Introduction to compilation:

Thus far, you've been writing code within the Udacity classroom. But you will also want to be able to run your programs locally on your own computer. This brings up another difference between Python and C++. You learned about the first major difference at the beginning of the C++ lesson; Python is dynamically typed while C++ is statically typed. Another major difference is that Python is an **interpreted language** whereas C++ is a **compiled language**.

When you write code in Python or C++, your computer can't actually understand the code that you are writing. But we humans can read and understand these languages, so they are convenient for us to code in.

Your code needs to be translated into a language that your CPU understands. Interpreted languages and compiled languages get translated in different ways. When you run a Python program, there is a translator (ie an interpreter), that reads a line of your code, translates a line of code for the CPU, and then executes your code line on the CPU. Then the next line gets translated and executed. Then the next, etc. The reality is a bit more complex, but that is the gist of an interpreted language like Python.

A compiled language, on the other hand, translates all of your code into the CPU's language. And then your code gets executed. So when running a C++ program, there is an extra step where you first compile your code and then a second step where you execute your code. Python, on the other hand, only has the execution step.

The C++ code you have been writing in the classroom actually is being compiled first and then executed, but it's happening behind the scenes. If you want to run your C++ programs locally on your computer, you're going to need to first compile the code and then execute it yourself.

In the next part of the lesson, you will get your own computer ready for compiling and executing C++ programs.

**Running Code Locally**

If you haven't already tried running your C++ code locally, now's the time to get things working. In the C++ section, we mentioned how to get code to run locally on your computer. Here are the instructions again for your reference:

Here are some suggested programs for writing and executing your code locally:

* [Sublime Text](https://www.sublimetext.com/) as a general text editor
* Download an IDE (Integrated Development Environment) such as [Xcode](https://developer.apple.com/xcode/downloads/) (Mac only), [Visual Studio](https://www.visualstudio.com/downloads/) or any other IDE.

**Detailed Setup Instructions**

Big thanks to Michael Ikemann! Michael is a student in this Nanodegree and put together some **exceptionally** detailed documentation to help you get started with C++.

* [Windows C++ Setup Guide](https://docs.google.com/document/d/1PA3YcIKpANBZuHLX6tz7bVw60ug_DM1_V_yFKyrMP-s/edit?usp=sharing)
  + Per student Tarun, the guide document for IDE setup suggests moving the .sln file to parent directory, along with the .vcproj file. To avoid errors, the .sln file needs to be hand modified with updated path.
* [Mac C++ Setup Guide](https://docs.google.com/document/d/1FoIc9X0NcM-0y0dgXCzJcpu5lVXGFqS62mb1ABLDT1M/edit)

Below you'll find further instructions as well, although Michael's documents should be enough to get things running.

**Compiling and Executing C++ Code**

If you are on a Windows, Mac, or Linux machine, the basic process is going to be the same; you will compile your code and then execute the compiled code. But the details of how to do this are slightly different on different machines and operating systems.

And there are two different ways you can compile and execute your code:

* using the windows console/mac terminal/linux terminal
* using an IDE (Integrated Development Environment), which is a software program that makes it easier to write, organize, compile and run your code.

**Running your Code through the Console or Terminal**

This is the quickest way to get started compiling and executing your code.

**For Windows**

You will first need to download and install the [Visual C++ Build Tools](https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=BuildTools&rel=15), which are provided by Microsoft. The installation process might vary depending on what version of windows you are using. The microsoft website has a helpful guide about how to make sure the installation went correctly. The guide also explains how to compile and run your code: [microsoft instructions](https://docs.microsoft.com/en-us/cpp/build/walkthrough-compiling-a-native-cpp-program-on-the-command-line).

If the Visual C++ Build Tools are installed, you should be able to compile and execute your code from the console. For example, if you have a main.cpp, you would open the console and navigate to the directory with the file.

You will see that the microsoft tutorial suggests compiling your code with the following command:

cl /W4 /EHsc main.cpp

The /W4 and /EHsc are options that will compile your code with warnings and [error handing](https://docs.microsoft.com/en-us/cpp/build/reference/eh-exception-handling-model).

The compiler outputs an .obj file, which contains the instructions for the CPU. You will also see a .exe file that will get your code running.

Now typing

main

at the command prompt should get your program running.

If your program has multiple .cpp files, then you compile with:

cl /W4 /EHsc file1.cpp file2.cpp file3.cpp

The name of the .obj and .exe files will be file1.

**For Mac**

Depending on your OS version, the installation process will be different. In general, you will need to download the "command line developer tools" from the Apple developer website.

As a first step, see if you already have a compiler on your system called **g++**. Open the Terminal application, and type

g++

If g++ is installed, you should get an error message like no input files. If g++ is not installed and you are using a relatively newer version of Mac OS, a dialogue box will pop up asking you if you would like to install the command line tools. Click "Install".

However, on older Mac OS systems, you'll need to go to the Apple developer's website and download the command line developer tools. Go to this [link](https://developer.apple.com/download/). You will need to create an Apple ID if you do not already have one. Enter your Apple ID and password.

On the downloads page, there is a search box in the top left corner. Search for "command line developer tools". Then download and install the developer tools for your OS version.

Now go back to the Terminal and type

g++

You should now get an error message no input files.

To compile a program you would type:

g++ filename.cpp

or with multiple .cpp files:

g++ filename1.cpp filename2.cpp filename3.cpp

The compiler will create an executable file named a.out. To run your program, in terminal type:

./a.out

**For Linux Ubuntu**

The g++ compiler might already be on your system. If you are using Ubuntu, open Terminal and type:

dpkg *--list | grep compiler*

You will see a list of compilers. Check if g++ is in the list. If not, you can install g++ by typing:

sudo apt-get **install** **g**++

The commands to compile and run a program are the same as for Mac:

g++ filename.cpp

./a.out

**Using an IDE**

Another option is to download an IDE (Integrated Development Environment) to help write, organize, debug, compile and execute your code.

Microsoft provides an IDE called Visual Studio, which you can download here: [Visual Studio](https://www.visualstudio.com/). Up until recently, Visual Studio was only available for Windows. But there is also a Mac version as well.

For Mac users, Apple also provides an IDE called Xcode, which you can download [here](https://developer.apple.com/xcode/).

You can use either of these IDEs to help you develop your C++ programs.

For Linux Ubuntu users, there are a number of free IDEs available such as [NetBeans](https://netbeans.org/features/cpp/index.html), [Code::Blocks](http://www.codeblocks.org/), [Eclipse](https://www.eclipse.org/cdt/), and [CodeLite](https://codelite.org/).

NEXT

# Versions of C++

The [International Organization for Standardization](https://www.iso.org/home.html) publishes standards for the C++ language. Every few years the standard is updated with new features and syntax. The C++ standards are written documents providing guidelines for what the C++ language should be able to do and what the language looks like.

The responsibility for implementing a standard is left to the company, person or team that designs a compiler; hence, not all compilers implement all features. And some compilers might implement a feature differently.

There are currently five published C++ standards with the earliest standard called **C++98** and the most recent standard **C++17**. The number designates the year in which the standard was published such as 1998 and 2017.

Thus far, the classroom has been compiling your code with the oldest standard: **C++98**. We have used C++98 so that you could get the basics of C++ down without worrying about advanced features.

As an example of a more advanced features, **C++11**, allows you to initialize a vector with the following syntax:

vector<**int**> myvector = {5, 4, 9, 1, 10}

There is generally a lag of a few years between publication of a standard and industry adoption.

# GCC Compiler

The classroom uses a compiler called gcc. Thus far, we have been compiling your code for you when you hit the "Test Run" button. In the "Performance Programming C++" module, you will use a different interface for writing, compiling and running your code. This interface also uses the gcc compiler.

In the classroom, gcc compiles with C++98 by default. As you will see, you can use the command line to tell gcc what version of C++ to use:

g++ -std=c++11 main.cpp

You might find different behavior on your system because of the compiler you are using or because of your compiler options.

**Structuring your Functions**

In the last lesson, you learned two different ways for declaring and defining functions:

* declaring functions above main() and defining functions below main
* declaring and defining functions simultaneously above main

As your C++ programs get longer and more complex, you might want to separate your code into multiple files. Keeping function definitions and declarations separate will help you with splitting and organizing your code.

For example, take a look at this code from the functions quiz in the previous lesson.

**#include <iostream>**

**float** **distance**(**float** velocity, **float** acceleration, **float** time\_elapsed);

**int** **main**() {

std::cout << distance(3, 4, 5) << std::endl;

std::cout << distance(7.0, 2.1, 5.4) << std::endl;

**return** 0;

}

**float** **distance**(**float** velocity, **float** acceleration, **float** time\_elapsed) {

**return** velocity\*time\_elapsed + 0.5\*acceleration\*time\_elapsed\*time\_elapsed;

}

Instead of putting everything into a main.cpp like in the previous quiz, the code could be organized into two files. Check out the code below and hit the "Test Run" button.

Behind the scenes, both main.cpp and distance.cpp are being compiled. This is happening on the backend of the classroom, so you are not seeing the compilation happen.

But if you were to run this program locally, you could open a terminal and navigate to the folder containing both files. Typing something like

g++ main.cpp distance.cpp

./a.out

would compile both files together and then execute the program.

Because you have defined your function in a separate file outside of main.cpp, you can more easily re-use the function in other parts of your code.

Notice that you still had to declare the distance function at the top of main.cpp to be able to use the function.

### Header Files

The function declaration

**float** **distance**(**float** velocity, **float** acceleration, **float** time\_elapsed);

is oftentimes put into its own file as well. The declaration is kept in what's called a **header** file because the header is the information above the main() function. Header files generally have either a .h or .hpp extension. Here is the same code above but with the function declaration in a header file. If you click the "Test Run" button, you'll see that this code works as well:

The code line

**#include "distance.h"**

will paste the contents of distance.h into main.cpp.

Now let's say you wanted to reuse your distance function in a different file. All you have to do is declare the function at the top of your new file with an include statement:

**#include "distance.h"**

and you can use the function in another part of your program.

Organizing your code into different .cpp files separates out the implementation from the declaration. Furthermore, using header files means that you do not have to remember what the function declaration looked like because you can include it with the simple syntax of

**#include "distance.h"**

To compile the code, you only need to compile the .cpp files but not the .h file:

g++ main.cpp distance.cpp

### File Naming

Naming conventions dictate that the header file and associated cpp file have the same name. Therefore it's clear that distance.h contains the header declarations for distance.cpp. You'll also notice that the function name itself was called distance(). So the function name, .cpp and .h files all match.

These are naming conventions, so your C++ code will still compile if you do not follow these conventions; however, it's highly recommended to stick with these conventions.

### Include syntax

You might be wondering why there are two different types of include statements:

**#include <iostream>**

**#include "distance.h"**

The include statement with quotes tells the program to look for the distance.h file in the current directory.

The <> syntax will depend on your C++ environment. Generally, environments are set up to look for the file where the C++ libraries are stored like the Standard Library.

# Input and Output

There is one last topic to discuss before moving onto the C++ Object Oriented Programming Lesson.

You have already seen how to call a function and then output the results to the terminal using cout. As an example:

std::cout << distance(3, 4, 5);

But how do you get user input from the terminal? Or how do you input data from a file into your program or write out your results to a file?

## cin

Much like the Standard Library provides a function for outputting to the terminal, the library also provides a function for reading in data from the terminal.

This code demonstrates how to use cin:

**#include <iostream>**

**#include <vector>**

**using** **namespace** std;

**int** **main**() {

**int** integerone;

**int** integertwo;

*// declare array and assign values*

cout << "Enter an integer between 1 and 100" << endl;

cin >> integerone;

cout << "Enter another integer between 1 and 100" << endl;

cin >> integertwo;

*// output the difference*

cout << "The difference between your two numbers is: ";

cout << integerone - integertwo << endl;

**return** 0;

}

To see how this code works, you will need to put the code into a .cpp file and run the program locally. The classroom playground does not allow for user input.

Next, you will learn how to input data from an external file.

# Reading in Text Files

The Standard Library includes functionality for reading text files line by line. You can then parse each line of the text file one line at a time.

Say, for example, you have a text file with numbers and commas representing a 3 by 4 matrix:

1, 6, 2, 10.5

11, 15.2, 2, 21

3, 9, 1, 7.5

You want to read in this file and create a 2D vector to represent the matrix. Here is code to do this, and then below you can run the code to see that it works.

**#include <iostream>**

**#include <fstream>**

**#include <string>**

**#include <sstream>**

**#include <vector>**

**using** **namespace** std;

**int** **main**() {

*// initialize string variables for reading in text file lines*

string line;

stringstream ss;

*// initialize variables to hold the matrix*

vector < vector <**float**> > matrix;

vector<**float**> row;

*// counter for characters in a text file line*

**float** i;

*// read in the file*

ifstream **matrixfile** ("matrix.txt");

*// read in the matrix file line by line*

*// parse the file*

**if** (matrixfile.is\_open()) {

**while** (getline (matrixfile, line)) {

*// parse the text line with a stringstream*

*// clear the string stream to hold the next line*

ss.clear();

ss.str("");

ss.str(line);

row.clear();

*// parse each line and push to the end of the row vector*

*// the ss variable holds a line of text*

*// ss >> i puts the next character into the i variable.*

*// the >> syntax is like cin >> some\_value or cout << some\_value*

*// ss >> i is false when the end of the line is reached*

**while**(ss >> i) {

row.push\_back(i);

**if** (ss.peek() == ',' || ss.peek() == ' ') {

ss.ignore();

}

}

*// push the row to the end of the matrix*

matrix.push\_back(row);

}

matrixfile.close();

*// print out the matrix*

**for** (**int** row = 0; row < matrix.size(); row++) {

**for** (**int** column = 0; column < matrix[row].size(); column++) {

cout << matrix[row][column] << " " ;

}

cout << endl;

}

}

**else** cout << "Unable to open file";

**return** 0;

}

Here is a demonstration of how to read in data from a text file. The code reads in a tile file that contains data for a matrix. Then the code outputs the matrix to the terminal display. Read through the code to try to figure out what it does. Then run the code by pressing the "Test Run" button.

Below the demo, there is an explanation of what the code is doing.

There are two parts of the code that you haven't seen yet: fstream and sstream. Both of these files are part of the C++ Standard Library.

fstream provides functions and classes for reading in and outputting files.

This line of code reads in the file "matrix.txt" and then creates an object called "matrixfile" that you can use for reading in the text file:

ifstream **matrixfile** ("matrix.txt");

The if statement that follows checks that the file opened correctly:

if (matrixfile.is\_open()) {

and then a while loop reads the file one line at a time. Each line is placed into a variable called "line":

if (matrixfile.is\_open()) {

while (getline (matrixfile, line)) {

If you look at the text file, each line in this case is a string consisting of floats, commas and spaces. For example, "1, 6, 2, 10.5".

The sstream file in the Standard Library provides functionality for manipulating and parsing the string. In the code, you'll see that first a sstream object was declared and then later the ss object was used to cycle through and parse each line of the text file:

stringstream ss;

....

ss.clear();

ss.str("");

ss.str(line);

**while**(ss >> i) {

row.push\_back(i);

**if** (ss.peek() == ',' || ss.peek() == ' ') {

ss.ignore();

}

}

In other words, the code finds a float number and appends the number to the vector called row. The line ss.peek() looks at the next character to see if it is a comma or a space and ignores commas or spaces.

Note as well that when you are done with reading in the file, it's a good habit to close the file.

matrixfile.close();

If you ever write a programming that opens many files simultaneously and never closes them, your program could crash.

# Outputting to Text Files

Much like how you can input data from a file, you can also output data to a file. Say you have a matrix and you want to save the results to a text file. You'll see that the code for outputting the matrix to a file looks quite similar to the code for outputting the matrix to the terminal.

You will need to run this code locally in order to see the outputted text file.

**#include <iostream>**

**#include <fstream>**

**#include <vector>**

**using** **namespace** std;

**int** **main**() {

*// create the vector that will be outputted*

vector < vector <**int**> > matrix (5, vector <**int**> (3, 2));

vector<**int**> row;

*// open a file for outputting the matrix*

ofstream outputfile;

outputfile.open ("matrixoutput.txt");

*// output the matrix to the file*

**if** (outputfile.is\_open()) {

**for** (**int** row = 0; row < matrix.size(); row++) {

**for** (**int** column = 0; column < matrix[row].size(); column++) {

**if** (column != matrix[row].size() - 1) {

outputfile << matrix[row][column] << ", ";

}

**else** {

outputfile << matrix[row][column];

}

}

outputfile << endl;

}

}

outputfile.close();

**return** 0;

}

You can see that you need to create an ofstream object and then use the object to create a new file.

ofstream outputfile;

outputfile.open ("matrixoutput.txt");

The rest of the code iterates through the matrix and outputs the matrix in the format you specify in the code:

**if** (outputfile.is\_open()) {

**for** (**int** row = 0; row < matrix.size(); row++) {

**for** (**int** column = 0; column < matrix[row].size(); column++) {

**if** (column != matrix[row].size() - 1) {

outputfile << matrix[row][column] << ", ";

}

**else** {

outputfile << matrix[row][column];

}

}

outputfile << endl;

}

}

The if statement is checking whether or not the end of the row is reached. If the current value is the end of a row, it's not necessary to put a comma separator after the number:

**if** (column != matrix[row].size() - 1) {

outputfile << matrix[row][column] << ", ";

}

**else** {

outputfile << matrix[row][column];

}