

1. Overview

<https://youtu.be/8V271hq1dfE>

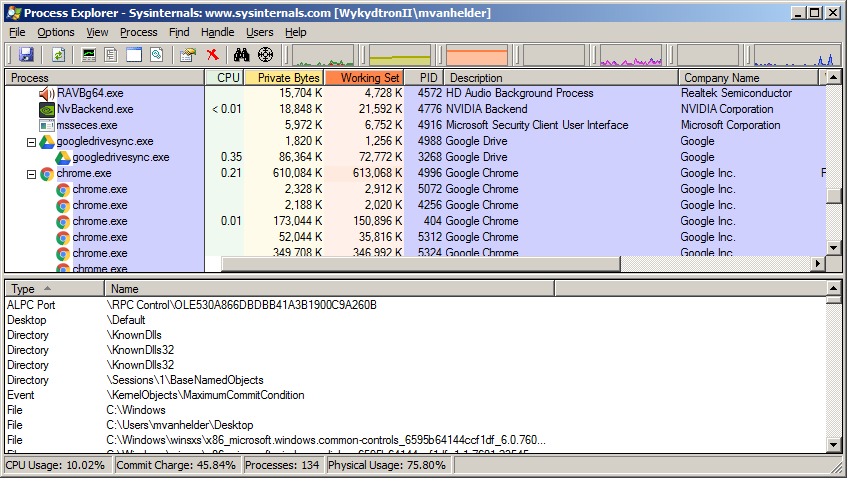
1. Bjarne on the Creation of C++ and classes

<https://youtu.be/pqPvz33zVkE>

1. System Monitor Project

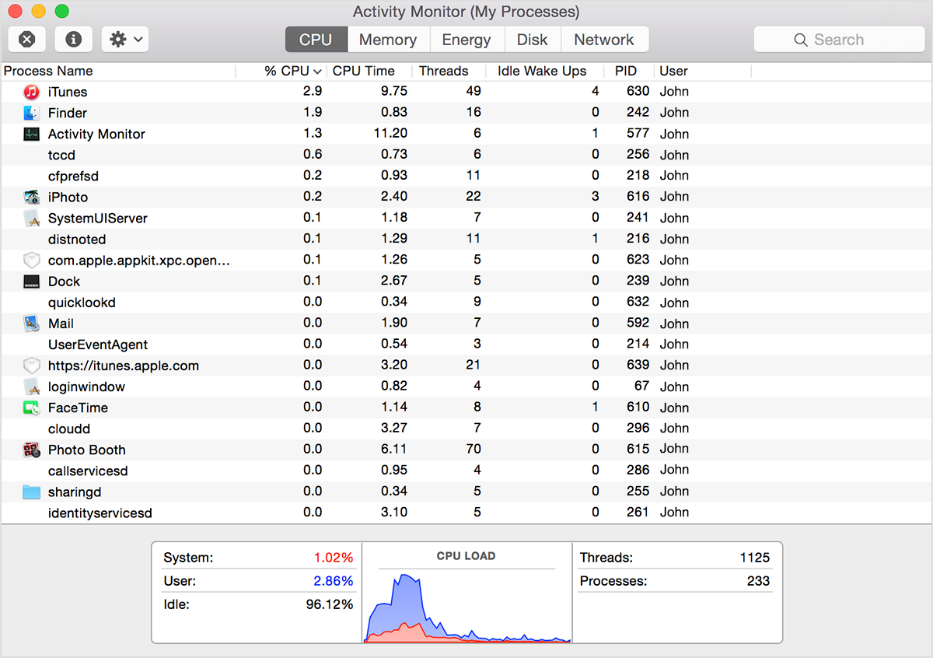
<https://youtu.be/pUoA2pFOxbI>

If you use Windows, then you might be familiar with the [**Process Explorer**](https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer) or [**Process Monitor**](https://docs.microsoft.com/en-us/sysinternals/downloads/procmon) applications:

**[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/ea33b353-b2ca-44e6-9cf8-2a34879fe005/concepts/7558eefc-b3c2-48ae-8634-b90f103c696d)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/ea33b353-b2ca-44e6-9cf8-2a34879fe005/concepts/7558eefc-b3c2-48ae-8634-b90f103c696d)**

**[Source:](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/ea33b353-b2ca-44e6-9cf8-2a34879fe005/concepts/7558eefc-b3c2-48ae-8634-b90f103c696d)**[**PC World**](https://www.pcworld.com/article/3181348/how-to-use-process-explorer-microsofts-free-supercharged-task-manager-alternative.html)

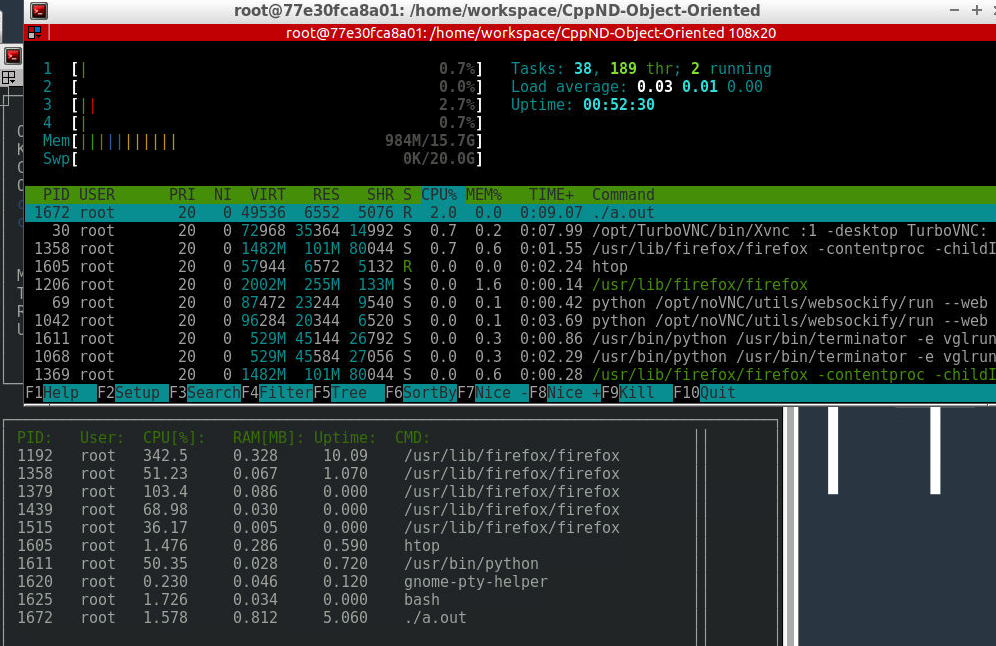
On the other hand, if you are a Mac user, you've probably seen the [**Activity Monitor**](https://support.apple.com/en-us/HT201464) before:

**[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/ea33b353-b2ca-44e6-9cf8-2a34879fe005/concepts/7558eefc-b3c2-48ae-8634-b90f103c696d)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/ea33b353-b2ca-44e6-9cf8-2a34879fe005/concepts/7558eefc-b3c2-48ae-8634-b90f103c696d)**

And finally, if you are a Linux user, you may have used [**top**](http://man7.org/linux/man-pages/man1/top.1.html) or **[htop](https://en.wikipedia.org/wiki/Htop" \t "_blank)** to view active processes on your computer.

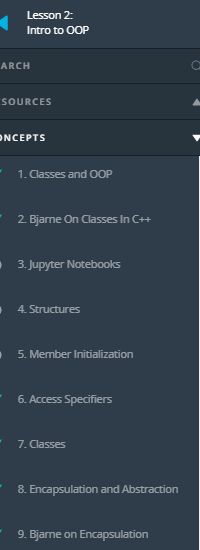
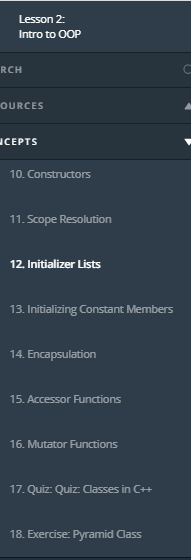
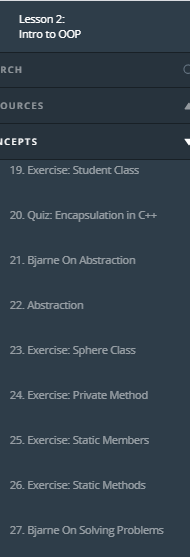
At the end of this course, you will use C++ to build a process monitor, similar to htop . This process monitor will run on Linux.

The process monitor will allow you to see all the active processes on the system, with their corresponding process ids (PIDs), CPU usage, and memory usage:

**[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/ea33b353-b2ca-44e6-9cf8-2a34879fe005/concepts/7558eefc-b3c2-48ae-8634-b90f103c696d)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/ea33b353-b2ca-44e6-9cf8-2a34879fe005/concepts/7558eefc-b3c2-48ae-8634-b90f103c696d)**

1. Let’s Get started

<https://youtu.be/_uuC7FphqcA>

1. Classes and OOP

<https://youtu.be/0Z9vdXNyEVc>

1. Bjarne on classes in c++

<https://youtu.be/leLjGXFcycE>

1. Jupyter Notebooks

# Jupyter Notebooks

**[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/287ef495-0297-459f-b339-cfe087c7b11a/concepts/a19329b5-7a2a-407c-8571-bff7827f50a4)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/287ef495-0297-459f-b339-cfe087c7b11a/concepts/a19329b5-7a2a-407c-8571-bff7827f50a4)**

## Jupyter with C++

In this lesson, you'll be writing and testing lots of C++ code. C++ is a compiled language, which is to say there is a separate program - the compiler - that converts your code to an executable program that the computer can run. This means that, after you save a new C++ program to file, running it is normally a two step process:

1. Compile your code with a compiler.
2. Run the executable file that the compiler outputs.

For example, in the notebook exercises that follow, you'll be saving your code in a file, let's say filename.cpp in a folder called /code. To compile it using the C++17 standard, you can run the following command:

g++ -std=c++17 ./code/filename.cpp

And then to run the resulting executable file, you can run:

./a.out

## Jupyter Notebooks in Udacity Classroom

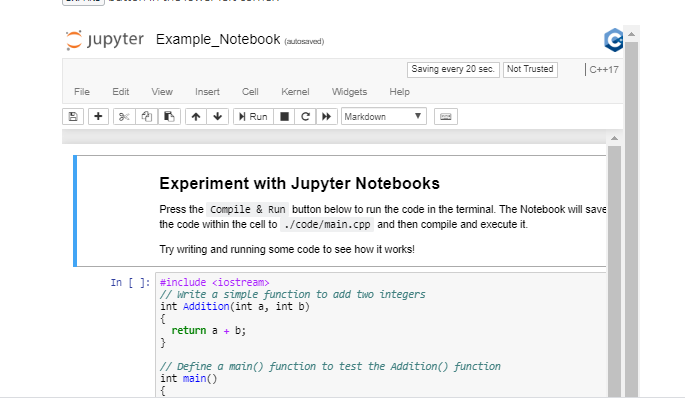
In this lesson, you will save, compile, and run executables over and over. To make your life simpler, we've set things up so you can save, compile and run with the click of a single Compile & Run button in the **[Jupyter](https://jupyter.org/index.html" \t "_blank)** Notebooks.

Later, when you build the project, you'll move out of the notebooks and into a Linux virtual machine. At that point, you'll need to remember to compile and run the programs yourself!

If you haven't seen Jupyter Notebooks before, you can test one out below. A Notebook is a web application that allows for code, text, and visualizations to be combined and shared.

Check out the Notebook below for an example of how these will be used in the course.

When you use a Notebook Workspace, we encourage you to expand to a full screen view. Click on the EXPAND button in the lower left corner.



1. Structures

<https://youtu.be/iKIEdY_pdzY>

Structures

Structures allow developers to create their own types ("user-defined" types) to aggregate data relevant to their needs.

For example, a user might define a Rectangle structure to hold data about rectangles used in a program.

**struct** Rectangle {

**float** length;

**float** width;

};

## Types

Every C++ variable is defined with a [**type**](http://www.cplusplus.com/doc/tutorial/variables/).

**int** value;

**Rectangle** rectangle;

**Sphere** earth;

In this example, the "type" of value is int. Furthermore, rectangle is "of type" Rectangle, and earth has type Sphere.

## Fundamental Types

C++ includes [**fundamental types**](https://en.cppreference.com/w/cpp/language/types), such as int and float. These fundamental types are sometimes called [**"primitives"**](https://www.geeksforgeeks.org/c-data-types/).

The Standard Library [includes additional types](, such as std::size\_t and std::string.

## User-Defined Types

Structures are "user-defined" types. Structures are a way for programmers to create types that aggregate and store data in way that makes sense in the context of a program.

For example, C++ does not have a fundamental type for storing a date. (The Standard Library does include types related to [**time**](https://en.cppreference.com/w/cpp/chrono), which can be converted to dates.)

A programmer might desire to create a type to store a date.

Consider the following example:

**struct** Date {

**int** day;

**int** month;

**int** year;

};

The code above creates a structure containing three "member variables" of type int: day, month and year.

If you then create an "instance" of this structure, you can initialize these member variables:

*// Create an instance of the Date structure*

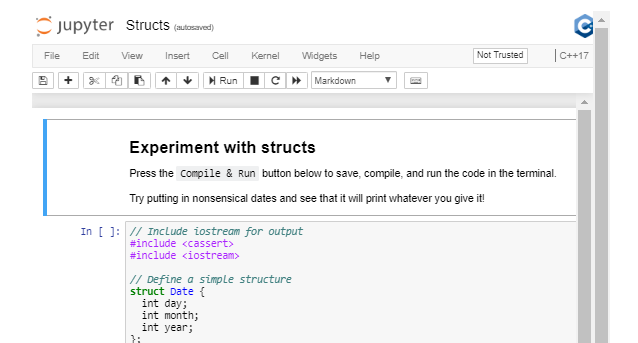
Date date;

*// Initialize the attributes of Date*

date.day = 1;

date.month = 10;

date.year = 2019;



1. Member Initialization

Member Initialization

Generally, we want to avoid instantiating an object with undefined members. Ideally, we would like all members of an object to be in a valid state once the object is instantiated. We can change the values of the members later, but we want to avoid any situation in which the members are ever in an invalid state or undefined.

In order to ensure that objects of our Date structure always start in a valid state, we can initialize the members from within the structure definition.

**struct** Date {

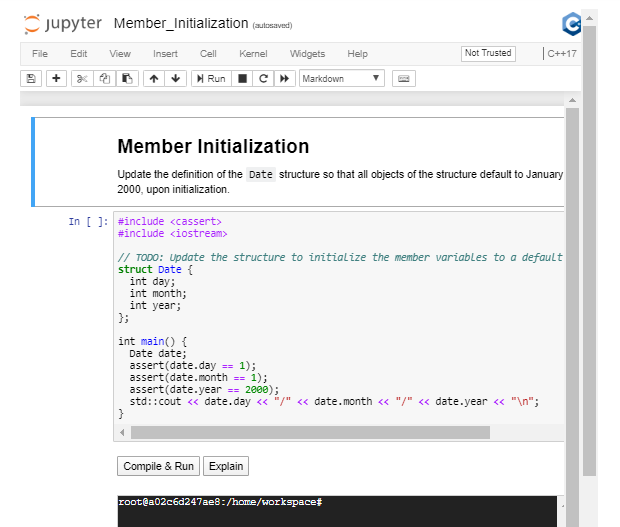
**int** day{1};

**int** month{1};

**int** year{0};

};

There are also several other approaches to either initialize or assign member variables when the object is instantiated. For now, however, this approach ensures that every object of Date begins its life in a defined and valid state.



1. Access Specifiers

<https://youtu.be/ZEt2LTRc2D0>

# Access Specifiers

Members of a structure can be specified as public or private.

By default, all members of a structure are public, unless they are specifically marked private.

Public members can be changed directly, by any user of the object, whereas private members can only be changed by the object itself.

## Private Members

This is an implementation of the Date structure, with all members marked as private.

**struct** Date {

**private**:

**int** day{1};

**int** month{1};

**int** year{0};

};

Private members of a class are accessible only from within other member functions of the same class (or from their "friends", which we’ll talk about later).

There is a third access modifier called protected, which implies that members are accessible from other member functions of the same class (or from their "friends"), and also from members of their derived classes. We'll also discuss about derived classes later, when we learn about inheritance.

## Accessors And Mutators

To access private members, we typically define public "accessor" and "mutator" member functions (sometimes called "getter" and "setter" functions).

**struct** Date {

**public**:

**int** **Day**() { **return** day; }

**void** **Day**(**int** day) { **this**.day = day; }

**int** **Month**() { **return** month; }

**void** **Month**(**int** month) { **this**.month = month; }

**int** **Year**() { **return** year; }

**void** **Year**(**int** year) { **this**.year = year; }

**private**:

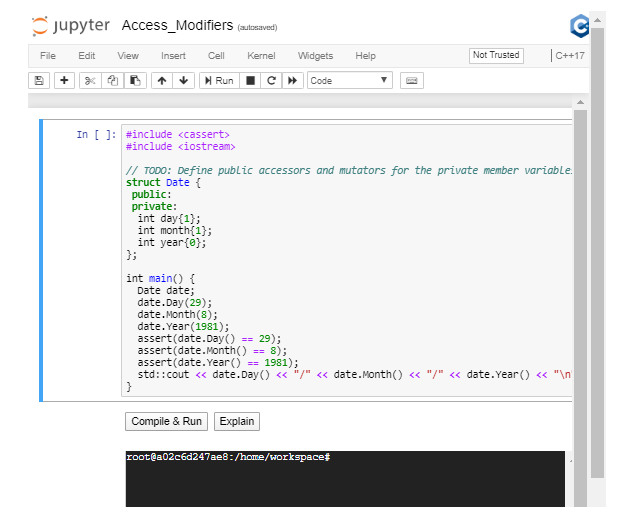
**int** day{1};

**int** month{1};

**int** year{0};

};

In the last example, you saw how to create a setter function for class member attributes. Check out the code in the Notebook below to play around a bit with access modifiers as well as setter and getter functions!



## Avoid Trivial Getters And Setters

Sometimes accessors are not necessary, or even advisable. The [**C++ Core Guidelines**](https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rh-get) recommend, "A trivial getter or setter adds no semantic value; the data item could just as well be public."

Here is the example from the Core Guidelines:

**class** **Point** {

**int** x;

**int** y;

**public**:

Point(**int** xx, **int** yy) : x{xx}, y{yy} { }

**int** **get\_x**() **const** { **return** x; } *// const here promises not to modify the object*

**void** **set\_x**(**int** xx) { x = xx; }

**int** **get\_y**() **const** { **return** y; } *// const here promises not to modify the object*

**void** **set\_y**(**int** yy) { y = yy; }

*// no behavioral member functions*

};

This class could be made into a struct, with no logic or "invariants", just passive data. The member variables could both be public, with no accessor functions:

**struct** Point { *// Good: concise*

**int** x {0}; *// public member variable with a default initializer of 0*

**int** y {0}; *// public member variable with a default initializer of 0*

};

1. Classes

<https://youtu.be/FTzrwV2LP5g>

**Classes**

Classes, like structures, provide a way for C++ programmers to aggregate data together in a way that makes sense in the context of a specific program. By convention, programmers use structures when member variables are independent of each other, and [**use classes when member variables are related by an "invariant"**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c2-use-class-if-the-class-has-an-invariant-use-struct-if-the-data-members-can-vary-independently).

**Invariants**

An "invariant" is a rule that limits the values of member variables.

For example, in a Date class, an invariant would specify that the member variable day cannot be less than 0. Another invariant would specify that the value of day cannot exceed 28, 29, 30, or 31, depending on the month and year. Yet another invariant would limit the value of month to the range of 1 to 12.

**Date Class**

Let's define a Date class:

*// Use the keyword “class” to define a Date class:*

**class** Date {

**int** day{1};

**int** month{1};

**int** year{0};

};

So far, this class definition provides no invariants. The data members can vary independently of each other.

There is one subtle but important change that takes place when we change struct Date to class Date. By default, all members of a struct default to public, whereas all members of a class default to private. Since we have not specified access for the members of class Date, all of the members are private. In fact, we are not able to assign value to them at all!

**Date Accessors And Mutators**

As the first step to adding the appropriate invariants, let's specify that the member variable day is private. In order to access this member, we'll provide accessor and mutatot functions. Then we can add the appropriate invariants to the mutators.

**class** Date {

**public**:

**int** **Day**() { **return** day\_; }

**void** **Day**(**int** d) { day\_ = d; }

**private**:

**int** day\_{1};

**int** month\_{1};

**int** year\_{0};

};

**Date Invariants**

Now we can add the invariants within the mutators.

**class** Date {

**public**:

**int** **Day**() { **return** day; }

**void** **Day**(**int** d) {

**if** (d >= 1 && d <= 31) day\_ = d;

}

**private**:

**int** day\_{1};

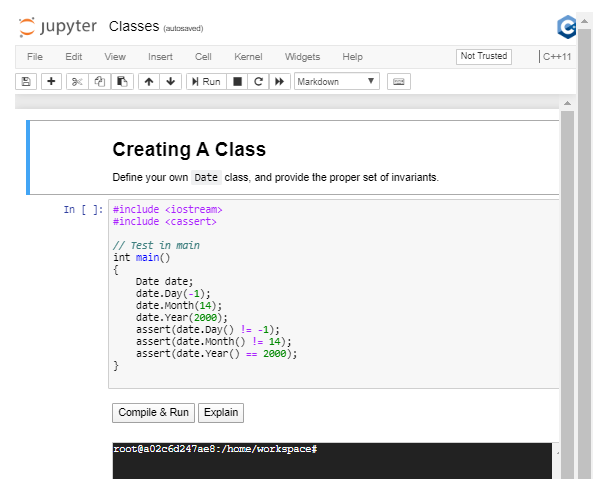
**int** month\_{1};

**int** year\_{0};

};

Now we have a set of invariants for the the class members!

As a general rule, member data subject to an invariant should be specified private, in order to enforce the invariant before updating the member's value.



1. Encapsulation and Abstraction

<https://youtu.be/KXojXgyW-O8>

1. Bjarne on Encapsulation

<https://youtu.be/M27atfK73_s>

1. Constructors

<https://youtu.be/zw5fqCmxD8o>

# Constructors

Constructors are member functions of a class or struct that initialize an object. The Core Guidelines [**define a constructor**](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/287ef495-0297-459f-b339-cfe087c7b11a/concepts/(http:/isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary)) as:

constructor: an operation that initializes (“constructs”) an object. Typically a constructor establishes an invariant and often acquires resources needed for an object to be used (which are then typically released by a destructor).

A constructor can take arguments, which can be used to assign values to member variables.

**class** Date {

**public**:

Date(**int** d, **int** m, **int** y) { *// This is a constructor.*

Day(d);

}

**int** **Day**() { **return** day; }

**void** **Day**(**int** d) {

**if** (d >= 1 && d <= 31) day = d;

}

**int** **Month**() { **return** month; }

**void** **Month**(**int** m) {

**if** (m >= 1 && m <= 12) month = m;

}

**int** **Year**() { **return** year\_; }

**void** **Year**(**int** y) { year = y; }

**private**:

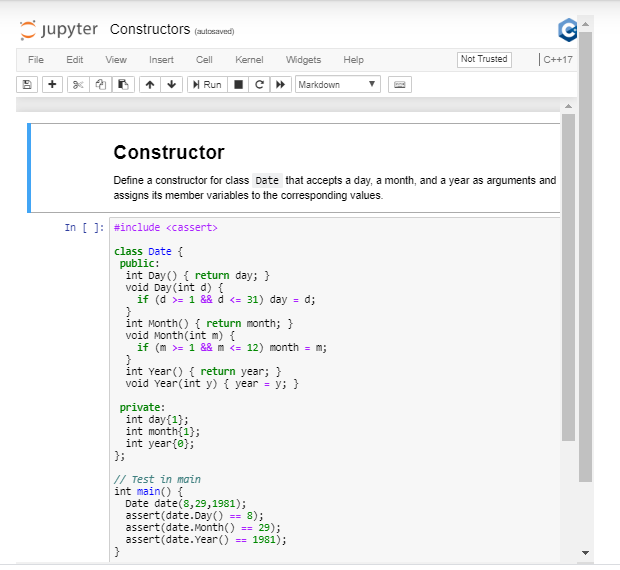
**int** day{1};

**int** month{1};

**int** year{0};

};

As you can see, a constructor is also able to call other member functions of the object it is constructing. In the example above, Date(int d, int m, int y) assigns a member variable by calling Day(int d).



**Default Constructor**

A class object is always initialized by calling a constructor. That might lead you to wonder how it is possible to initialize a class or structure that does not define any constructor at all.

For example:

**class** Date {

**int** day{1};

**int** month{1};

**int** year{0};

};

We can initialize an object of this class, even though this class does not explicitly define a constructor.

This is possible because of the [**default constructor**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#cctor-constructors-assignments-and-destructors). [**The compiler will define a default constructor**](https://en.cppreference.com/w/cpp/language/default_constructor), which accepts no arguments, for any class or structure that does not contain an explicitly-defined constructor.

1. Scope Resolution

<https://youtu.be/U2ItvJEwuHQ>

**Scope Resolution**

C++ allows different [**identifiers**](https://en.cppreference.com/w/cpp/language/identifiers) (variable and function names) to have the same name, as long as they have different scope. For example, two different functions can each declare the variable int i, because each variable only exists within the scope of its parent function.

In some cases, scopes can overlap, in which case the compiler may need assistance in determining which identifier the programmer means to use. The process of determining which identifier to use is called [**"scope resolution"**](https://docs.microsoft.com/en-us/cpp/cpp/scope-resolution-operator?view=vs-2019).

**Scope Resultion Operator**

:: is the [**scope resolution operator**](https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_74/rzarg/cplr175.htm). We can use this operator to specify which namespace or class to search in order to resolve an identifier.

Person::move(); \\ **Call** the **move** the **function** that **is** a **member** **of** the Person **class**.

**std**::**map** **m**; \\ Initialize the map container from the C++ Standard Library.

**Class**

Each class provides its own scope. We can use the scope resolution operator to specify identifiers from a class.

This becomes particularly useful if we want to separate class *declaration* from class *definition*.

**class** Date {

**public**:

**int** **Day**() **const** { **return** day; }

**void** **Day**(**int** day); *// Declare member function Date::Day().*

**int** **Month**() **const** { **return** month; }

**void** **Month**(**int** month) {

**if** (month >= 1 && month <= 12) Date::month = month;

}

**int** **Year**() **const** { **return** year; }

**void** **Year**(**int** year) { Date::year = year; }

**private**:

**int** day{1};

**int** month{1};

**int** year{0};

};

*// Define member function Date::Day().*

**void** Date::Day(**int** day) {

**if** (day >= 1 && day <= 31) Date::day = day;

}

**Namespaces**

[**Namespaces**](https://en.cppreference.com/w/cpp/language/namespace) allow programmers to group logically related variables and functions together. Namespaces also help to avoid conflicts between to variables that have the same name in different parts of a program.

**namespace** English {

**void** **Hello**() { std::cout << "Hello, World!\n"; }

} *// namespace English*

**namespace** Spanish {

**void** **Hello**() { std::cout << "Hola, Mundo!\n"; }

} *// namespace Spanish*

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

English::Hello();

Spanish::Hello();

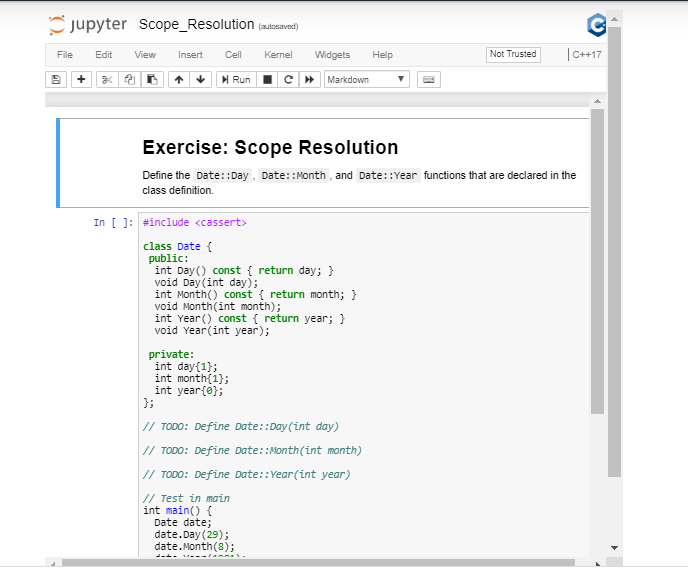
}

In this example, we have two different void Hello() functions. If we put both of these functions in the same namespace, they would conflict and the program would not compile. However, by declaring each of these functions in a separate namespace, they are able to co-exist. Furthermore, we can specify which function to call by prefixing Hello() with the appropriate namespace, followed by the :: operator.

**std Namespace**

You are already familiar with the std namespace, even if you didn't realize quite what it was. std is the namespace used by the [**C++ Standard Library**](https://en.wikipedia.org/wiki/C%2B%2B_Standard_Library).

Classes like [**std::vector**](https://en.cppreference.com/w/cpp/container/vector) and functions like [**std::sort**](https://en.cppreference.com/w/cpp/algorithm/sort) are defined within the std namespace



1. Initializer Lists

<https://youtu.be/cqKuYu1oiow>

**Initializer Lists**

[**Initializer lists**](https://en.cppreference.com/w/cpp/language/initializer_list) initialize member variables to specific values, just before the class constructor runs. This initialization ensures that class members are automatically initialized when an instance of the class is created.

Date::Date(**int** day, **int** month, **int** year) : year\_(y) {

Day(day);

Month(month);

}

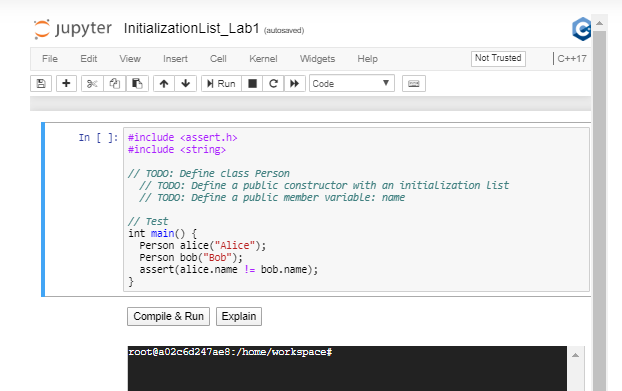
In this example, the member value year is initialized through the initializer list, while day and month are assigned from within the constructor. Assigning day and month allows us to apply the invariants set in the mutator.

In general, [**prefer initialization to assignment**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c49-prefer-initialization-to-assignment-in-constructors). Initialization sets the value as soon as the object exists, whereas assignment sets the value only after the object comes into being. This means that assignment creates and opportunity to accidentally use a variable before its value is set.

In fact, initialization lists ensure that member variables are initialized *before* the object is created. This is why class member variables can be declared const, but only if the member variable is initialized through an initialization list. Trying to initialize a const class member within the body of the constructor will not work.

**Instructions**

1. Declare class Person.
2. Add std::string name to class Person.
3. Create a constructor for class Person.
4. Add an initializer list to the constructor.
5. Create class object.



1. Initializing Constant Members

<https://youtu.be/Ms4Li58ZvwA>

**Exercise: Constructor Syntax**

Initializer lists exist for a number of reasons. First, the compiler can optimize initialization faster from an initialization list than from within the constructor.

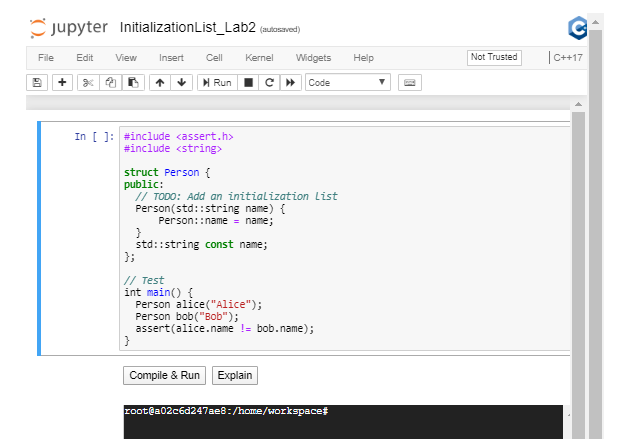
A second reason is a bit of a technical paradox. If you have a const class attribute, you can only initialize it using an initialization list. Otherwise, you would violate the const keyword simply by initializing the member in the constructor!

The third reason is that attributes defined as [**references**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary) must use initialization lists.

This exercise showcases several advantages of initializer lists.

**Instructions**

1. Modify the exist code to use an initialization list.
2. Verify that the test passes.



1. Encapsulation

<https://youtu.be/1-j6AxEoK3c>

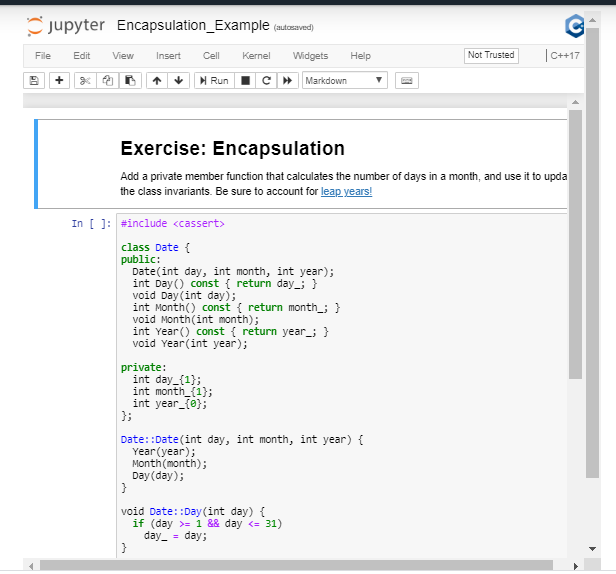
**Encapsulation**

[**Encapsulation**](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming%29) is the grouping together of data and logic into a single unit. In object-oriented programming, classes encapsulate data and functions that operate on that data.

This can be a delicate balance, because on the one hand we want to group together relevant data and functions, but on the hand we want to [**limit member functions to only those functions that need direct access to the representation of a class**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c4-make-a-function-a-member-only-if-it-needs-direct-access-to-the-representation-of-a-class).

In the context of a Date class, a function Date Tomorrow(Date const & date) probably does not need to be encapsulated as a class member. It can exist outside the Date class.

However, a function that calculates the number of days in a month probably should be encapsulated with the class, because the class needs this function in order to operate correctly.



1. Accessor Functions

<https://youtu.be/HfVOiSpzFaA>

**Accessor Functions**

Accessor functions are public member functions that allow users to access an object's data, albeit indirectly.

**const**

Accessors should only retrieve data. They should not change the data stored in the object.

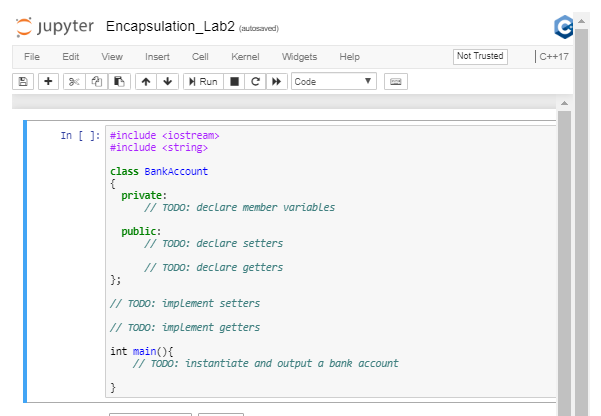
The main role of the const specifier in accessor methods is to protect member data. When you specify a member function as const, the compiler will prohibit that function from changing any of the object's member data.

**Exercise: Bank Account Class**

Your task is to design and implement class called BankAccount. This will be a generic account defined by its account number, the name of the owner and the funds available.

Complete the following steps:

1. Create class called BankAccount
2. Use typical info about bank accounts to design attributes, such as the account number, the owner name, and the available funds.
3. Specify access so that member data are protected from other parts of the program.
4. Create accessor and mutator functions for member data.



1. Mutator Functions

<https://youtu.be/-WRtCE3MZB8>

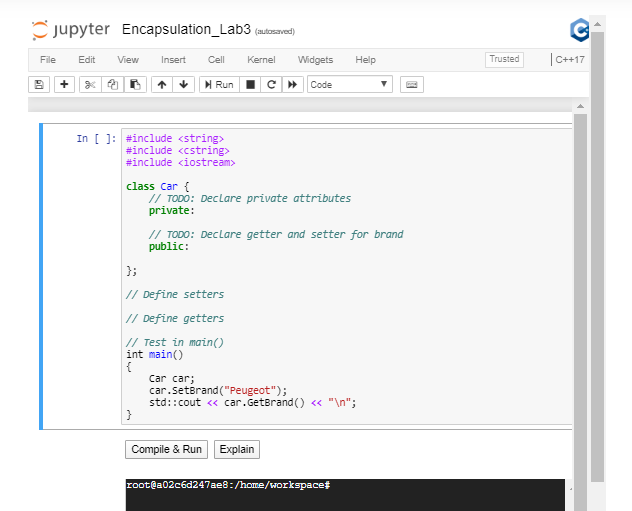
**Mutator Functions**

A mutator ("setter") function can apply logic ("invariants") when updating member data.

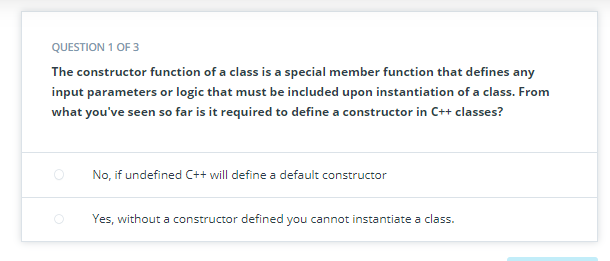
**Exercise: Car Class**

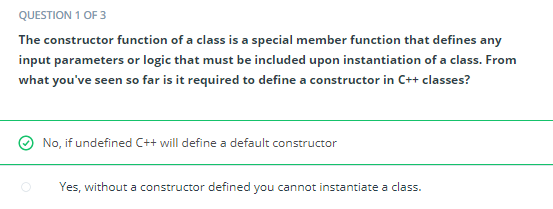
In this lab you will create a setter method that receives data as an argument an converts it to a different type. Specifically, you will receive a string as input and convert it to a character array.

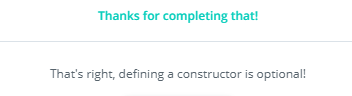
1. Create a class called Car.
2. Create 3 member variables: horsepower, weight and brand. The brand attribute must be a character array.
3. Create accessor and mutator functions for all member data. The mutator function for brand must accept a C++ string as a parameter and convert that string into a [**C-style string**](https://www.learncpp.com/cpp-tutorial/66-c-style-strings/) (a character array ending in null character) to set the value of brand.
4. The accessor function for the brand must return a string, so in this function you first will need to convert brand to std::string, and then return it.

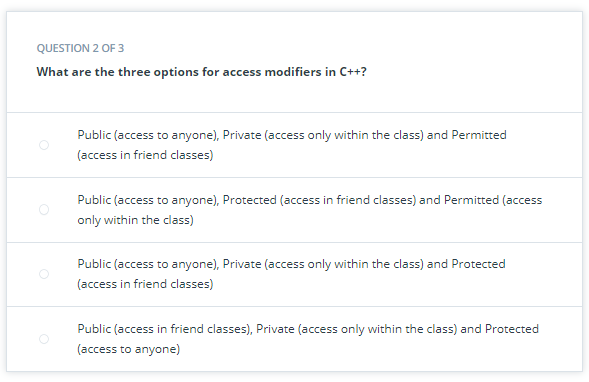


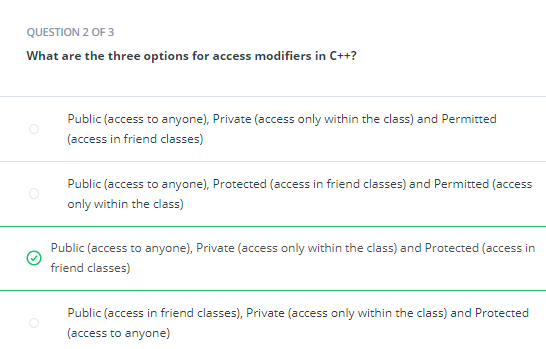
1. Quiz: Classes in C++

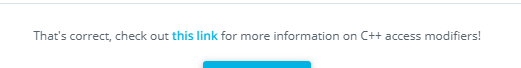




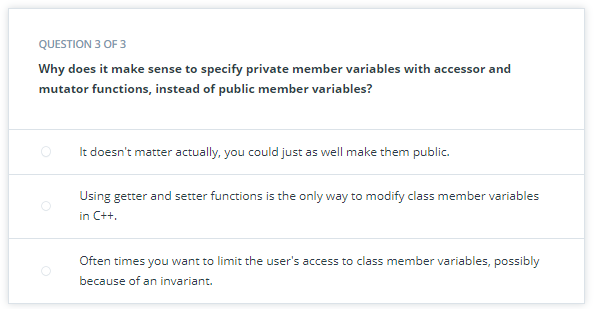


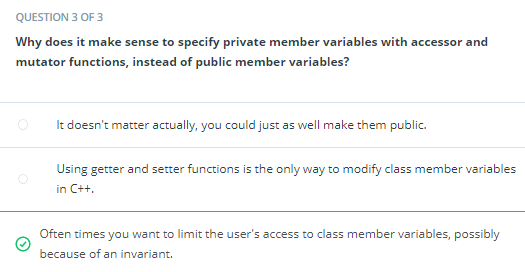


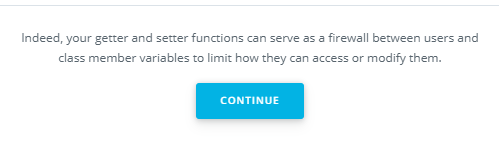




<https://www.tutorialspoint.com/cplusplus/cpp_class_access_modifiers.htm>







1. Exercise: Pyramid Class

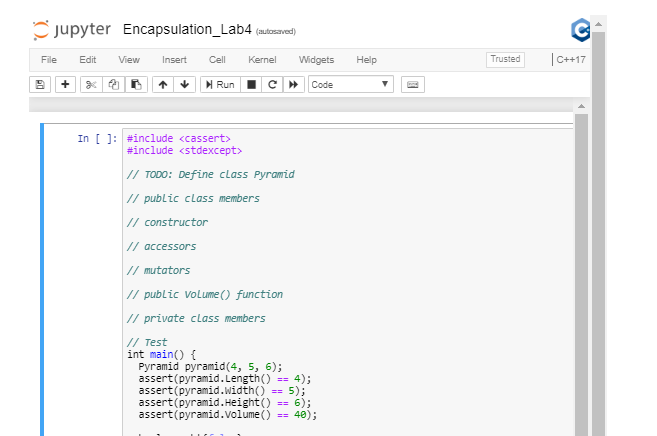
<https://youtu.be/g3CM02Bnamk>

**Exercise: Pyramid Class**

1. Create a class: Pyramid.
2. Create 3 attributes: length, width, and height.
3. Create a constructor to initialize all the attributes.
4. Create accessor and mutator functions for all attributes.
5. Think about the appropriate invariants and enforce them by throwing exceptions.
6. Create a member function to calculate the volume of the pyramid.
7. Optional: create a member function to calculate the surface area of the pyramid.

**Volume**

The volume of a pyramid is [**length \* width \* height / 3**](https://www.wikihow.com/Calculate-the-Volume-of-a-Pyramid).



1. Exercise: Student Class

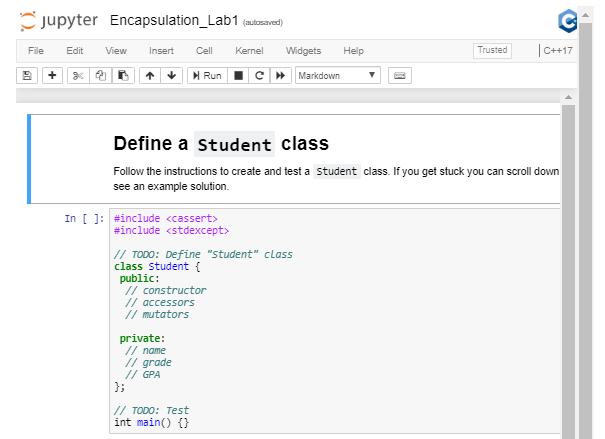
<https://youtu.be/--WHwiY1Z1k>

**Exercise: Student Class**

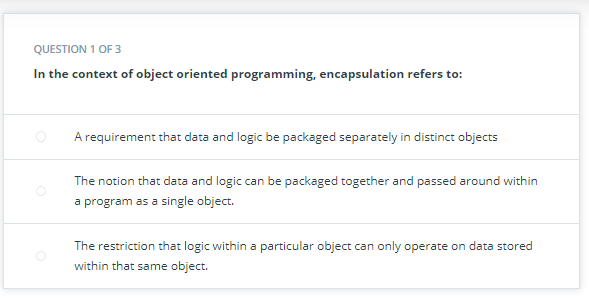
Your goal in this lab is to write a class called Student that will hold data about a particular student. Your class simply needs to store the data and provide accessors and mutators.

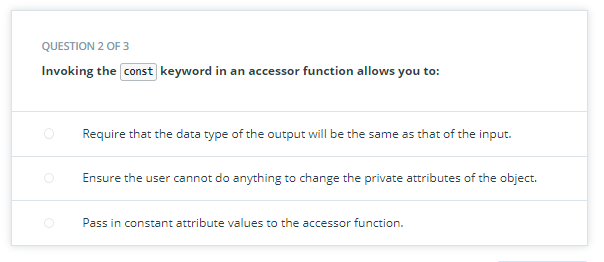
**Directions**

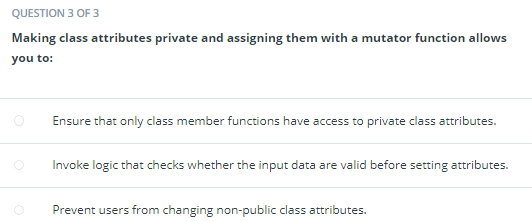
1. Add 3 *private* member variables to the class:
   * Name
   * Grade (e.g. 9th grade)
   * Grade Point Average
2. Write a public constructor function to set the private attributes.
3. Create *public* member functions to set each of the member variables. Before setting the values verify that:
   * Grade is from 0 (kindergarten) to 12
   * GPA is from 0.0 to 4.0
   * The function must either throw an exception if any of the invariants are violated
4. Within the main() (outside of the class), declare an object of type Student and test out each of the member function calls.

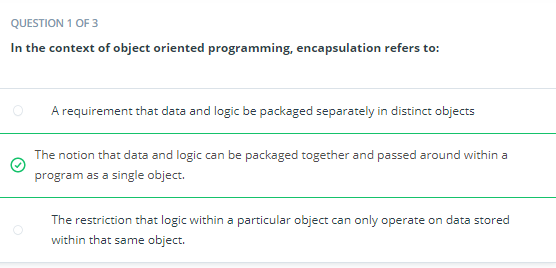


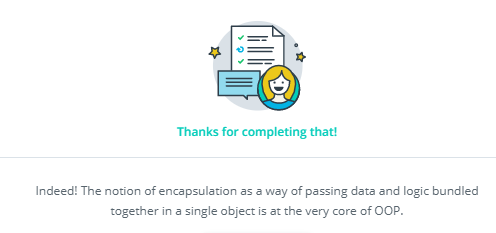
1. QUIZ: Encapsulation in c++

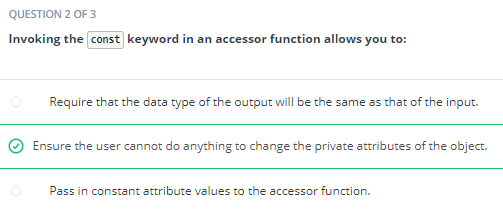


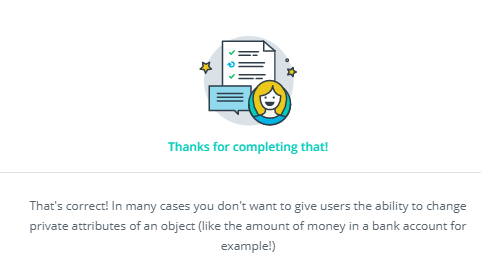


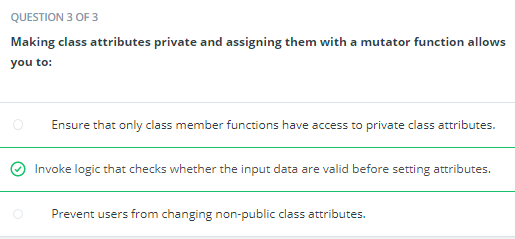


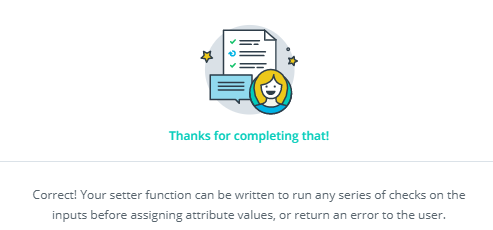












1. Bjarne on Abstraction

<https://youtu.be/eUf7QBJNIFc>

1. Abstraction

<https://youtu.be/C0MJyP0dOhU>

# Abstraction

Abstraction refers to the separation of a class's interface from the details of its implementation. The interface provides a way to interact with an object, while hiding the details and implementation of how the class works.

## Example

The String() function within this Date class is an example of abstraction.

**class** Date {

**public**:

...

std::string **String**() **const**;

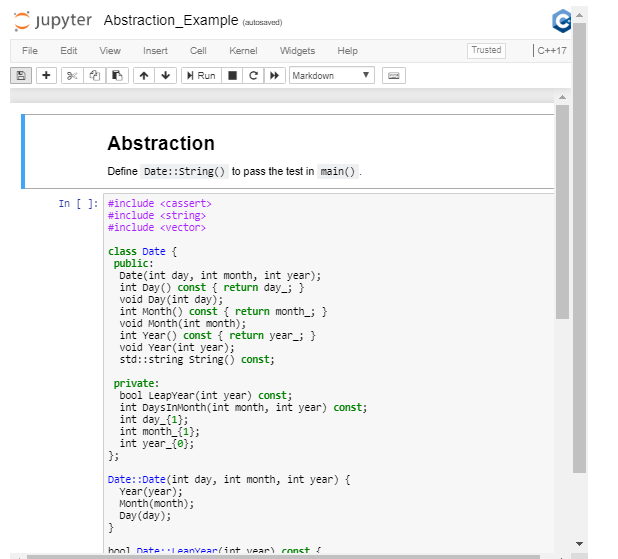
...

};

The user is able to interact with the Date class through the String() function, but the user does not need to know about the implementation of either Date or String().

For example, the user does not know, or need to know, that this object internally contains three int member variables. The user can just call the String() method to get data.

If the designer of this class ever decides to change how the data is stored internally -- using a vector of ints instead of three separate ints, for example -- the user of the Date class will not need to know.



1. Exercise: Sphere Class

<https://youtu.be/S5iKMpA-gFE>

**Exercise: Sphere Class**

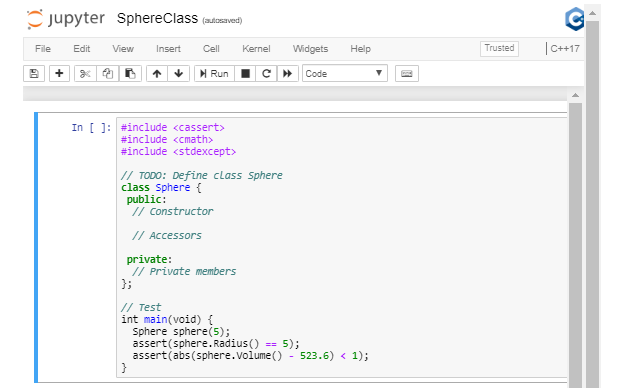
In this exercise you will practice abstraction by creating a class which represents a sphere.

Declare:

1. A constructor function that takes the radius as an argument
2. A member function that returns the [**volume**](https://www.google.com/search?q=sphere+volume)

**Directions**

1. Define a class called Sphere.
2. Add one private member variable: radius.
3. Define a constructor to initialize the radius.
4. Define an accessor method that returns the radius.
5. Define a member function to return the volume of the sphere.
6. Write a main() function to initialize an object of type Sphere.



1. Exercise: Private Method

<https://youtu.be/Z75_sEnapD4>

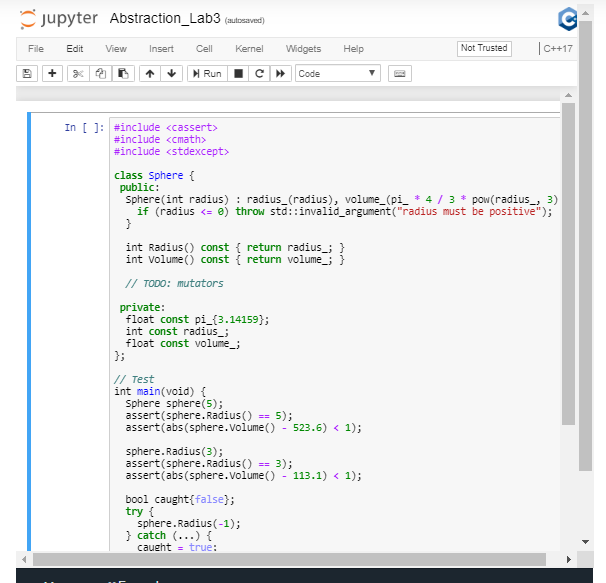
**Exercise: Private Method**

Abstraction is used to expose only relevant information to the user. By hiding implementation details, we give ourselves flexibility to modify how the program works. In this example, you'll practice abstracting implementation details from the user.

**Directions**

In this exercise, you'll update the class Sphere so that it becomes possible to change the radius of a sphere after it has been initialized. In order to do this, you'll move the two [**class invariants**](https://en.wikipedia.org/wiki/Class_invariant) into private member functions.

1. Move the range-check on radius\_ into a private member function.
2. Move the volume\_ calculation, which depends on the value of radius\_ into the same private member function.
3. Verify that the class still functions correctly.
4. Add a mutator method to change the radius of an existing Sphere.
5. Verify that the mutator method successfully updates both the radius and the volume.

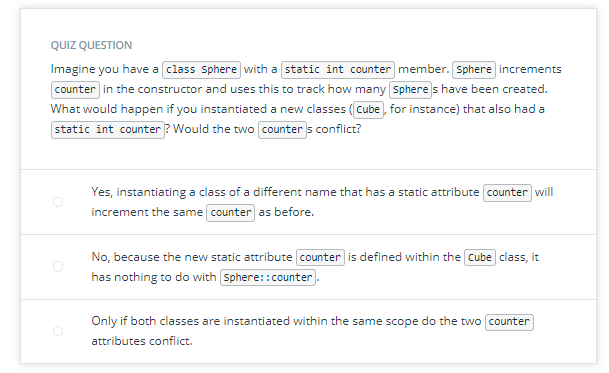


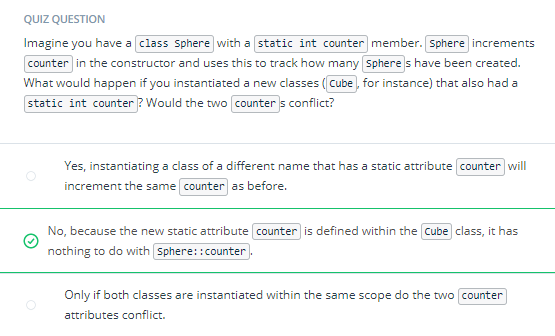
1. Exercise: Static Members

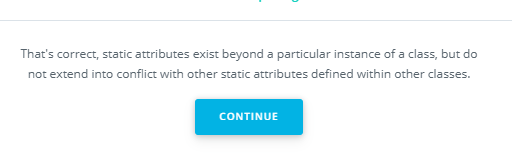
<https://youtu.be/7fBkcIL6d8k>

## Static Members

Class members can be declared static, which means that the member belongs to the entire class, instead of to a specific instance of the class. More specifically, a static member is created only once and then shared by all instances (i.e. objects) of the class. That means that if the static member gets changed, either by a user of the class or within a member function of the class itself, then all members of the class will see that change the next time they access the static member.







## Implementation

static members are **declared** within their class (often in a header file) but in most cases they must be **defined** within the global scope. That's because memory is allocated for static variables immediately when the program begins, at the same time any global variables are initialized.

Here is an example:

**#include <cassert>**

**class** Foo {

**public**:

**static** **int** count;

Foo() { Foo::count += 1; }

};

**int** Foo::count{0};

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

Foo f{};

assert(Foo::count == 1);

}

An exception to the global definition of static members is if such members can be marked as **[constexpr](https://en.cppreference.com/w/cpp/language/constexpr" \t "_blank)**. In that case, the static member variable can be both declared and defined within the class definition:

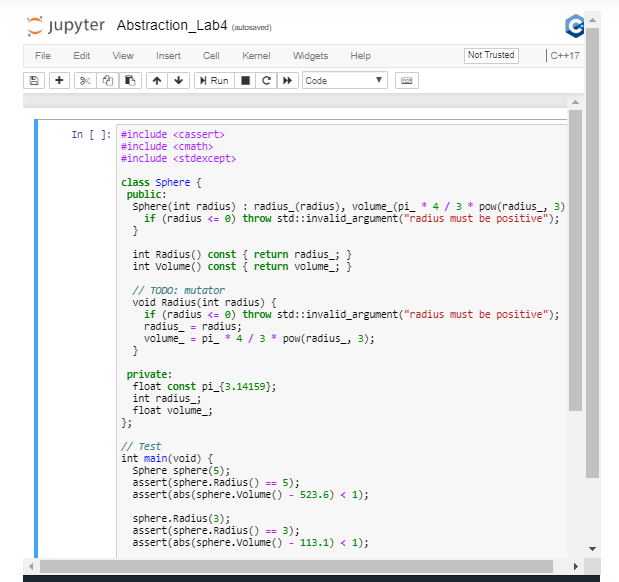
**struct** Kilometer {

**static** **constexpr** **int** meters{1000};

};

## Exercise: Pi

class Sphere has a member const double pi. Experiment with specifying pi to be const, constexpr, and static. Which specifications work and which break? Do you understand why?



1. Exercise: Static Methods

**Exercise: Static Method**

In addition to static member variables, C++ supports static member functions (or "methods"). Just like static member variables, static member functions are instance-independent: they belong to the class, not to any particular instance of the class.

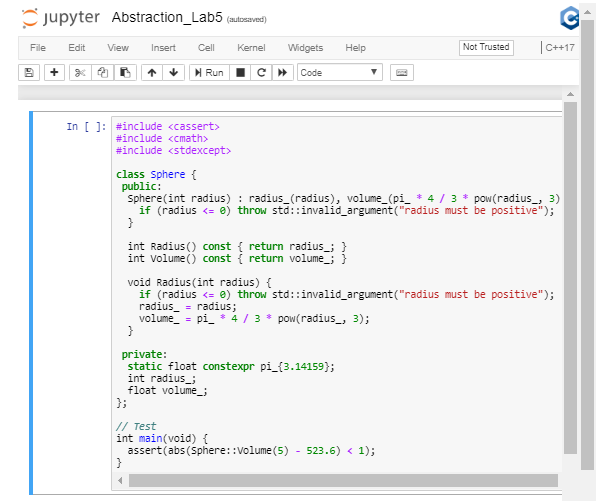
One corollary to this is that we can method invoke a static member function *without ever creating an instance of the class*.

You will try just that in this exercise.

**Instructions**

1. Refactor class Sphere to move the volume calculation into a static function.
2. Verify that the class still functions as intended.
3. Make that static function public.
4. Call that static function directly from main() to calculate the hypothetical volume of a sphere you have not yet instantiated.

<https://youtu.be/8ezbkN76msY>



1. Bjarne on Solving Methods

<https://youtu.be/Jo1NTa-krEE>