**Lesson Overview**

In this lesson, you are going to learn how to write classes in C++. The lesson assumes a basic knowledge of object oriented programming as taught in the "Working with Matrices - State and Object Oriented Programming" lessons from earlier in the nanodegree program.

The first part of the lesson is organized with a top down approach. We are going to show you a Python class and then compare the same class in C++. Once you get a feel for C++ object oriented syntax, you'll then code your own C++ class step by step.

Keep in mind that the fundamentals of object oriented programming remain the same whether you are using Python or C++. In both Python and C++, you define class variables and class methods; the difference is mostly in the syntax. And there are a few aspects of C++ that Python does not have as you'll find out in this lesson.

Continue on to get started!

# How the Lesson is Organized

This lesson has two parts. In the first part, we show you a complete Python class and its equivalent in C++. You'll get an overview of what C++ classes look like and how to use them.

Then, in the second part, you'll code your own C++ class building up the code piece by piece. By the end of the lesson, you should feel comfortable reading C++ object oriented code and writing a basic class; however, remember that learning a new programming language takes practice. And C++ is often considered a language that takes [years to master](https://www.quora.com/How-long-does-it-take-to-learn-C++-coding-on-an-average-level).

# Example: Python Class

Let's go directly to an example. Below is the code for a Python class called 'Gaussian'.

You learned about Gaussian distributions and saw the Gaussian equation earlier in the nanodegree. This class stores the values for the standard deviation and mean. The class also has methods for calculating the probability density function, the sum of two gaussians, and the product of two gaussians.

The class contains two class variables called mu, which is the average and sigma2, which is the variance.

Here is a summary of the three methods contained in the class:

1. evaluate, which represents the probability density function: \LARGE\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2}\frac{(x - \mu)^2}{\sigma^2}}2*πσ*2​1​*e*−21​*σ*2(*x*−*μ*)2​

In this formula \sigma^2*σ*2 is the variance.

1. multiply, which multiplies two independent Gaussian distributions together
2. add, which adds two independent Gaussian distributions together

Read through the code so that you understand how the class works.

### Python Code for a Gaussian Class

**class** **Gaussian**():

**def** **\_\_init\_\_**(self, mean, variance):

self.mu = mean

self.sigma2 = variance

**def** **evaluate**(self, x):

coefficient = 1.0 / sqrt(2.0 \* pi \* self.sigma2)

exponential = exp(-0.5 \* (x-self.mu) \*\* 2 / self.sigma2)

**return** coefficient \* exponential

**def** **multiply**(self, other):

*# calculate new mean*

denominator = self.sigma2 + other.sigma2

numerator = self.mu \* other.sigma2 + other.mu \* self.sigma2

new\_mu = numerator / denominator

*# calculate new variance*

new\_var = 1.0 / ( (1.0 / self.sigma2) + (1.0 / other.sigma2) )

*# generate new gaussian*

**return** Gaussian(new\_mu, new\_var)

**def** **add**(self, other):

new\_mu = self.mu + other.mu

new\_sigma2 = self.sigma2 + other.sigma2

**return** Gaussian(new\_mu, new\_sigma2)

# Example: C++ Class

Now, you will take a look at an equivalent class in C++. Like in other cases you have already seen, the C++ code is longer and has aspects that the Python version did not have. It takes more time and practice to learn object oriented programming in C++ compared to Python.

For example, you will notice that in the C++ class, all of its variables and all of its functions need to be declared first before writing the implementation. The class also has a part labeled private and another part labeled public. Furthermore, the C++ class includes extra functions like setMu, setSigma2, getMu, and getSigma2.

You are going to learn about all of these differences in this lesson. For now, read through the code and see if you can figure out what the set functions and get functions do. Then answer the quiz at the bottom of the page.

**#include <math.h>**

**class** Gaussian

{

*// private variable declaration*

**private**:

**float** mu, sigma2;

*// public variable and function declarations*

**public**:

*// constructor functions*

Gaussian ();

Gaussian (**float**, **float**);

*// change value of average and standard deviation*

**void** **setMu**(**float**);

**void** **setSigma2**(**float**);

*// output value of average and standard deviation*

**float** **getMu**();

**float** **getSigma2**();

*// functions to evaluate*

**float** **evaluate** (**float**);

Gaussian **multiply** (Gaussian);

Gaussian **add** (Gaussian);

};

*// constructor function definitions*

Gaussian::Gaussian() {

mu = 0;

sigma2 = 1;

}

Gaussian::Gaussian (**float** average, **float** sigma) {

mu = average;

sigma2 = sigma;

}

*// set function definitions*

**void** Gaussian::setMu (**float** average) {

mu = average;

}

**void** Gaussian::setSigma2 (**float** sigma) {

sigma2 = sigma;

}

*// get function definitions*

**float** Gaussian::getMu () {

**return** mu;

}

**float** Gaussian::getSigma2() {

**return** sigma2;

}

*// evaluate function definition*

**float** Gaussian::evaluate(**float** x) {

**float** coefficient;

**float** exponential;

coefficient = 1.0 / sqrt (2.0 \* M\_PI \* sigma2);

exponential = exp ( pow (-0.5 \* (x - mu), 2) / sigma2 );

**return** coefficient \* exponential;

}

*// multiply function definition*

Gaussian Gaussian::multiply(Gaussian other) {

**float** denominator;

**float** numerator;

**float** new\_mu;

**float** new\_var;

denominator = sigma2 + other.getSigma2();

numerator = mu \* other.getSigma2() + other.getMu() \* sigma2;

new\_mu = numerator / denominator;

new\_var = 1.0 / ( (1.0 / sigma2) + (1.0 / other.sigma2) );

**return** Gaussian(new\_mu, new\_var);

}

*// add function definition*

Gaussian Gaussian::add(Gaussian other) {

**float** new\_mu;

**float** new\_sigma2;

new\_mu = mu + other.getMu();

new\_sigma2 = sigma2 + other.getSigma2();

**return** Gaussian(new\_mu, new\_sigma2);

}

A screenshot of a cell phone

Description automatically generated

# Using a Class in a Program

Let's start with the big picture. We've written code below with a fully implemented class and a main.cpp file that uses the class. Even though you might not be familiar with some of the C++ syntax, your knowledge of Python object oriented programming should help you understand what the C++ code is doing.

Study the code and then run the code as instructed below.

# Main.cpp and Gaussian.cpp

At the bottom of the page, you'll find a main.cpp file and a gaussian.cpp file so that you can see how they work together.

The gaussian.cpp file contains the class definition including all the variables and functions that the Gaussian class needs. You would make a similar file in Python probably called gaussian.py.

The main.cpp file uses the class to run some calculations. You'll see one important difference between C++ and Python. In C++, you need to declare your class before you can use the class. Both main.cpp and gaussian.cpp have the same class declaration at the top of their files:

**class** Gaussian

{

**private**:

**float** mu, sigma2;

**public**:

*// constructor functions*

Gaussian ();

Gaussian (**float**, **float**);

*// change value of average and standard deviation*

**void** **setMu**(**float**);

**void** **setSigma2**(**float**);

*// output value of average and standard deviation*

**float** **getMu**();

**float** **getSigma2**();

*// functions to evaluate*

**float** **evaluate** (**float**);

Gaussian **mul** (Gaussian);

Gaussian **add** (Gaussian);

};

Below are all of the files that would be used in this code so that you can see the relationship between the main.cpp and gaussian.cpp

You can't see it on the backend, but this program is first being compiled via the command:

g++ main.cpp gaussian.cpp

# Explanation of the Main.cpp File

In the previous section, there were two files. The gaussian.cpp contained the code that defined the Gaussian class. The main.cpp used the Gaussian class.

The main.cpp file had three parts:

* a header, which is where the #include statements were
* a class declaration
* a main function.

# Header

You saw headers in the C++ getting started lessons. In the main.cpp, the header included the iostream library for outputting to the terminal:

**#include <iostream>**

# Class Declaration

Then comes the class declaration. The class declaration is very similar to function declarations, which you learned about previously. In fact, as you'll see later in the lesson, you can put the class declaration into a separate .h file just like you did with function declarations.

The purpose of the class declaration is to give the main function access to the Gaussian class.

*// class declaration*

**class** Gaussian

{

**private**:

**float** mu, sigma2;

**public**:

*// constructor functions*

Gaussian ();

Gaussian (**float**, **float**);

*// change value of average and standard deviation*

**void** **setMu**(**float**);

**void** **setSigma2**(**float**);

*// output value of average and standard deviation*

**float** **getMu**();

**float** **getSigma2**();

*// functions to evaluate*

**float** **evaluate** (**float**);

Gaussian **mul** (Gaussian);

Gaussian **add** (Gaussian);

};

Notice that a class declaration looks a lot like the variable declarations and function declarations you've already been using. Declarations tell the program what the variable types will be. The declarations also show the input and output types for functions.

# Main Function

And finally comes the main function. The main function instantiates objects of the Gaussian class. So the main function uses the class whereas gaussian.cpp defined the class. You could take the code in gaussian.cpp and put it at the bottom of main.cpp; however, your code files will become quite large and hard to read through.

Here is the code from the main function:

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

{

Gaussian **mygaussian**(30.0,20.0);

Gaussian **othergaussian**(10.0,30.0);

std::cout << "average " << mygaussian.getMu() << std::endl;

std::cout << "evaluation " << mygaussian.evaluate(15.0) << std::endl;

std::cout << "mul results sigma " << mygaussian.mul(othergaussian).getSigma2() << std::endl;

std::cout << "mul results average " << mygaussian.mul(othergaussian).getMu() << std::endl;

std::cout << "add results sigma " << mygaussian.add(othergaussian).getSigma2() << std::endl;

std::cout << "add results average " << mygaussian.add(othergaussian).getMu() << std::endl;

**return** 0;

}

NEXT

# Anatomy of a Class

If you look back at the gaussian.cpp class file, you'll notice that there were four distinct sections. The file contained:

* a header
* class declarations
* constructor functions
* and method definitions

It might help to think about these four sections as you write your own code. As a review, here is the code from each section of the gaussian.cpp file.

# Header

The **header** has all of the include statements.

**#include <math.h>**

# Class Declaration

The **class declaration** is a lot like a variable or function declaration. In the class declaration, you let the compiler know what all of the class variables and methods look like in terms of data types, inputs and outputs.

**class** Gaussian

{

**private**:

**float** mu, sigma2;

**public**:

*// constructor functions*

Gaussian ();

Gaussian (**float**, **float**);

*// change value of average and standard deviation*

**void** **setMu**(**float**);

**void** **setSigma2**(**float**);

*// output value of average and standard deviation*

**float** **getMu**();

**float** **getSigma2**();

*// functions to evaluate*

**float** **evaluate** (**float**);

Gaussian **multiply** (Gaussian);

Gaussian **add** (Gaussian);

};

We'll talk about the difference between public and private later in the lesson. In essence, a private function or variable is only reachable within the class code whereas a public function or definition is accessible to objects as well.

# Constructor Functions

**Constructor** functions determine how a new object will be initiated. When you declare a new object, should the object have default values? Or will you provide values every time you instantiate an object?

Python had an equivalent syntax with the \_\_init\_\_.

**def** **\_\_init\_\_**(self, variable1, variable2, ..., variablen):

The first constructor function is for when you instantiate an object without specifying the average and variance.:

Gaussian::Gaussian() {

mu = 0;

sigma2 = 1;

}

The other constructor function specifies what to do when you do specify the average and variance:

Gaussian::Gaussian (**float** average, **float** sigma) {

mu = average;

sigma2 = sigma;

}

# Class Methods

And finally, the class methods define all of the functions that your class needs to have.

The rest of the code contains the definitions for all of the functions, also called methods, that your class has.

The get and set functions are specifically for getting variable values or changing the value of private variables. Again, we'll go into more detail about private and public later in the lesson.

**void** Gaussian::setMu (**float** average) {

mu = average;

}

**void** Gaussian::setSigma2 (**float** sigma) {

sigma2 = sigma;

}

**float** Gaussian::getMu () {

**return** mu;

}

**float** Gaussian::getSigma2() {

**return** sigma2;

}

The rest of the functions (evaluate, multiply, add) are the same functions that were in the Python version of the class.

**float** Gaussian::evaluate(**float** x) {

**float** coefficient;

**float** exponential;

coefficient = 1.0 / sqrt (2.0 \* M\_PI \* sigma2);

exponential = exp ( pow (-0.5 \* (x - mu), 2) / sigma2 );

**return** coefficient \* exponential;

}

Gaussian Gaussian::multiply(Gaussian other) {

**float** denominator;

**float** numerator;

**float** new\_mu;

**float** new\_var;

denominator = sigma2 + other.getSigma2();

numerator = mu \* other.getSigma2() + other.getMu() \* sigma2;

new\_mu = numerator / denominator;

new\_var = 1.0 / ( (1.0 / sigma2) + (1.0 / other.sigma2) );

**return** Gaussian(new\_mu, new\_var);

}

Gaussian Gaussian::add(Gaussian other) {

**float** new\_mu;

**float** new\_sigma2;

new\_mu = mu + other.getMu();

new\_sigma2 = sigma2 + other.getSigma2();

**return** Gaussian(new\_mu, new\_sigma2);

}

# Other Facets of C++ Classes

If you haven't taken one yet, now might be a good time to take a stretch break. We've still got three more topics to cover before you code your own class:

* Private versus Public
* Header Files
* Inclusion Guards

As you'll see in the next lesson node, **private** variables and functions are only available within your class code. **Public** functions and variables, on the other hand, are accessible within your class and also by an object of the class.

You are already familiar with **header files** from the "C++ Getting Started" lesson. While header files are not needed to run code, they are very helpful for organizing and reusing code. We'll explain how to use header files when organizing your C++ code.

C++ compilers do not like it when your code declares the same variables, functions or classes more than once. As your code gets longer and more complex, you'll oftentimes include more than one header file at the top of your code. These header files could contain the same class or function declarations, and then your code won't compile. You'll see how to avoid this situation in the "Inclusion Guards" lesson node.

Continue on to learn about these three aspects of C++ programming.

# What do Private, Protected and Public Mean?

In the Gaussian class declaration, the **mu** and **sigma2** variables were marked as private whereas the rest of the variables and functions were in a section marked public. Here is a reminder of the class declaration:

**class** **Gaussian**

{

**private**:

**float** mu, sigma2;

**public**:

*// constructor functions*

Gaussian ();

Gaussian (**float**, **float**);

*// change value of average and standard deviation*

**void** **setMu**(**float**);

**void** **setSigma2**(**float**);

*// output value of average and standard deviation*

**float** **getMu**();

**float** **getSigma2**();

*// functions to evaluate*

**float** **evaluate** (**float**);

Gaussian **multiply** (Gaussian);

Gaussian **add** (Gaussian);

};

# Private and Public

These keywords **private** and **public** determine which part of your program will have access to the variables and functions. If a variable or function is **private**, then only the class code itself has access to these variables and functions.

On the other hand, anything marked **public** can be accessed outside the class; for example, when you instantiate an object, your program will be able to use the set and get functions as well as the evaluate, multiply and add functions; however, your program will not be able to access the **mu** and **sigma2** variables directly.

# Protected

There is another keyword called **protected**, which wasn't used in the example. Basically, **protected** classes and variables can be accessed by any subclasses. For example, if you wrote a Vehicle class, you might write a Car class, a Van class, and a Truck class that would all inherit from the more general Vehicle class. Any protected variables in the Vehicle class could be accessed in the child classes.

# Example of Public vs. Private

Below is another example of the Gaussian class except mu and sigma2 have been made public. Notice how it is no longer necessary to have getMu, getSigma2, setMu and setSigma2 functions because the object has direct access to those variables.

**Why Keep Things Private**

By default, C++ makes all class variables and functions private. That means you can actually declare private variables and functions at the top of your class declaration without even labeling them **private**:

**class** Gaussian

{

**float** mu, sigma2;

**public**:

*// constructor functions*

Gaussian ();

Gaussian (**float**, **float**);

*// change value of average and standard deviation*

**void** **setMu**(**float**);

**void** **setSigma2**(**float**);

*// output value of average and standard deviation*

**float** **getMu**();

**float** **getSigma2**();

*// functions to evaluate*

**float** **evaluate** (**float**);

Gaussian **mul** (Gaussian);

Gaussian **add** (Gaussian);

};

C++ thus encourages you to make everything private unless you have a good reason not to do so. For example, by making the mu and sigma2 variables private, you have separated how **mu** and **sigma2** are implemented versus how **mu** and **sigma2** are accessed.

What happens if the way your class calculates **mu** and **sigma2** changes? If these variables had been public, then any code that uses your class might break. When **mu** and **sigma2** were public, a program could directly change the value of **mu** and **sigma** with code like:

mygaussian.mu = 25;

But when **mu** and **sigma2** were private, a program had to use code like this:

mygaussian.setMu(25)

If you needed to change something about the implementation of the **mu** variable, you would be much less likely to break existing code in the private case. A program using the Gaussian class does not need to know how **mu** was implemented as long as the program can get the **mu** value and change the value in **mu**.

# Header Files

In the previous example, you saw how to separate a class into one file (gaussian.cpp) that was separate from main.cpp. But both the main program file and the gaussian class file needed the exact same class declaration at the top of the code:

*// class declaration*

**class** Gaussian

{

**private**:

**float** mu, sigma2;

**public**:

*// constructor functions*

Gaussian ();

Gaussian (**float**, **float**);

*// change value of average and standard deviation*

**void** **setMu**(**float**);

**void** **setSigma2**(**float**);

*// output value of average and standard deviation*

**float** **getMu**();

**float** **getSigma2**();

*// functions to evaluate*

**float** **evaluate** (**float**);

Gaussian **mul** (Gaussian);

Gaussian **add** (Gaussian);

};

Instead of writing the entire declaration twice, a better option is to put the declaration into a header file. Then you can include the entire declaration with a single line of code:

**What goes in a header file?**

You were introduced to header files in the previous lesson. Header files allowed you to put function declarations in a separate file. Ultimately, using and including header files makes it easier to re-use functions in different parts of your program. Furthermore, if the class declaration changes, you only have to change the declaration in one place.

For classes, header files serve the exact same purpose. When you use the Gaussian class in main.cpp, you can simply include the header file at the top include "gaussian.h". That gives main.cpp access to the Gaussian class.

Likewise, for gaussian.cpp, you can include the header file as well rather than writing out the entire declaration.

**Using Include in Header Files**

Generally, it's recommended to put the minimum number of #include statements needed into a header file. However, because header files essentially get copied into .cpp files, you might inadvertently include the same library or file multiple times. The consequence is that variables, functions or classes might be declared multiple times as well, and the code will not compile. In the next part of the lesson, you will learn what happens when elements get declared multiple times, and you will also learn how to avoid the multiple declarations.

# Declaring variables, functions or classes multiple times

C++ programs will not compile if a variable, function or class gets declared more than once. This might seem easy to avoid when the codebase is small. But imagine what happens when you have a large codebase with many different classes, .cpp files, and personnel working on different parts of the code.

Take a look at this code below. There are two different classes in two separate files. One class represents an engine with a variable storing the size of the engine. The other class represents a car, which has a color variable and a variable representing the number of doors.

But there's a catch. The car class also uses the engine class in order to store the car's engine properties. That is why the car.h header file includes the engine header file with the line:

**#include "engine.h"**

However, the code will not compile in its current state. Think about why it does not compile correctly when you press the "Test Run" button.

As a hint, look at the include statements on the top of main.cpp. Include statements will essentially copy a file into another file. Remember that a program that defines a class more than once will not compile.

The code would not compile because of these statements:

**#include "engine.h"**

**#include "car.h"**

The first include statement will copy the contents of the engine header file into main.cpp. So main.cpp will have the definition of the Engine class.

But then, main.cpp will copy the contents of "car.h" as well. But the "car.h" file also includes engine.h:

**#include "engine.h"**

The "engine.h" file ends up being included twice, so the Engine class is declared twice. The Car uses the engine class, and main.cpp also uses the engine class.

The modularity of .cpp and .h files is a big advantage of C++. But how can you avoid the multiple declarations?

# ifndef

The solution is to use # ifndef statements, which allow you to implement a technique called inclusion guards.

The ifndef statement stands for "if not defined". When you wrap your header files with #ifndef statements, the compiler will only include a header file if the file has not yet been defined. In the current main.cpp example, the "engine.h" file would be included first. Then the compiler includes "car.h". But "car.h" will try to include "engine.h" again; however, the inclusion guard in the "engine.h" file will ensure that "engine.h" does not get included again.

Here is what the "engine.h" file looks like with an ifndef statement:

**#ifndef ENGINE\_H**

**#define ENGINE\_H**

**#include <string>**

**class** Engine

{

**private**:

std::string enginesize;

**public**:

Engine ();

Engine (std::string);

**void** **setSize**(std::string);

std::string **getSize**();

};

**#endif */\* ENGINE\_H \*/***

**#ifndef FILENAME\_H**

**#define FILENAME\_H**

header code ...

**#endif */\* FILENAME\_H \*/***

Using all caps with the \_H is a naming convention. It is also customary to put a comment after the #endif statement with the filename.

You would want to wrap all of your header files with #ifndef statements. That way other programs do not have to keep track of what files have already been included when they want to use your code.

Here are the results of including #ifndef statements in the engine and car header files. If you click on "Test Run", you will see that the code now compiles.

# Namespaces in Header Files

As an aside, you'll notice that the header files did not use the standard namespace. It's generally recommended to avoid using namespaces in a header file. This can help avoid naming conflicts later as functions and classes are reused in different parts of a code base.

# Implement a Class

Now it's time to code your own class. For the remainder of this lesson, you are going to implement a matrix class much like what you did for the Python object oriented programming lesson.

At this point, we're assuming you are familiar with basic matrix operations. So the main focus of the lesson will be practicing writing C++ code. You are going to build up the class step by step starting with declaring variables and functions, writing functions, using inclusion guards and instantiating an object.

In the next part of the lesson, you'll be given certain tasks to complete. You'll also find solutions at the bottom of each page.

**Class Variables**

Your first task will be to declare the variables in your Matrix class. As a reminder, here is the general syntax for declaring a C++ class:

**class** Classname

{

**private**:

declare **private** variables;

declare **private** functions;

**public**:

declare **public** variables;

declare **public** functions;

};

The lines for actually declaring the variables are the same as any other C++ variable declaration:

datatype variablename;

The Matrix class has three private variables:

* grid - a 2D float vector to hold the matrix values
* rows - the number of rows in the matrix
* columns - the number of columns in the matrix

The rows and columns variables should be declared as a size\_type. A size\_type variable holds the size of a vector.

If your vector holds integers, the size\_type declaration looks like this:

std::vector<**int**>::size\_type variablename;

If your vector holds floats, then the size\_type declaration would look like this:

std::vector<**float**>::size\_type variablename;

The value that goes inside the brackets <> is based on whatever the original vector declaration was. A size\_type variable is actually an unsigned int. The size\_type variable is guaranteed to be able to hold up to the maximum size of a float vector.

# Solution

**class** Matrix

{

**private**:

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

std::vector<**float**>::size\_type rows;

std::vector<**float**>::size\_type cols;

};

In the next step, you will declare your class functions and then define your class functions.

# Class Functions

To write functions in your Matrix class, you need to declare those functions first. For the Matrix class, you can think of these functions as belonging to three separate categories:

* constructor functions
* set and get functions
* functions for Matrix functionality

Declaring these functions will be exactly like declaring functions in the previous lesson. The difference is that now you have to decide if a function is private, protected or public. And the function declarations go inside the class declaration.

You will define your functions in matrix.cpp. But first, let's briefly talk about each type of function. Constructor functions are for initializing objects. Python does this with the def \_\_init\_\_ syntax. The C++ syntax is a bit different, and you will learn about the differences in the next part of the lesson.

Set and get functions are specifically for accessing and assigning values to private variables. Because an object will not have direct access to private variables, the set and get functions give indirect access. Set and get functions have the same syntax as any other C++ function. Using set and get is a convention of object oriented programming rather than a specific C++ syntax.

And finally, there are the functions that consist of the matrix functionality such as printing out the matrix, adding matrices together, multiplying matrices, etc. You will implement these functions as part of the exercises.

Go to the next part of the exercise to declare and define the Matrix constructor functions.

# Set and Get Function Declarations

Set and Get functions allow your objects to gain access to private variables. An object cannot access a private variable directly, so instead, set and get functions are used. You can see how this is done in the Gaussian object from earlier in the lesson.

Here were the declarations for the set and get functions:

**class** Gaussian

{

**private**:

...

**public**:

...

**void** **setMu**(**float**);

**void** **setSigma2**(**float**);

**float** **getMu**();

**float** **getSigma2**();

....

};

A set function changes the value of a variable whereas a get function returns the value of a variable. You'll notice that set and get function syntax is the same as any regular function. In fact, set and get are conventions rather than specific to C++. It's traditional to name these functions getVariablename() and setVariablename() although there is no requirement to do so.

You would declare set and get functions as public so that objects could have access to these functions.

### QUESTION 1 OF 2

Why do set functions return a value of void?

* A set function only changes the value of a variable.
* The void return type implies that the function does not return anything.
* A set function returns an empty variable
* A set function returns the value of a variable

SUBMIT

# Functions for Matrix Functionality

The third set of functions to declare are for the matrix functionality. The syntax is exactly the same as the get and set function syntax as well as any normal C++ function. You need to specify the return datatype, the function name, and the datatype for the input variables.

For example, the Gaussian class had three functions: evaluate, multiply and add. Here is how these functions were declared in the gaussian.h file:

**class** Gaussian

{

....

**public**:

...

*// functions to evaluate*

**float** **evaluate** (**float**);

Gaussian **multiply** (Gaussian);

Gaussian **add** (Gaussian);

};

# Declaring Functions in the Matrix Class

Now it's your turn to declare functions in the matrix.h file. Fill out the TODOS in the matrix.h file below. Don't forget that every function in matrix.cpp needs to be declared in matrix.h.

The answer is underneath the code block.

# Declaring Constructor Functions

Both Python and C++ have constructor functions. Constructor functions define what happens when you instantiate an object.

#### Python Constructor

These are the functions that define what happens when an object is instantiated. In Python, the syntax is:

**def** **\_\_init\_\_**(self, variable1, variable2, ..., variablen):

self.variable1 = variable1

self.variable2 = variable2

self.variablen = variablen

#### C++ Constructor Declaration

In C++, you declare a constructor like this:

Classname (datatype variable1, datatype variable2, …, datatype variablen);

You can also simultaneously declare a default constructor function, which implies the function has no inputs:

Classname ();

This default constructor function is used when you instantiate an object without providing values for the variables. To be more concrete, you will initialize a Matrix variable with a two-dimensional vector. If you do not provide a two-dimensional vector, you could initialize your Matrix variable with a default vector. This second case is what the empty constructor function is for.

The Gaussian constructor declarations looked like this:

**class** Gaussian

{

**private**:

...

**public**:

...

Gaussian ();

Gaussian (**float**, **float**);

....

};

Whether or not you use a default constructor function will depend on your particular application and use-case. For example, if an object always has the same initial values, then it would make sense to have a default constructor function.

# Defining Constructor Functions

Once you've declared your constructor functions, you need to actually define them in a .cpp file.

The constructor function definitions have the following syntax:

*// empty constructor function syntax*

Classname::ClassName() {

constructor function definition

}

*// constructor function syntax*

Classname::ClassName(datatype variable1, datatype variable2, …, datatype variablen) {

constructor function definition

}

You can see how this was done for the Gaussian class:

Gaussian::Gaussian() {

mu = 0;

sigma2 = 1;

}

Gaussian::Gaussian (**float** average, **float** sigma) {

mu = average;

sigma2 = sigma;

}

Note that constructor functions do not return anything. They merely initialize class variables. You might also be wondering how the function definitions can access mu and sigma2 if those were private variables. Remember that private variables can be accessed from within the class code itself but not from outside the class.

# Initializing with Default Values

In both Python and C++, you can use default values in your construction functions. In Python, the syntax is:

**def** **\_\_init\_\_**(self, variable1 = default1, variable2 = default2, ..., variablen = defaultn):

self.variable1 = variable1

self.variable2 = variable2

self.variablen = variablen

You can also get this functionality in C++ although the syntax might not be what you'd expect; you actually define default values in the .h file function definition. Here is a trivial example for an addition Class that holds two integers and outputs their sum.

Here is the header file add.h:

**class** Add

{

**public**:

**int** a;

**int** b;

Add(**int**, **int** second = 17);

**int** **addition**();

};

and then here are the definitions in add.cpp:

**#include "add.h"**

Add::Add(**int** first, **int** second) {

a = first;

b = second;

}

**int** Add::addition() {

**return** a + b;

}

Notice that the default value was declared in the header file. Now, if you only specify one value when instantiating an add object, variable b will have a default value of 17:

**#include <iostream>**

**#include "add.h"**

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

Add **adder**(5);

std::cout << adder.addition() << std::endl;

**return** 0;

}

The above code outputs 22.

# Declare and Define Matrix Class Constructors

Now it's your turn. Fill out the TODO sections in the matrix.h and matrix.cpp files.

# Set and Get Function Declarations

Set and Get functions allow your objects to gain access to private variables since objects cannot access private variables directly. You can see how this is done in the Gaussian object from earlier in the lesson.

Here were the declarations for the set and get functions:

**class** Gaussian

{

**private**:

...

**public**:

...

**void** **setMu**(**float**);

**void** **setSigma2**(**float**);

**float** **getMu**();

**float** **getSigma2**();

....

};

And here were the function definitions:

**void** Gaussian::setMu (**float** average) {

mu = average;

}

**void** Gaussian::setSigma2 (**float** sigma) {

sigma2 = sigma;

}

**float** Gaussian::getMu () {

**return** mu;

}

**float** Gaussian::getSigma2() {

**return** sigma2;

}

The syntax for defining set or get functions is the same as any other class function (besides constructors):

**return** datatype Classname::functionname() {

code to define the function;

}

In fact, get and set functions are a convention rather than a special function with a special syntax. It's traditional to name these functions getVariablename() and setVariablename() although there is no requirement to do so.

You would declare set and get functions as public so that objects could have access to these functions.

# Matrix Class Set and Get Functions

Continue developing your Matrix class code.

* Use a set function to be able to modify the grid variable.
* All three private variables (grid, rows, cols) should have get functions.

Make sure to fill out the TODOs in both matrix.cpp and matrix.h

# Matrix Functions

The last part of the Matrix class involves implementing the matrix functionality. You are welcome to program as many matrix operations as you'd like: addition, multiplication, transpose, inverse, etc.

We recommend at least implementing matrix addition and a function called matrix\_print that outputs the matrix to the terminal using cout. In the solution given at the end of this page, we've also provided code for a matrix\_transpose function.

Implementing these class functions is the same as implementing the get and set functions from the previous part of the lesson; you will need to declare your functions in matrix.h and define your functions in matrix.cpp. The general syntax is the same:

#### class function declaration syntax

output\_datatype functionname(datatype variable1,

datatype variable2, ..., datatype variablen)

#### class function definition syntax

output\_datatype Classname::functionname(datatype variable1,

datatype variable2, ..., datatype variablen) {

code defining the function;

}

# Writing the Matrix Functions

In this exercise, you will declare and define a Matrix class function that adds two matrices together. Here are the inputs and outputs of the matrix addition function:

INPUTS:

* a matrix, which will be added to the grid variable

OUTPUTS

* a matrix containing the sum of the grid variable matrix and input matrix

Because the input to the matrix\_addition function is a Matrix, you will need to declare and define your function using the Matrix class as the data type. This might seem a bit confusing, but the Gaussian class presented earlier in the lesson did the same thing with the mul and add functions. You can use those as a guide for writing your matrix\_addition functions.

As a reminder, here are the function declarations for the mul and add functions in gaussian.h:

Gaussian **mul** (Gaussian);

Gaussian **add** (Gaussian);

Both of these functions receive a Gaussian and output a Gaussian. Here are the function definitions from gaussian.cpp:

Gaussian Gaussian::mul(Gaussian other) {

**float** denominator;

**float** numerator;

**float** new\_mu;

**float** new\_var;

denominator = sigma2 + other.getSigma2();

numerator = mu \* other.getSigma2() + other.getMu() \* sigma2;

new\_mu = numerator / denominator;

new\_var = 1.0 / ( (1.0 / sigma2) + (1.0 / other.sigma2) );

**return** Gaussian(new\_mu, new\_var);

}

Gaussian Gaussian::add(Gaussian other) {

**float** new\_mu;

**float** new\_sigma2;

new\_mu = mu + other.getMu();

new\_sigma2 = sigma2 + other.getSigma2();

**return** Gaussian(new\_mu, new\_sigma2);

}

Although the implementation of the matrix\_addition function will be different, the general structure will be the same as the mul and add functions from the Gaussian example.

You will also write a matrix\_print function that outputs a matrix to the terminal using cout. The matrix\_print function has no inputs and no outputs.

Fill out the TODOS in the matrix.cpp and matrix.h code.

**Solution matrix.h**

**#include <vector>**

**#include <iostream>**

**#include <stdexcept>**

**class** Matrix

{

**private**:

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

std::vector<**float**>::size\_type rows;

std::vector<**float**>::size\_type cols;

**public**:

*// constructor functions*

Matrix ();

Matrix (std::vector< std::vector<**float**> >);

*// set functions*

**void** **setGrid**(std::vector< std::vector<**float**> >);

*// get functions*

std::vector< std::vector<**float**> > getGrid();

std::vector<**float**>::size\_type **getRows**();

std::vector<**float**>::size\_type **getCols**();

*// matrix functions*

Matrix **matrix\_transpose**();

Matrix **matrix\_addition**(Matrix);

*// matrix printing*

**void** **matrix\_print**();

};

**Solution matrix.cpp**

**#include "matrix.h"**

Matrix::Matrix() {

std::vector <std:: vector <**float**> > initial\_grid (10, std::vector <**float**>(5, 0.5));

grid = initial\_grid;

rows = initial\_grid.size();

cols = initial\_grid[0].size();

}

Matrix::Matrix(std::vector <std:: vector <**float**> > initial\_grid) {

grid = initial\_grid;

rows = initial\_grid.size();

cols = initial\_grid[0].size();

}

**void** Matrix::setGrid(std::vector< std::vector<**float**> > new\_grid) {

grid = new\_grid;

rows = new\_grid.size();

cols = new\_grid[0].size();

}

std::vector< std::vector<**float**> > Matrix::getGrid() {

**return** grid;

}

std::vector<**float**>::size\_type Matrix::getRows() {

**return** rows;

}

std::vector<**float**>::size\_type Matrix::getCols() {

**return** cols;

}

Matrix Matrix::matrix\_transpose() {

std::vector< std::vector<**float**> > new\_grid;

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

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

row.clear();

**for** (**int** j = 0; j < rows; j++) {

row.push\_back(grid[j][i]);

}

new\_grid.push\_back(row);

}

**return** Matrix(new\_grid);

}

Matrix Matrix::matrix\_addition(Matrix other) {

**if** ((rows != other.getRows()) || (cols != other.getCols())) {

**throw** std::invalid\_argument( "matrices are not the same size" );

}

std::vector< std::vector<**float**> > othergrid = other.getGrid();

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

std::vector<**float**> new\_row;

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

new\_row.clear();

**for** (**int** j = 0; j < cols; j++) {

new\_row.push\_back(grid[i][j] + othergrid[i][j]);

}

result.push\_back(new\_row);

}

**return** Matrix(result);

}

**void** Matrix::matrix\_print() {

std::cout << std::endl;

**for** (**int** i = 0; i < rows; i++)

{

**for** (**int** j = 0; j < cols; j++)

{

std::cout << grid[i][j] << " ";

}

std::cout << std::endl;

}

std::cout << std::endl;

}

NEXT

# ifndef

In this case, you don't really need an ifndef statement because the code is simple. You have only written one class, so there isn't a way to mistakenly include another class multiple times. However, it's a good habit to write inclusion guards with an ifndef statement.

For this exercise, use the ifndef syntax to write an inclusion guard in the matrix.h file.

# Instantiate an Object

Now it's time to use your matrix class in a program! The C++ syntax for instantiating an object is like this:

Classname **objectname**(inputs **for** initializing an object of Classname);

Then you can access any public variables like:

objectname.variablename

And you can access your public functions with:

objectname.methodname(inputs)

Remember that any private variables or functions will not be available to your program. That was why you wrote the public get and set functions for your private variables.

# Gaussian.cpp Example

Before you start using your matrix class, here is a reminder of what the main.cpp looked like for the Gaussian.cpp example:

**#include <iostream>**

**#include "gaussian.h"**

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

{

Gaussian **mygaussian**(30.0,20.0);

Gaussian **othergaussian**(10.0,30.0);

std::cout << "average " << mygaussian.getMu() << std::endl;

std::cout << "evaluation " << mygaussian.evaluate(15.0) << std::endl;

std::cout << "mul results sigma " <<

mygaussian.mul(othergaussian).getSigma2() << std::endl;

std::cout << "mul results average " <<

mygaussian.mul(othergaussian).getMu() << std::endl;

std::cout << "add results sigma " <<

mygaussian.add(othergaussian).getSigma2() << std::endl;

std::cout << "add results average " <<

mygaussian.add(othergaussian).getMu() << std::endl;

**return** 0;

}

Now it's your turn to instantiate a Matrix object. You will find some starter code below with a few TODOs.

**Solution main.cpp**

#include <iostream>

#include <vector>

#include "matrix.h"

int main () {

// assign a 7x5 matrix to the variable initial\_grid

// all values in the matrix are 0.4

std::vector <std:: vector <float> >

initial\_grid (7, std::vector <float>(5, 0.4));

// TODO: Use the initial grid variable to instantiate a matrix object

// The matrix object should be called matrixa

Matrix matrixa(initial\_grid);

// TODO: Use the matrix\_print() method to print out matrixa

matrixa.matrix\_print();

// TODO: Print out the number of rows in matrixa. You will need

// to use the getRows() function and std::cout

std::cout << matrixa.getRows();

// TODO: Print out the number of columns in matrixa

std::cout << matrixa.getCols();

// TODO: Take the transpose of matrixa and store the results in

// a variable called transposea

Matrix transposea = matrixa.matrix\_transpose();

// TODO: Print out transposea

transposea.matrix\_print();

// Now you will use another 7x5 matrix called matrixb to

// give the results of the matrix\_addition function

// 7x5 2-dimensional vector with values 0.2

std::vector <std:: vector <float> >

second\_grid (7, std::vector <float>(5, 0.2));

// TODO: Instantiate an object called matrixb. Use the second\_grid

// variable as the input to initialize matrixb

Matrix matrixb(second\_grid);

// TOOD: Add matrixa and matrixb. Store the results in a new matrix

// variable called matrixsum

Matrix matrixsum(matrixa.matrix\_addition(matrixb));

// TODO: Print out the matrix contained in the matrixsum variable

matrixsum.matrix\_print();

return 0;

}

NEXT

# Compiling your Program

If you haven't gotten C++ running locally on your computer, now's a good time to get that done!

This is just a quick note about compiling and running the matrix code on your local computer. Put your main.cpp, matrix.cpp and matrix.h into the same directory. On Linux and Mac you can compile your code with a command like:

g++ main.cpp matrix.cpp

or whatever the equivalent is for your system or compiler. You need to compile both main.cpp and matrix.cpp for the code to compile. Then you can execute your code with something like

./a.out

On Windows, compiling and executing your code will look something like this:

cl /W4 /EHsc main.cpp matrix.cpp

main