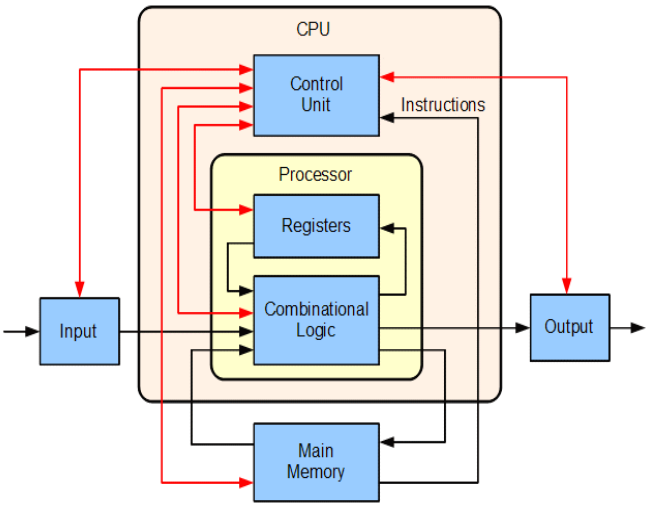
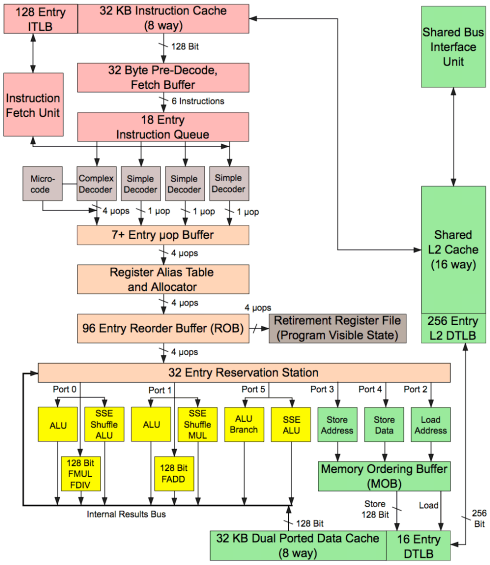
## Basic Structure of a Computer




Intel Core 2 Architecture

## What is a compiler?

Computers understand only one language and that language consists of sets of instructions made of ones and zeros. This computer language is appropriately called *machine language*. A single instruction to a computer could look like this:

| | |
|---|---|
| 00000 | 10011110 |

A particular computer's machine language program that allows a user to input two numbers, adds the two numbers together, and displays the total could include these machine code instructions:

| | |
|---|---|
| 00000 | 10011110 |
| 00001 | 11110100 |
| 00010 | 10011110 |
| 00011 | 11010100 |
| 00100 | 10111111 |
| 00101 | 00000000 |

Programming a computer directly in machine language using only ones and zeros is very tedious and error prone. To make programming easier, high level languages have been developed. High level programs also make it easier for programmers to inspect and understand each other's programs easier.

This is a portion of code written in C++ that accomplishes the exact same purpose:

```
1 int a, b, sum;
2
```

```
3  cin >> a;
4  cin >> b;
5
6  sum = a + b;
7  cout << sum << endl;
```

Even if you cannot really understand the code above, you should be able to appreciate how much easier it will be to program in the C++ language as opposed to machine language.

Because a computer can only understand machine language and humans wish to write in high level languages high level languages have to be re-written (translated) into machine language at some point. This is done by special programs called compilers, interpreters, or assemblers that are built into the various programming applications.

C++ is designed to be a compiled language, meaning that it is generally translated into machine language that can be understood directly by the system, making the generated program highly efficient. For that, a set of tools are needed, known as the development toolchain, whose core are a compiler and its linker.

## The Compilation Process

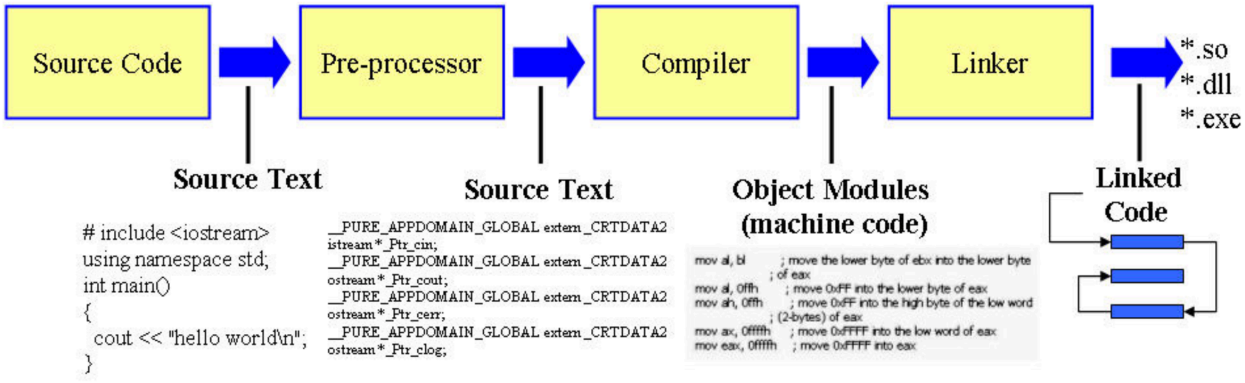



**Figure 1 The phases of compiling and linking a C++ program**

The major phases of building a C++ program are shown in Figure 1. The process starts with the C++ source file, which contains ASCII text encoding statements in the C++ language syntax. There are also preprocessor directives indicated by the # symbol. The preprocessor expands the directives such as including files into additional lines of C++. The C++ compiler is then invoked which generates statements in the machine language of the target computer, such as an Intel CPU. This representation is called object code. Each object module has a set of symbols defined that may not exist within the module itself. These symbols correspond to variables or entry points of functions. The final stage is linking, where these symbols are resolved so that variables can be accessed as well as calls to functions. The whole process is called “making” the program and is directed by some form of “makefile.” In Linux and Unix the process is directed by a utility called make, which operates on makefiles. Under MS Windows Visual C++, and under OSX Xcode, the process is directed by project files.

## Console programs

Console programs are programs that use text to communicate with the user and the environment, such

as printing text to the screen or reading input from a keyboard. Console programs are easy to interact with, and generally have a predictable behavior that is identical across all platforms. They are also simple to implement and thus are very useful to learn the basics of a programming language: The examples in these tutorials are all console programs. The way to compile console programs depends on the particular tool you are using. The easiest way for beginners to compile C++ programs is by using an Integrated Development Environment (IDE). An IDE generally integrates several development tools, including a text editor and tools to compile programs directly from it.

If you happen to have a Linux or Mac environment with development features, you should be able to compile any of the examples directly from a terminal just by including C++11 flags in the command for the compiler:

| Compiler | Platform | Command |
|---|---|---|
| **GCC** | Linux, among others... | g++ -std=c++0x example.cpp -o example_program |
| **Clang** | OS X, among others... | clang++ -std=c++11 -stdlib=libc++ example.cpp -o example_program |

## Assignment 00: Set up the development environment, compile and install the HelloWorld C++ applications

The first order of business is to create a tool chain to develop C++ applications. This assignment is not to be handed-in. It will not be graded.

### OSX

We will use Xcode to compile C++ application. Xcode can be downloaded for free from the App Store. If you already have an earlier version of Xcode installed in your computer, upgrade it to the latest version. We will also use the Command Line Tools Package, comes bundled with Xcode. This is a small self-contained package available for download separately from Xcode and that allows you to do command line development in OS X. It consists of two components: OS X SDK and command-line tools such as Clang, which are installed in /usr/bin. Once Xcode is installed, start it and accept the license agreement.

We will use Homebrew (http://brew.sh) as a package manager, to install and to keep up to date additional tools. If you don't have Homebrew installed in your machine, just visit the Homebrew web page and follow the instructions. Open a Terminal, cut the command from the Homebrew web page and paste at the terminal prompt, run it, and then wait a few minutes for the installation to complete. Once Homebrew is installed, run in the same terminal the following command

```
brew doctor
```

and fix any detected errors. Then run

```
brew update
brew upgrade
```

We will also use CMake (http://www.cmake.org), the cross-platform, open-source build system developed by Kitware. You can use Homebrew to install CMake. Just run the following command

```
brew install cmake
```

or visit the CMake Download page (http://www.cmake.org/download/), and download the following file

```
cmake-3.9.1-Darwin-x86_64.dmg
```

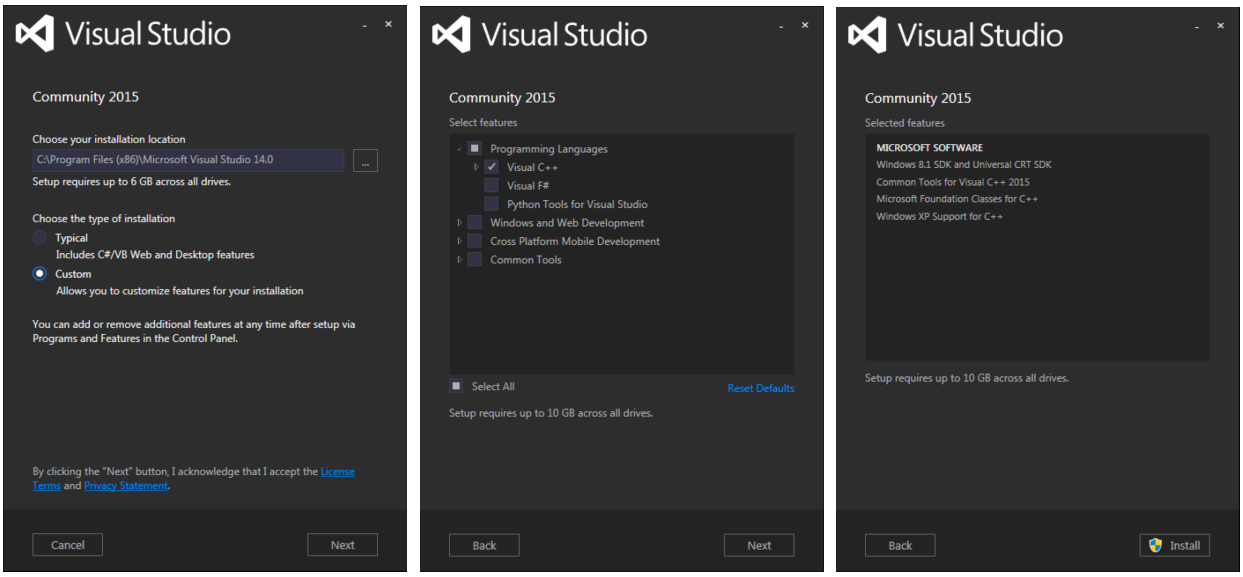
Once the download is completed, double-click on the file in Finder and follow the instructions.

### Windows

We will use Visual Studio to compile C++ applications. You can use an older version of Visual Studio if you already have it installed in your computer. Otherwise Visual Studio Community 2017 can be downloaded for free from the following web site

https://www.visualstudio.com/products/visual-studio-community-vs

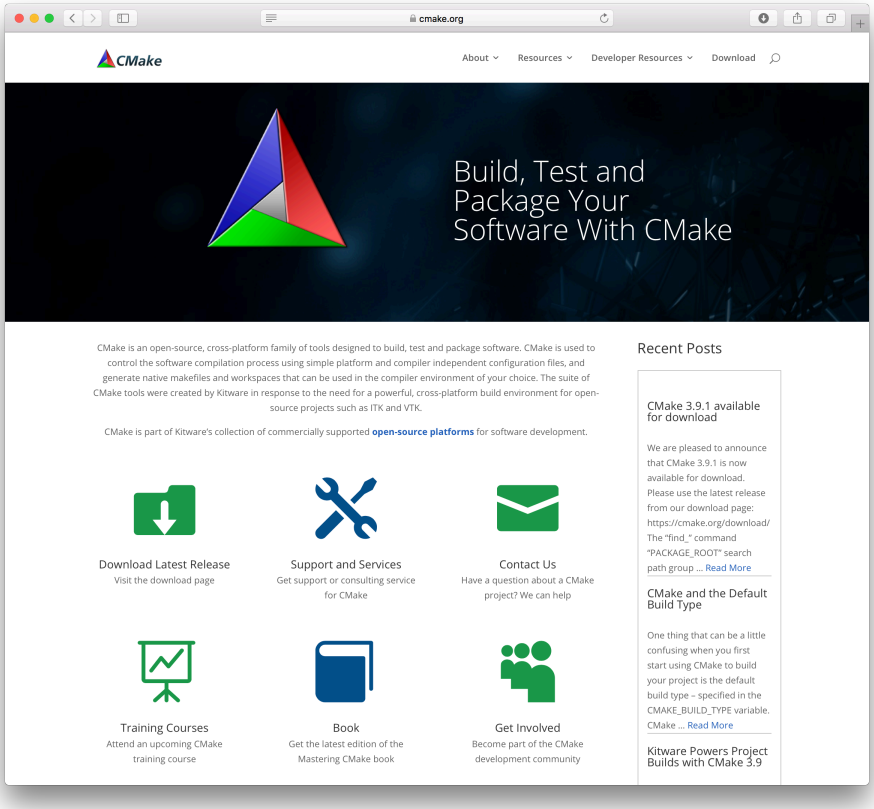
Run the downloaded file (vs_community.exe) and select Custom as the type of installation. In the next panel select only Visual C++ and deselect everything else that may have been pre-selected. In the next panel press the Install button, and wait until the installation process finishes, which will take some time.



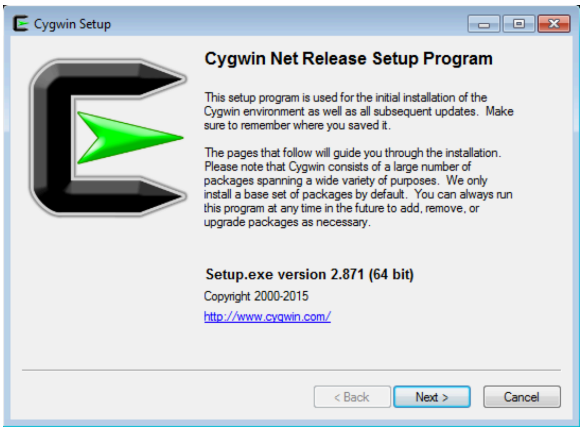

You also use CMake (http://www.cmake.org), the cross-platform, open-source build system developed by Kitware. Visit the CMake Download page (http://www.cmake.org/download/), and download the following file

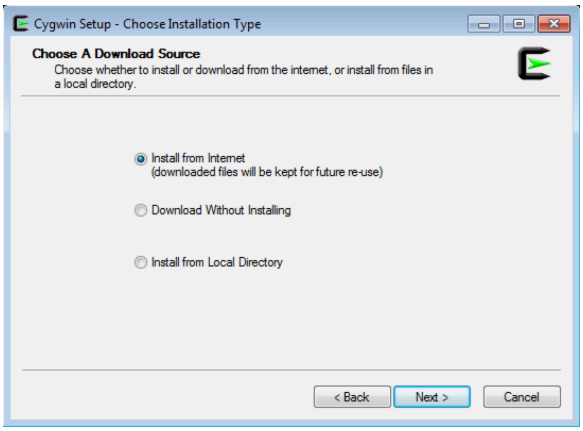
`cmake-3.9.1-win32-x86.exe`

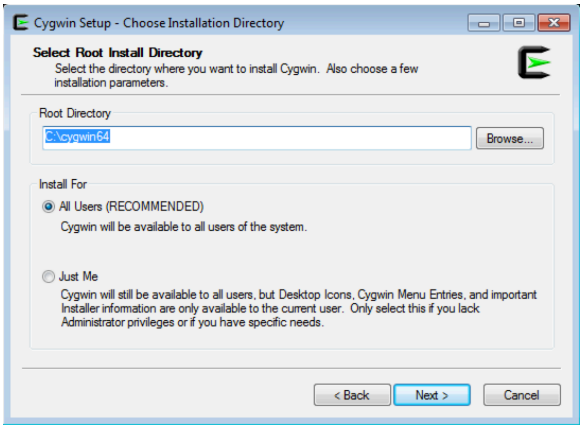
Once the download is completed, double-click on the file in Windows Explorer and follow the instructions.

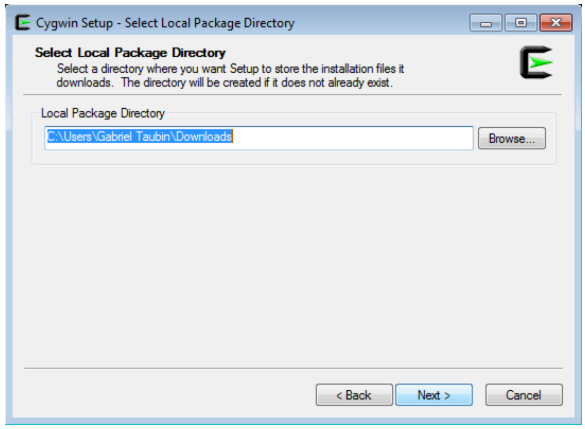


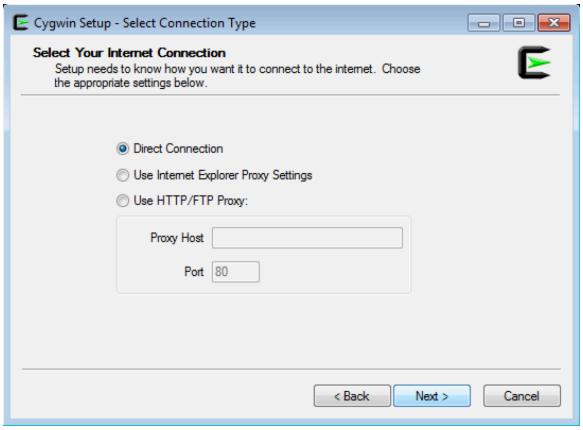
Since Windows does not include a UNIX-like terminal, we will install Cygwin (https://www.cygwin.com). The most recent version of the Cygwin DLL is 2.8.2. Install it by running setup-x86.exe (32-bit installation) or setup-x86_64.exe (64-bit installation). Use the setup program to perform a fresh install or to update an existing installation. Select all the defaults, select a download site, then in the packages step you should select three important packages (cmake, make and gcc-g++) you can search for them and expand the Devel category to find them. Run the installation process, which will take some time to complete. Do not delete the setup executable, since you will use it later to install additional packages.

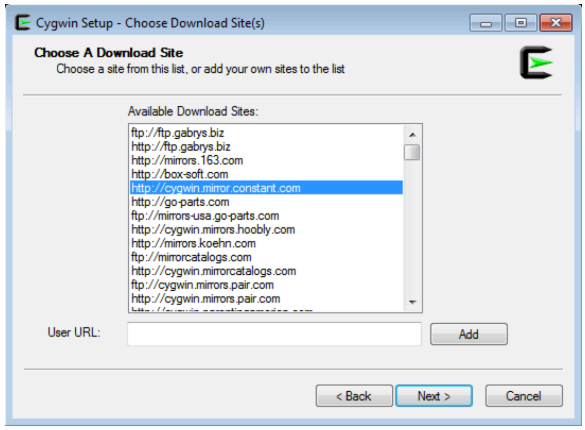


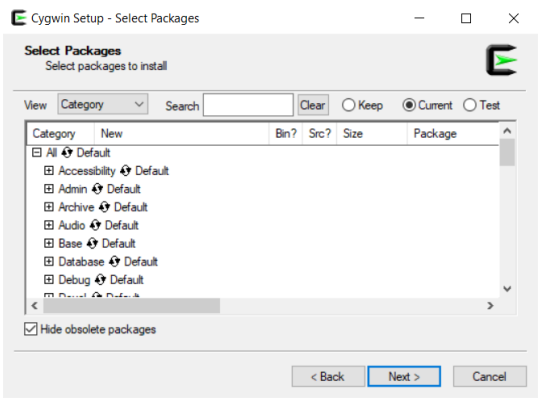



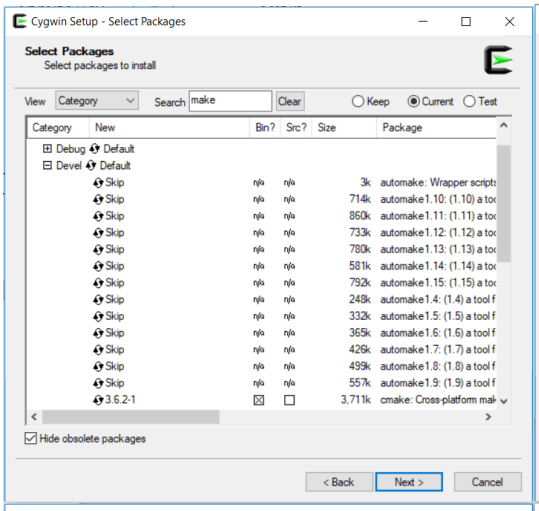



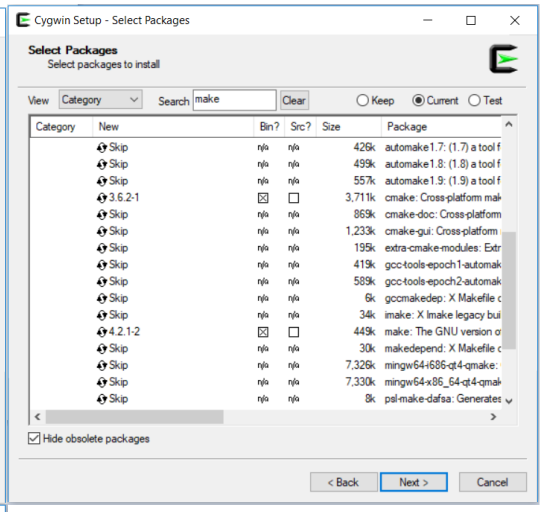



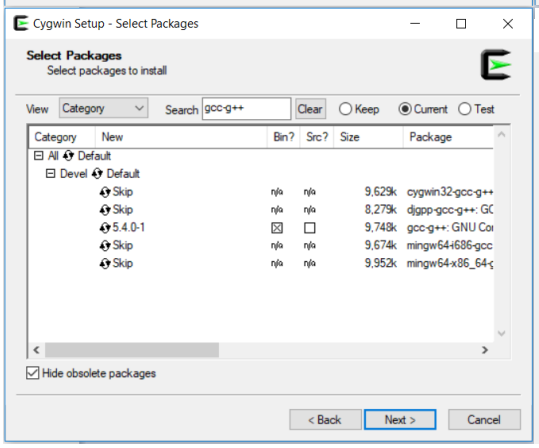



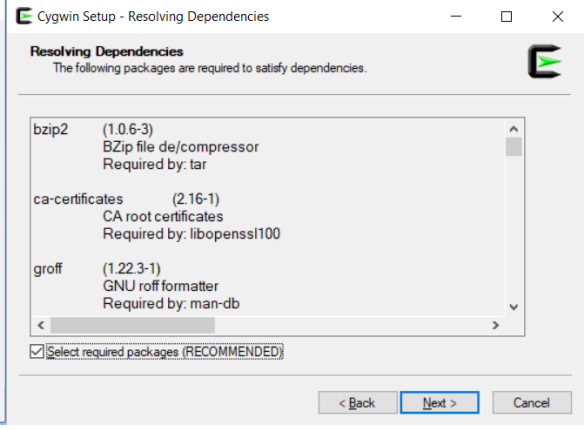




Cygwin Setup - Select Packages
Select Packages
Select packages to install
View
Category
Search
Clear
Keep
Current
Test
Category
New
Bin?
Src?
Size
Package
All Default
Accessibility Default
Admin Default
Archive Default
Audio Default
Base Default
Database Default
Debug Default
Hide obsolete packages
< Back
Next >
Cancel


Cygwin Setup - Select Packages
Select Packages
Select packages to install
View
Category
Search
make
Clear
Keep
Current
Test
Category
New
Bin?
Src?
Size
Package
Debug Default
Devel Default
Skip
automake: Wrapper script
automake1.10: (1.10) a to
automake1.11: (1.11) a to
automake1.12: (1.12) a to
automake1.13: (1.13) a to
automake1.14: (1.14) a to
automake1.15: (1.15) a to
automake1.4: (1.4) a tool f
automake1.5: (1.5) a tool f
automake1.6: (1.6) a tool f
automake1.7: (1.7) a tool f
automake1.8: (1.8) a tool f
automake1.9: (1.9) a tool f
3.6.2-1
cmake: Cross-platform mak
Hide obsolete packages
< Back
Next >
Cancel


Cygwin Setup - Select Packages
Select Packages
Select packages to install
View
Category
Search
make
Clear
Keep
Current
Test
Category
New
Bin?
Src?
Size
Package
Skip
automake1.7: (1.7) a tool f
automake1.8: (1.8) a tool f
automake1.9: (1.9) a tool f
3.6.2-1
cmake: Cross-platform mak
cmake-doc: Cross-platform
cmake-gui: Cross-platform
extra-cmake-modules: Extr
gcc-tools-epoch1-automak
gcc-tools-epoch2-automak
gccmakedep: X Makefile
imake: X Imake legacy bui
4.2.1-2
make: The GNU version o
makedepend: X Makefile
mingw64-i686-qt4-qmake:
mingw64-x86_64-qt4-qmak
psl-make-dafsa: Generates
Hide obsolete packages
< Back
Next >
Cancel


Cygwin Setup - Select Packages
Select Packages
Select packages to install
View
Category
Search
gcc-g++
Clear
Keep
Current
Test
Category
New
Bin?
Src?
Size
Package
All Default
Devel Default
Skip
9,629k
cygwin32-gcc-g++
8,279k
djgpp-gcc-g++: GC
5.4.0-1
9,748k
gcc-g++: GNU Co
9,674k
mingw64-i686-gcc
9,952k
mingw64-x86_64-g
Hide obsolete packages
< Back
Next >
Cancel


Cygwin Setup - Resolving Dependencies
Resolving Dependencies
The following packages are required to satisfy dependencies.
bzip2 (1.0.6-3)
BZip file de/compressor
Required by: tar
ca-certificates (2.16-1)
CA root certificates
Required by: libopenssl100
groff (1.22.3-1)
GNU roff formatter
Required by: man-db
Select required packages (RECOMMENDED)
< Back
Next >
Cancel

## Developing the first C++ application: Hello World

Whether you are working in a Windows or a OSX environment, create a directory for your projects, and a subdirectory within the projects for this particular project. If you open a terminal window, you will get a prompt in your home directory. We will assume that your projects directory is a subdirectory of your home directory (`~/Projects`), and we will name the directory for this project using the assignment number followed by your Brown email account name (`~/Projects/00_Gabriel_Taubin`). Create 3 subdirectories named `src`, `build`, and `bin`, so that the complete paths to the three subdirectories are `~/Projects/00_Gabriel_Taubin/src`, `~/Projects/00_GabrielTaubin/build`, and `~/Projects/00_Gabriel_Taubin/bin` (note that in the pictures shown bellow, instead of `00_Gabriel_Taubin`, the directory name `20150914` is used).

In the `src` subdirectory create a text file named `HelloWorld.cpp` containing the following lines (without the line numbers).

```
1. // HelloWorld.cpp
2. #include <iostream>
3. int main(int argc, const char * argv[]) {
4.   std::cout << "Hello, World!\n";
5.   return 0;
6. }
```

Once compiled, this file will turn into an executable command line application, which when run within a terminal will print "`Hello, World!`", followed by a carriage return, and will subsequently exit with no errors.

We are concerned now with setting up the compilation process, compiling and installing the application. The `build` directory will be used for temporary storage for the compilation and debugging processes. The application will eventually be installed in the `bin` directory. It is a good practice to keep the source files, the temporary build files, and the final applications in separate directories. After the application is installed, all the contents of the `build` directory can be deleted. In fact, at any time the contents of the `build` and `bin` directories can be deleted, and the application can be rebuilt from the source.

Also in the `src` subdirectory create a text file named `CMakeLists.txt` containing the following lines (again, without the line numbers).

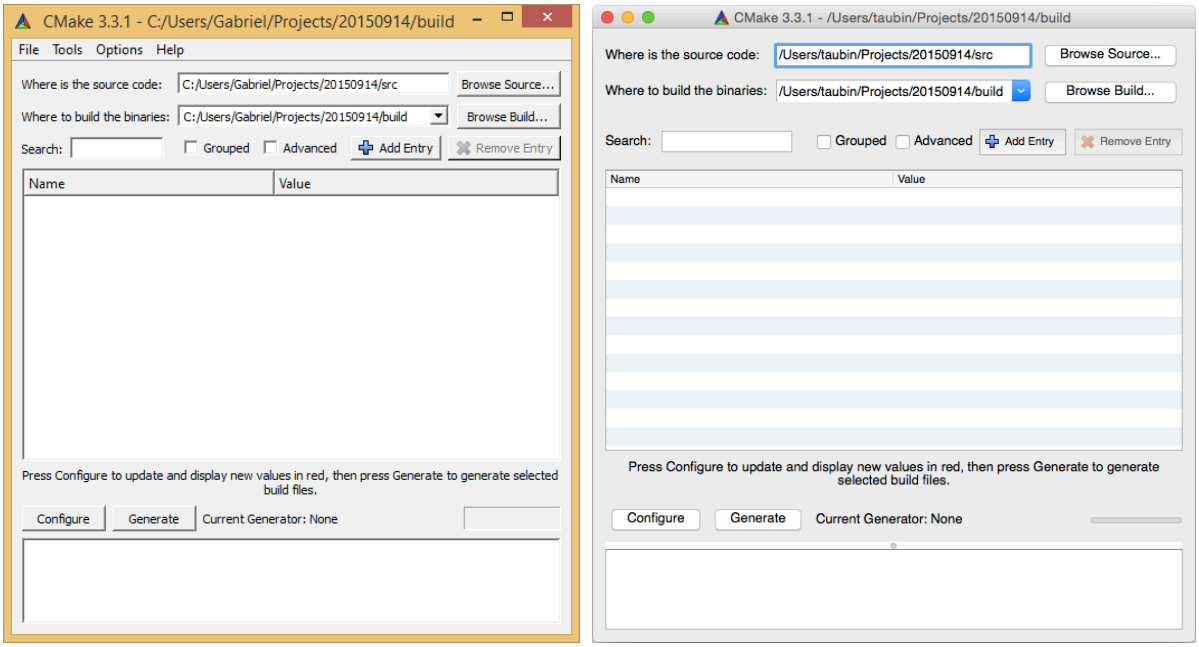
```
1. PROJECT(HelloWorld)
2. cmake_minimum_required(VERSION 2.8)
3. # list of source files
4. set(hello_world_files HelloWorld.cpp)
5. # define the executable
6. if(WIN32)
7.   add_executable(helloWorld WIN32 ${hello_world_files})
8. else()
9.   add_executable(helloWorld ${hello_world_files})
10. endif()
11. # in Windows + Visual Studio we need this to make it a console application
12. if(WIN32)
13.   if(MSVC)
14.     set_target_properties(helloWorld PROPERTIES LINK_FLAGS "/SUBSYSTEM:CONSOLE")
15.   endif(MSVC)
16. endif(WIN32)
17. # install application
```

```
18. set(BIN_DIR ${CMAKE_INSTALL_PREFIX}/bin)
19. install(TARGETS helloWorld DESTINATION ${BIN_DIR})
```

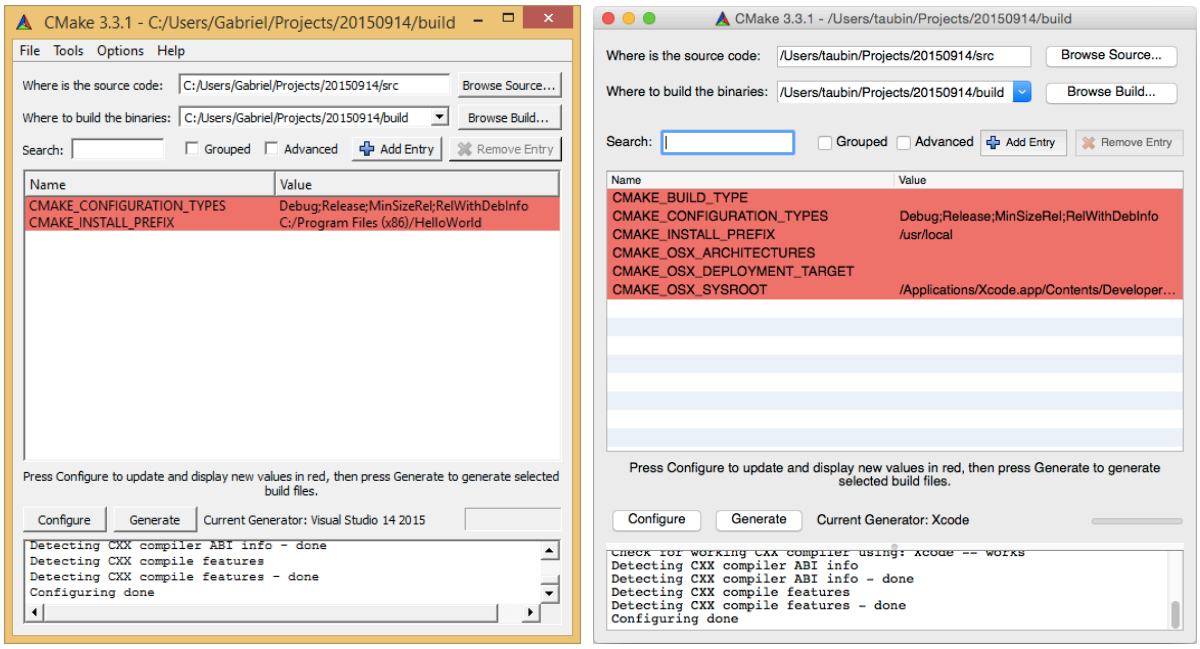
The same file will be used by CMake to create the project files necessary to compile the application in Windows or OSX

**CMAKE**

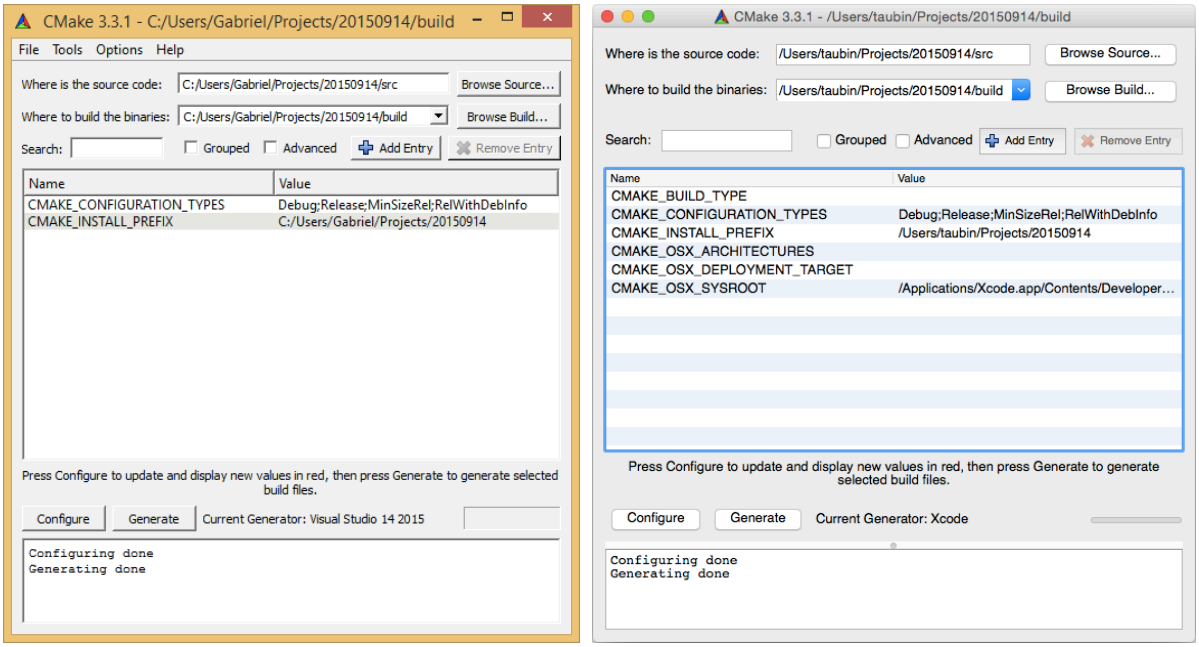
Run the CMake application, and enter the full path of the src and build directories. This is how the CMake application looks in Windows and OSX.




Press the Configure button. A new window will pop up. In this window select “Visual Studio 15 2017” as the generator for the project in Windows, and “Xcode” in OSX, select “Use default native compilers”, press the “Done” or “Finish” button, and wait until the configuration process completes. The configuration process is guided by the content of the CMakeLists.txt file. CMake reports syntax errors in the CMakeLists.txt file in the lower panel. This is how the CMake window looks like after the configuration process completes without errors.

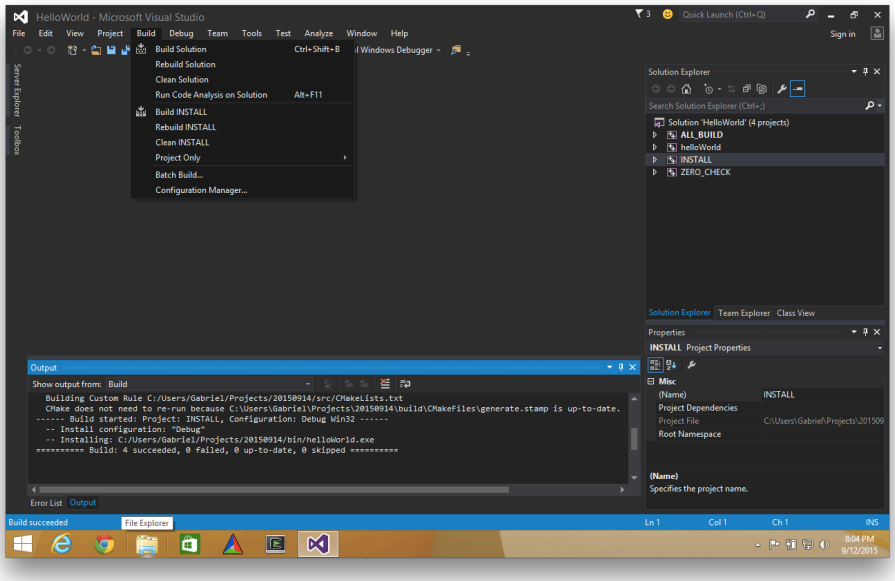



At this point we need to fix the value of the `CMAKE_INSTALL_PREFIX` variable, since the default value is not appropriate and the compilation process may not even have permission to write in those directories. We will use the project directory `~/Projects/20150914` instead (actually `~/Projects/00_Gabriel_Taubin` this year). Click on the value `CMAKE_INSTALL_PREFIX` variable and type the proper value. You can copy the value from the src or built directory paths and paste the value. Then press the "Configure" button again, and then press the "Generate" button. This is how the CMake window looks like after these steps.

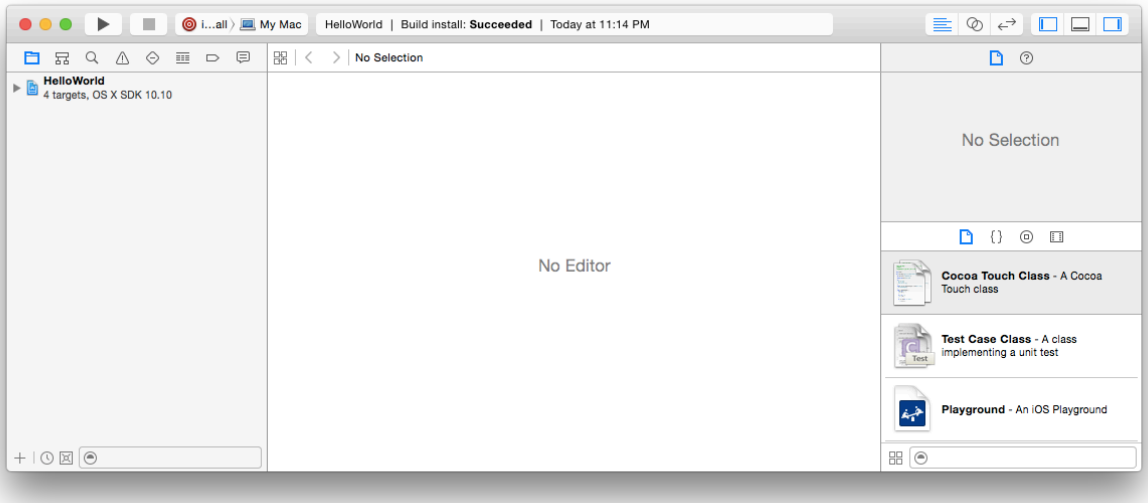



After CMake creates the project files, you can close the CMake window. Then go to the `~/Projects/20170914/build` directory (`~/Projects/00_Gabriel_Taubin/build`) and open the corresponding project file in your IDE.

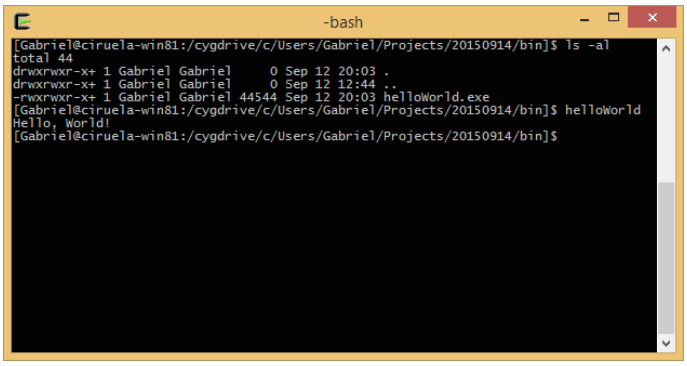
In the Windows environment run Visual Studio by double-clicking on the Microsoft Visual Studio Solution file `HelloWorld.sln` in Windows Explorer. Select the INSTALL project in the Solution Explorer window, and Build INSTALL from the Build pull-down menu.

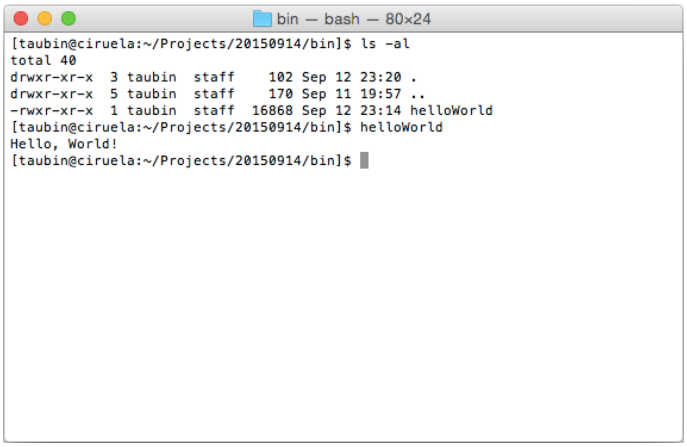



In the OSX environment run Xcode by double-clicking on the HelloWorld.xcodeproj in Finder. Set the Install scheme as the active scheme (in the upper left corner), and press the Build button in the Product pull-down menu.




In both cases the linker should produce an executable file, which it would save in the `bin` directory. In the Windows environment, the executable file is named `helloWorld.exe`, and in OSX just `helloWorld`. Type `helloWorld` at the prompt, and press the Return key.


-bash
[Gabriel@ciruela-win81:/cygdrive/c/Users/Gabriel/Projects/20150914/bin]$ ls -al
total 44
drwxrwxr-x+ 1 Gabriel Gabriel     0 Sep 12 20:03 .
drwxrwxr-x+ 1 Gabriel Gabriel     0 Sep 12 12:44 ..
-rwxrwxr-x+ 1 Gabriel Gabriel 44544 Sep 12 20:03 helloWorld.exe
[Gabriel@ciruela-win81:/cygdrive/c/Users/Gabriel/Projects/20150914/bin]$ helloWorld
Hello, World!
[Gabriel@ciruela-win81:/cygdrive/c/Users/Gabriel/Projects/20150914/bin]$


bin — bash — 80×24
[taubin@ciruela:~/Projects/20150914/bin]$ ls -al
total 40
drwxr-xr-x  3 taubin  staff    102 Sep 12 23:20 .
drwxr-xr-x  5 taubin  staff    170 Sep 11 19:57 ..
-rwxr-xr-x  1 taubin  staff  16868 Sep 12 23:14 helloWorld
[taubin@ciruela:~/Projects/20150914/bin]$ helloWorld
Hello, World!
[taubin@ciruela:~/Projects/20150914/bin]$

### The `CMakeLists.txt` file

```
1. PROJECT(HelloWorld)
2. cmake_minimum_required(VERSION 2.8)
3. # list of source files
4. set(hello_world_files HelloWorld.cpp)
5. # define the executable
6. if(WIN32)
7.   add_executable(helloWorld WIN32 ${hello_world_files})
8. else()
9.   add_executable(helloWorld ${hello_world_files})
10. endif()
11. # in Windows + Visual Studio we need this to make it a console application
12. if(WIN32)
13.   if(MSVC)
14.     set_target_properties(helloWorld PROPERTIES LINK_FLAGS "/SUBSYSTEM:CONSOLE")
15.   endif(MSVC)
16. endif(WIN32)
17. # install application
18. set(BIN_DIR ${CMAKE_INSTALL_PREFIX}/bin)
19. install(TARGETS helloWorld DESTINATION ${BIN_DIR})
```

Lines 3, 5, 11, and 17 are comment lines. Any line that starts with a # symbol is comment line, and it is ignored by CMake. Line 1 defines the name of the project as HelloWorld. Line 2 defines the minimum version of CMake required to parse this file, since the syntax has changed from version to version. Line 4 defines the variable hello_world_files as a list of file names, containing a single file HelloWorld.ccp in this case. WIN32 is a binary variable set up by CMake which holds the value TRUE in a Windows environment, and FALSE otherwise. Lines 7 and 9 state that the executable helloWorld.exe or helloWorld, depending on the environment, should be compiled as a function of the files listed in the variable hello_world_files. MSVC is a binary variable set up by CMake which holds the value TRUE if the development environment is Visual Studio. The default application type for compilation of applications in Visual Studio is an interactive windows application. Since this is a command line application, Line 14 sets the linker flag /SUBSYSTEM:CONSOLE. Otherwise the compilation fails. Line 18 defines the path to the directory where executable files should be installed. And line 19 states that the executable helloWorld should be installed in this directory.

**Structure of a Program**

The best way to learn a programming language is by writing programs. Typically, the first program beginners write is a program called "Hello World", which simply prints "Hello World" to your computer screen. Although it is very simple, it contains all the fundamental components C++ programs have:

```
1 // my first program in C++
2 #include <iostream>
3
4 int main()
5 {
6   std::cout << "Hello World!";
7 }
```

```
Hello World
```

The left panel above shows the C++ code for this program. The right panel shows the result when the program is executed by a computer. The grey numbers to the left of the panels are line numbers to make discussing programs and researching errors easier. They are not part of the program.

Let's examine this program line by line:

Line 1: // my first program in C++
Two slash signs indicate that the rest of the line is a comment inserted by the programmer but which has no effect on the behavior of the program. Programmers use them to include short explanations or observations concerning the code or program. In this case, it is a brief introductory description of the program.

Line 2: #include <iostream>
Lines beginning with a hash sign (#) are directives read and interpreted by what is known as the *preprocessor*. They are special lines interpreted before the compilation of the program itself begins. In this case, the directive #include <iostream>, instructs the preprocessor to include a section of standard C++ code, known as *header iostream*, that allows to perform standard input and output operations, such as writing the output of this program (Hello World) to the screen.

Line 3: A blank line.
Blank lines have no effect on a program. They simply improve readability of the code.

Line 4: int main ()
This line initiates the declaration of a function. Essentially, a function is a group of code statements which are given a name: in this case, this gives the name "main" to the group of code statements that follow. Functions will be discussed in detail in a later chapter, but essentially, their definition is introduced with a succession of a type (int), a name (main) and a pair of parentheses (()), optionally including parameters.

The function named main is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the main function, regardless of where the function is actually located within the code.

Lines 5 and 7: { and }
The open brace ({) at line 5 indicates the beginning of main's function definition, and the closing brace (}) at line 7, indicates its end. Everything between these braces is the function's body that defines what happens when main is called. All functions use braces to indicate the beginning and end of their definitions.

Line 6: std::cout << "Hello World!";
This line is a C++ statement. A statement is an expression that can actually produce some effect. It is the meat of a program, specifying its actual behavior. Statements are executed in the same order that they appear within a function's body.

This statement has three parts: First, std::cout, which identifies the **st**andar**d** **c**haracter **out**put device (usually, this is the computer screen). Second, the insertion operator (<<), which indicates that what follows is inserted into std::cout. Finally, a sentence within quotes ("Hello world!"), is the content inserted into the standard output.

Notice that the statement ends with a semicolon (;). This character marks the end of the statement, just as the period ends a sentence in English. All C++ statements must end with a semicolon character. One of the most common syntax errors in C++ is forgetting to end a statement with a semicolon.

You may have noticed that not all the lines of this program perform actions when the code is executed. There is a line containing a comment (beginning with //). There is a line with a directive for the preprocessor (beginning with #). There is a line that defines a function (in this case, the main function). And, finally, a line with a statements ending with a semicolon (the insertion into cout), which was within the block delimited by the braces ( { } ) of the main function.

The program has been structured in different lines and properly indented, in order to make it easier to understand for the humans reading it. But C++ does not have strict rules on indentation or on how to split instructions in different lines. For example, instead of

```
1  int main ()
2  {
3    std::cout << " Hello World!";
4  }
```

We could have written:

```
int main () { std::cout << "Hello World!"; }
```

all in a single line, and this would have had exactly the same meaning as the preceding code.

In C++, the separation between statements is specified with an ending semicolon (;), with the separation into different lines not mattering at all for this purpose. Many statements can be written in a single line, or each statement can be in its own line. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it, but has no effect on the actual behavior of the program.

Now, let's add an additional statement to our first program:

```
1  // my second program in C++
2  #include <iostream>
3
4  int main () {
5   std::cout << "Hello World! ";
6   std::cout << "I'm a C++ program";
7  }
8
```

```
Hello World! I'm a C++ program
```

In this case, the program performed two insertions into std::cout in two different statements. Once again, the separation in different lines of code simply gives greater readability to the program, since main could have been perfectly valid defined in this way:

```
int main () { std::cout << " Hello World! "; std::cout << " I'm a C++ program "; }
```

The source code could have also been divided into more code lines instead:

```
1  int main ()
2  {
3    std::cout <<
4     "Hello World!";
5    std::cout
6      << "I'm a C++ program";
7  }
```

And the result would again have been exactly the same as in the previous examples.

Preprocessor directives (those that begin by #) are out of this general rule since they are not statements. They are lines read and processed by the preprocessor before proper compilation begins. Preprocessor directives must be specified in their own line and, because they are not statements, do not have to end with a semicolon (;).

### Comments

As noted above, comments do not affect the operation of the program; however, they provide an important tool to document directly within the source code what the program does and how it operates.

C++ supports two ways of commenting code:

```
1 // line comment
2 /* block comment
   * ...
   */
```

The first of them, known as *line comment*, discards everything from where the pair of slash signs (//) are found up to the end of that same line. The second one, known as *block comment*, discards everything between the /* characters and the first appearance of the */ characters, with the possibility of including multiple lines.

Let's add comments to our second program:

```
1 /* my second program in C++
2    with more comments */
3
4 #include <iostream>
5
6 int main () {
7   std::cout << "Hello World! ";    // prints Hello World!
8   std::cout << "I'm a C++ program"; // prints I'm a C++ program
9 }
```

```
Hello World! I'm a C++ program
```

If comments are included within the source code of a program without using the comment characters combinations //, /* or */, the compiler takes them as if they were C++ expressions, most likely causing the compilation to fail with one, or several, error messages.

### Using namespace std

If you have seen C++ code before, you may have seen cout being used instead of std::cout. Both name the same object: the first one uses its *unqualified name* (cout), while the second qualifies it directly within the *namespace* std (as std::cout).

cout is part of the standard library, and all the elements in the standard C++ library are declared within what is a called a *namespace*: the namespace std.

In order to refer to the elements in the std namespace a program shall either qualify each and every use of elements of the library (as we have done by prefixing cout with std::), or introduce visibility of its components. The most typical way to introduce visibility of these components is by means of *using declarations*:

```
using namespace std;
```

The above declaration allows all elements in the std namespace to be accessed in an *unqualified* manner (without the std:: prefix).

With this in mind, the last example can be rewritten to make unqualified uses of cout as:

```
1  // my second program in C++
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    cout << "Hello World! ";
8    cout << "I'm a C++ program";
9  }
```

```
Hello World! I'm a C++ program
```

Both ways of accessing the elements of the std namespace (explicit qualification and *using* declarations) are valid in C++ and produce the exact same behavior. For simplicity, and to improve readability, we will more often use this latter approach with *using* declarations, although note that *explicit qualification* is the only way to guarantee that name collisions never happen.

Namespaces are explained in more detail in a later chapter.

### Command Line Compilation

Beside using Visual Studio or XCode to compile the application, it's also important to learn how to compile it using the command line. We will use Cygwin on Windows and default terminal in Mac or Linux systems. You should familiarize yourself with the basic Linux terminal commands first.

Find below an example that we run on a Mac system, it will produce similar results if we run it using Cygwin on Windows. We will follow these steps when grading the assignments as well, so please make sure that your code can be run in this way before submitting.

First we move all our source files to src folder (on Cygwin, the default installation location is C:\cygwin64\home\username. You can create folders in this location using Windows explorer and access them from Cygwin command line).

1) We start with 'src' directory only, move all your files inside it:

```
ammar@MacBook:~/projects/hw1$ ls
src
ammar@MacBook:~/projects/hw1$ ls src/
CMakeLists.txt  HelloWorld.cpp
ammar@MacBook:~/projects/hw1$
```

2) We create 'build' and 'bin' directories:

```
ammar@MacBook:~/projects/hw1$ mkdir build bin
ammar@MacBook:~/projects/hw1$ ls
bin  build  src
```

```
ammar@MacBook:~/projects/hw1$
```

3) We go to the 'build' directory and run cmake

```
ammar@MacBook:~/projects/hw1$ cd build/
ammar@MacBook:~/projects/hw1/ build$ cmake -DCMAKE_INSTALL_PREFIX=.. ../src/
-- The C compiler identification is AppleClang 4.2.0.4250028
-- The CXX compiler identification is AppleClang 4.2.0.4250028
-- Check for working C compiler:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cc
-- Check for working C compiler:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cc -
- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c+
-- Check for working CXX compiler:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c+
+ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/ammar/projects/hw1/build
ammar@MacBook:~/projects/hw1/build$
```

4) We compile and install the code

```
ammar@MacBook:~/projects/hw1/build$ make install
Scanning dependencies of target helloWorld
[ 50%] Building CXX object CMakeFiles/helloWorld.dir/HelloWorld.cpp.o
[ 100%] Linking CXX executable helloWorld
[ 100%] Built target helloWorld
Install the project...
-- Install configuration: ""
-- Installing: /Users/ammar/projects/hw1/bin/helloWorld
ammar@MacBook:~/projects/hw1/build$
```

5) At this point the executable is in the 'bin' and we execute it from the command line:

```
ammar@MacBook:~/projects/hw1/build$ ls ../bin/
helloWorld
ammar@MacBook:~/projects/hw1/build$ ../bin/helloWorld
Hello, World!
```