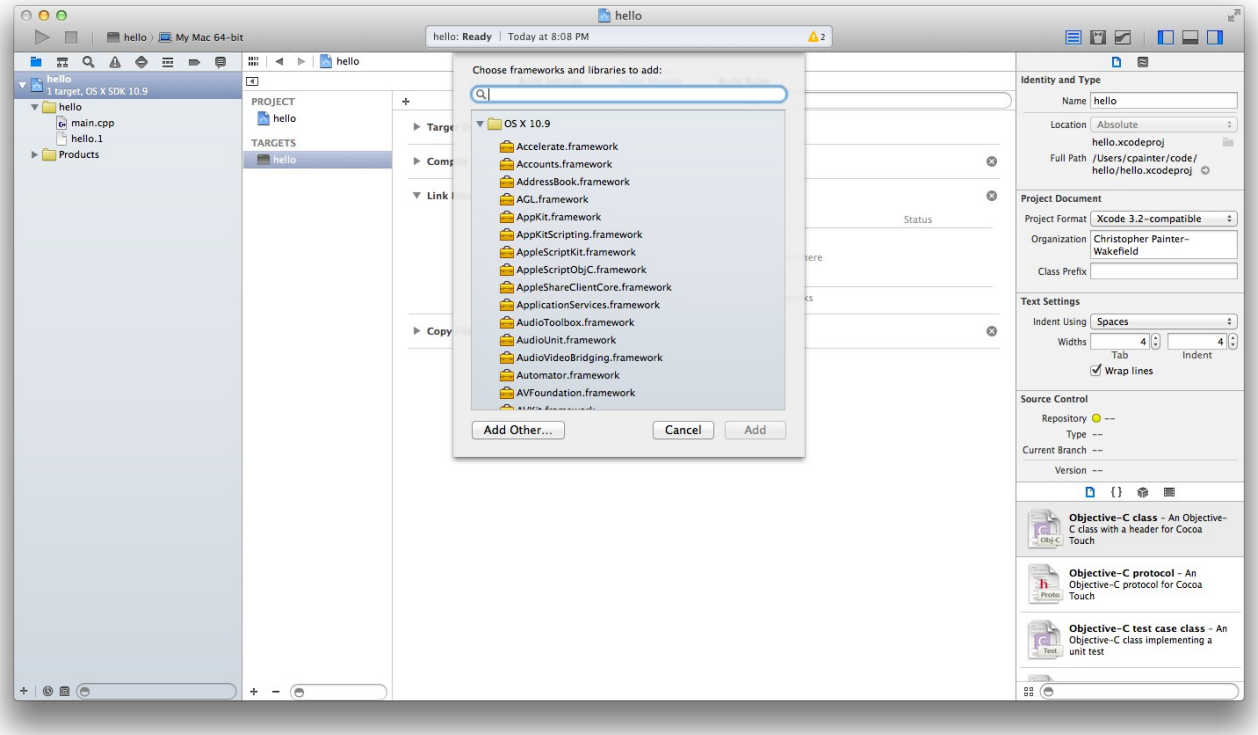
Before beginning, you should read the section on “Using g++ or clang on the Command Line” to familiarize yourself with how the compiler interprets command line options for selecting and locating libraries to link with. In principle, adding third-party libraries to your project under Xcode should be simple; but (at least in version 5.1.1) there are some glaring limitations to what you can do easily via the GUI, making it necessary to know how to do things via the command line.

The easiest cases occur when you need to link with a dynamic library that is in a standard location (e.g., /usr/lib) or with a static library. In these cases, you can just use the GUI. To start, bring up the project settings page. In the left portion of the center pane you can select either the project or a target; select the **target** (you should only have one).

Then select “Build Phases” at the top of the pane. The window should look something like the picture below:



Expand the section titled “Link Binary With Libraries”. To add a library to your project, click on the “+” symbol at the bottom of the section. That will bring up this screen:



If you are adding a library or framework in a standard location, you should find it in the list provided, and can select it and click “Add”. If you want to add a static library, then you need to click on “Add Other...”. Using “Add Other...” will give you a dialog box allowing you to navigate to or find the library you want to add. After either operation you should return to the “Build Phases” settings and see your library listed in the “Link Binary With Libraries” section.

The tricky part comes in if you want to link with a third-party dynamic library that doesn't live in /usr/lib. Some libraries, for instance, install to /usr/local/lib, which is a commonly recognized location for third-party dynamic libraries. However, Xcode doesn't seem to know about that location, and won't easily let you add libraries in that location. In this case, you just tweak the build settings to make the g++ compile do what you want it to.

To do this, start by bringing up the project settings page and select the project in the left portion of the center pane. Then select “Build Settings” at the top of the pane. Below “Build Settings”, make sure both “All” and “Combined” are selected. You need to modify entries in two sections. First, in the “Search Paths” section, modify “Library Search Paths” to add any directories where your dynamic libraries live, e.g., “/usr/local/lib”. Next, in the “Linking” section, modify the “Other Linker Flags” by adding “-l” options as you would for g++ on the command line; for instance, if you have the dynamic library file “libfoo.dylib” that you want to link in, you would add the string “-lfoo” to the “Other Linker Flags” field.

Quick Command Line Tutorial

If you are new to using the linux or Mac OS X command line, here are few tips to get you started.

First, when you are in a terminal window (or more properly, in a “shell”), there is a notion of “current working directory,” which identifies a particular folder in the file system where you can be said to “be”. You can find out where you are using the command `pwd`. Just type it into the command line (without the period) and hit enter and it will display your current working directory.

Your working directory is a hierarchy of directories hanging off of the root directory. The root directory is simply labeled `/` (a single forward slash). Each level below root is set off by another `/`. So for instance, if `pwd` tells you you are in `/Users/christopher`, then you are in a directory named “christopher” which is contained in a directory named “Users”, which is located in the root directory.

To see what is in your current directory, use the command `ls`. Your directory may contain files or other directories. To go into a subdirectory named “foo”, enter the command `cd foo`. To go up one level in the hierarchy, enter

```
cd ..
```

That's `cd` followed by two periods. Two periods is the Unix alias for “my parent directory”. Your current directory also has an alias; it is just a single period.

You can also go straight to a location if you know its full path. Simply enter `cd` followed by the full path, starting with the forward slash:

```
cd /Users/christopher/code/hello
```

To run a program that is in your current working directory, say the program “hello”, type a period followed by a forward slash and the name of the program and hit enter:

```
./hello
```

This will launch the program hello. If you forget the “`./`”, your computer will complain (probably) that it cannot find the program hello. You can also give the full path to a program to launch it:

```
/Users/christopher/code/hello/hello
```

Some other commands you may wish to know about:

`mv`	move or rename a file
`cp`	copy a file
`rm`	delete a file

To find out how to use these and other commands (including more advanced options for `ls`, for example), you can use the `man` program. Start by entering the command

```
man man
```

to find out how to use `man`.