**Vectors and Matrix Algebra**

Great work! You made it through a bunch of C++ syntax. As you've probably realized, programming in C++ is arguably harder than programming in Python. C++ was designed for fast execution, and the language gives you a lot of different ways to get the same results. Python was designed for writing code quickly but at the expense of execution speed.

There is one last piece of syntax you'll need to translate your Python code from earlier in the nanodegree: C++ vectors, which are like Python lists.

**The Vector Library**

When you were writing Python programs to store and manipulate matrices, you used Python lists. C++ vectors are just like Python lists. In this lesson, you are going to practice using C++ vectors in preparation for translating Python code to C++.

But hold on! C++ also has something called a list. But this is where things get confusing. However, C++ lists do not work the same way as Python lists.

C++ lists and C++ vectors are both in a family of structures called [sequence containers](http://en.cppreference.com/w/cpp/container). These containers allow you to store values in series and then access those values. C++ has a handful of sequence containers including lists, vectors, and arrays.

Don't get confused! C++ vectors are the closest to Python lists. You can add elements to a C++ vector just like you can in a Python list. You can remove elements as well and also easily access any element in the vector.

**Declaring C++ Vectors**

Declaring C++ vector variables is like declaring any other type of variable:

typedefinition variablename;

But the vector type definition has a funny looking syntax because you also need to declare what kind of values will go inside the vector such as integer, char, float, string, etc. Here are some examples of variable declarations using vectors:

std::vector<**char**> charactervectorvariable;

std::vector<**int**> integervectorvariable;

std::vector<**float**> floatvectorvariable;

std::vector<**double**> doublevectorvariable:

**Including the Vector Library**

In an actual program, you would need to include the vector file from the Standard Library:

**#include <vector>**

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

std::vector<**float**> floatvectorvariable;

**return** 0;

}

The above code will declare an empty vector of type float.

More generically, you declare a vector with:

std::vector<datatype> variablename;

# Namespaces

C++ vector syntax is a little bit hard to read especially because you have to type std over and over again: like for example, std::cout or std::string or std::vector.

Thankfully, C++ provides a way to avoid writing std all the time.

Std is something called a [namespace](http://en.cppreference.com/w/cpp/language/namespace). Without getting too much into the details, namespaces let you organize code into logical groups. In this case, std is the namespace for the Standard Library.

You can actually declare your namespace at the top of your main.cpp file and then avoid writing

std::

over and over again. Here is an example:

**#include <iostream>**

**#include <vector>**

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

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

vector<**int**> intvectorvariable;

**int** intvariable = 5;

cout << intvariable << endl;

**return** 0;

}

Now, the vector declaration, cout and endl no longer needed std::.

### Benefits of Namespaces

Declaring the namespace makes the code easier to read and write. The downside is that you have to be careful with how you name your own variables and functions. Previously, you might have written:

std::cout

which lets your program know that you meant the cout function from the standard library.

C++ would have let you actually create a variable or function named cout as well. That's probably not a good idea, but the code won't produce an error. Once you declared the std namespace, your cout variable or function would be in conflict with the standard library cout.

Going forward from this point, the exercises and code examples will include the using namespace std; line of code.

You can now simplify the vector syntax using namespaces. Let's compare Python list and C++ vector syntax and then practice coding C++ vectors.

## Namespace Practice

Use the standard library namespace and change the code so that the code no longer uses "std::".

A screenshot of a cell phone

Description automatically generated

You can now simplify the vector syntax using namespaces. Let's compare Python list and C++ vector syntax and then practice coding C++ vectors.

**Python and C++ Comparison**

Let's get back to vectors! You have already seen how to declare an empty vector.

In the code below, you can compare Python lists and C++ vector syntax. You'll see that the C++ vector is using a method called push\_back, which appends values to the end of a vector. And the line of code

vector<**float**> myvector (5);

declares vector of size five but without assigning any values. Assigning values to a C++ vector can be a bit tricky; later in this lesson you will see a few different ways to assign values to a vector variable.

A screenshot of a cell phone

Description automatically generated

The Python code is, as you've seen previously, much shorter to write than the C++ code. However, there are other ways for inputting values in a vector, which you will see in the next section.

# Initializing Vector Values

In the previous part of the lesson, you learned to declare a vector first and then assign values:

vector<**float**> myvector(5);

myvector[0] = 5.0;

myvector[1] = 3.0;

myvector[2] = 2.7;

myvector[3] = 8.2;

myvector[4] = 7.9;

There are various other ways for assigning initial values to a vector. Here are two other ways:

## Declaring and Defining Simultaneously

When declaring a vector, you can also assign initial values simultaneously.

std::vector<**int**> myvector (10, 6);

The code will declare a vector with ten elements, and each element will have the value 6.

## Declaring and Defining Simultaneously with Brackets

There is another way to initialize a vector as well if you are using one of the more recent versions of C++ such as C++11 or C++17; You could also do something like:

std::vector<**float**> myvector = {5.0, 3.0, 2.7, 8.2, 7.9}

The different versions of C++ (C++98, C++11, C++14, and C++17) will be discussed in the Practical C++ lesson.

# Vector Methods

Vectors have a handful of useful functions, which you can see [here](http://www.cplusplus.com/reference/vector/vector/). In this part of the lesson, you will go over the ones you will be using in the object oriented programming lesson.

## assign

Assign helps you quickly populate a vector with fixed values. For example this code,

vector<**int**> intvariable;

intvariable.assign(10,16);

is going to populate the vector with ten integers all having the value of 16.

The assign method lets you override your current vector with a new vector.

Remember, you've already seen a similar way to initialize values in a vector:

vector<**int**> intvariable(10,16);

The difference is that the assign method lets you override your current vector with new values.

## push\_back

Pushback adds an element to the end of the vector:

vector<**int**> intvariable;

intvariable.push\_back(25);

## size

Size returns the size of the vector.

intvariable.size();

# Practice with Vector Methods

Before getting practice with these vector methods, move on to the next section to learn about accessing vectors with for loops. In the next section, you'll combine the methods you just learned with for loops.

# Vectors and For Loops

Much of the time, you will be using for loops to manipulate vectors. Once you are comfortable using for loops with vectors, you can do things like:

* populate a vector with values
* do math with vectors

Here is a program that initializes a vector and then uses a for loop to populate the vector with values. Then another for loop reads out the vector values.

**#include <iostream>**

**#include <vector>**

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

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

vector<**float**> example;

**for** (**int** i = 0; i < 5; i++) {

example.push\_back(i\*5.231);

}

**for** (**int** i = 0; i < example.size(); i++) {

cout << example[i] << endl;

}

**return** 0;

}

The output looks like this:

0

5.231

10.462

15.693

20.924

## Using i++ Versus ++i

So far, you've learned to write C++ for loops like the following:

**for** (**int** i = 0; i < 10; i++) {}

This syntax matches closely to the Python for loop syntax; however, you can also write a for loop like this:

**for** (**int** i = 0; i < 10; ++i) {}

**What is the difference and why do both ways work?**

In practice, both i++ and ++i will give you the same results; these are a shorthand way of writing i = i + 1. The difference between the two is subtle.

**int** i = 5;

**int** x = i++; *// x = 5, i = 6 (called postfix)*

**int** x = ++i; *// x = 6, i = 6 (called prefix)*

In both cases, the i variable increases by 1. In the postfix case, i++, int x = i is evaluated first and then i = i + 1 occurs.

In the prefix case, ++i, i = i + 1 occurs first and then int x = i executes.

Many code guidelines recommend using ++i over i++. In reality neither one is more efficient than the other when using integer variables.

However, there is a difference when you write a C++ class that overloads the ++ operator. You saw operational overloading in the Python matrix project; the code overloaded mathematical signs to carry out matrix addition, subtraction, multiplication, etc.

When overloading the postfix operator, C++ needs to keep track of two values. In the example, the values would be 5 and 6. For the prefix operator, C++ only needs to keep track of one value: 6. Hence, when overloading the ++ operator, it's generally more efficient to use prefix than the postfix.

Overloading is an advanced C++ topic that isn't covered in depth here. If you'd like to learn more, here are a few resources:

[Stackoverflow](https://stackoverflow.com/questions/3846296/how-to-overload-the-operator-in-two-different-ways-for-postfix-a-and-prefix)

[IBM Knowledge Center](https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.2.0/com.ibm.zos.v2r2.cbclx01/cplr330.htm)

# Math and Vectors

Any vector math you did in Python, you can also do in C++ with for loops.

## Example 1

For example, you might want to multiply every element in a vector by a constant:

**#include <iostream>**

**#include <vector>**

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

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

vector<**float**> example;

*// assign 5 floats with value 10*

example.assign(5,10.0);

*// print out the vector*

**for** (**int** i = 0; i < example.size(); i++) {

cout << example[i] << endl;

}

*// blank line outputted to terminal*

cout << endl;

*//multiply each value in the vector by 20*

**for** (**int** i = 0; i < example.size(); i++) {

example[i] = 20 \* example[i];

}

*// print out the vector*

**for** (**int** i = 0; i < example.size(); i++) {

cout << example[i] << endl;

}

**return** 0;

}

Which gives the output:

10

10

10

10

10

200

200

200

200

200

## Example 2

Or you might want to add two vectors together:

**#include <iostream>**

**#include <vector>**

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

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

vector<**int**> exampleone (5);

vector<**int**> exampletwo (5);

vector<**int**> examplesum (5);

exampleone[0] = 2;

exampleone[1] = 6;

exampleone[2] = 25;

exampleone[3] = 1;

exampleone[4] = 18;

exampletwo[0] = 3;

exampletwo[1] = 19;

exampletwo[2] = 8;

exampletwo[3] = 12;

exampletwo[4] = 191;

cout << "vector one ";

*// print out the first vector*

**for** (**int** i = 0; i < exampleone.size(); i++) {

cout << exampleone[i] << " ";

}

*// create a new line in the terminal*

cout << endl;

cout << "vector two ";

*// print out the second vector*

**for** (**int** i = 0; i < exampletwo.size(); i++) {

cout << exampletwo[i] << " ";

}

*// create a new line in the terminal*

cout << endl;

cout << "vector sum ";

*//add the vectors together*

**for** (**int** i = 0; i < exampleone.size(); i++) {

examplesum[i] = exampleone[i] + exampletwo[i];

}

*// print out the vector*

**for** (**int** i = 0; i < examplesum.size(); i++) {

cout << examplesum[i] << " ";

}

*// create a new line in the terminal*

cout << endl;

**return** 0;

}

Which gives output:

vector one 2 6 25 1 18

vector two 3 19 8 12 191

vector sum 5 25 33 13 209

Now it's your turn to write programs with C++ vectors. Move on to the next part of the lesson to get some practice coding vectors.

# Playground

Now it's your turn to make some programs with vectors. Here is a playground where you can write your program.

#### Idea 1

First, try writing a program that initializes a vector of size 3. The values for this vector are [5, 10, 27][5,10,27]. Initialize another vector of size 3 with the values [3, 17, 12][3,17,12]. Now subtract the two vectors from each other and output the results.

To get even more practice, write a function that takes in two vectors and then outputs the difference between the vectors. Assume that the two vectors are the same size; otherwise you would have to check that they are the same size and include some error checking.

#### Idea 2

Initialize a vector with the values [17, 10, 31, 5, 7][17,10,31,5,7]. Initialize another vector with the values [3, 1, 6, 19, 8][3,1,6,19,8]. Then, output another vector that contains the product of each element. In other words, the vector should have [17\times3, \space10\times1, \space 31\times6, \space 5\times19, \space 7\times8][17×3, 10×1, 31×6, 5×19, 7×8].

To get even more practice, write a function that takes in two vectors and then outputs a new vector that is the result of element by element multiplication. Assume that the two vectors are the same size; otherwise you would have to check that they are the same size and do some error checking.

# Two Dimensional Vectors

Next, you are going to use vectors to store matrices. Much like how Python uses a list of lists to store matrices, for the C++ lessons you will use a vector of vectors. The syntax for declaring two-dimensional vectors is a bit tricky.

Say you're using Python and want to store a 3 by 5 matrix. You could do something like this:

matrixexample = [[2, 1, 5], [7, 9, 2], [16, 5, 9], [5, 2, 1], [1, 2, 4]]

In C++, you are going to create a similar structure by appending vectors to vectors. Here is a comparison of Python and C++ code to see what this looks like:

A screenshot of a social media post

Description automatically generated

## Explanation of the Code

First, the line

vector < vector <**int**> > twodvector;

declares an empty two dimensional vector named twodvector. A couple of things to notice:

* like with all variables in C++, you have to declare what type of values will go into the vector. In this case, integers will go into the twodvector variable.
* Notice the spaces between brackets < vector <int> >. Your program probably won't run if the spacing is not done correctly.

Then a one dimensional vector called singlerow is declared. The singlerow vector has the form [2, 2, 2].

vector<**int**> singlerow (3,2);

Then the singlerow vector is appended to the twodvector five times:

**for** (**int** i = 0; i < 5; i++) {

twodvector.push\_back(singlerow);

}

You end up with the same two-dimensional structure just like in the Python code. If you were to run this C++ code, the terminal would print out:

2 2 2

2 2 2

2 2 2

2 2 2

2 2 2

## Alternative Initialization

Here is another way you could have set up the vector from the previous example:

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

The syntax is a little bit more complicated. But if you start from the inside of the parenthesis and work your way out, you see that you have already seen all of the functionality.

The line:

vector <**int**> (3, 2)

would set up an integer vector like {2, 2, 2}. So even though you don't see the inner vector, the code is essentially doing something like this:

vector < vector <**int**> > twodvector (5, {2, 2, 2});

So then the code copies {2, 2, 2} five times into the twodvector variable:

{{2, 2, 2},

{2, 2, 2},

{2, 2, 2},

{2, 2, 2},

{2, 2, 2}}

Just keep in mind that only Python represents vectors or matrices with square brackets []. Newer versions of C++ can use squiggly brackets to represent vectors {}, but older implementations of C++ do not have an equivalent representation.

A line of code like the following would not run in C++:

vector < vector <**int**> > twodvector (5, [2, 2, 2]);

# 2D Vectors and For Loops

Because 2D vectors are just vectors inside a vector, a 2D vector has the [same methods](http://www.cplusplus.com/reference/vector/vector/) as a 1D vector.

That way the cout code from the example works:

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

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

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

}

cout << endl;

When you type twodvector.size(), that will give you the size of the outside vector. The outside vector had five elements, which represents the number of rows in the matrix being represented:

{2 2 2}

{2 2 2}

{2 2 2}

{2 2 2}

{2 2 2}

When you write twodvector[0].size(), you are taking the first element of the outside vector, [2 2 2], and asking for the size of that vector, which in this case is three. So essentially the for loop is saying:

**for** (**int** row = 0; row < 5; row++) {

**for** (**int** column = 0; column < 3; column++) {

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

}

cout << endl;

NEXT