3. Object-Oriented Programming

CORE CURRICULUM

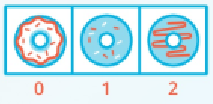
Assessment: System Monitor due by Jun 23rd | 32% VIEWED | PROJECTS

LESSON 1

Welcome

CONTINUE →

25% — 45 minutes left



LESSON 2

Intro to OOP

In this lesson, you will explore some of the basic object oriented functionality of the C++ language.

CONTINUE →

33% — 1 hour 20 minutes left

LESSON 3

Advanced OOP

In this lesson, we'll get into some of the more advanced topics in object oriented programming, including inheritance, polymorphism and templates.

CONTINUE →

48% — 1 hour 3 minutes left

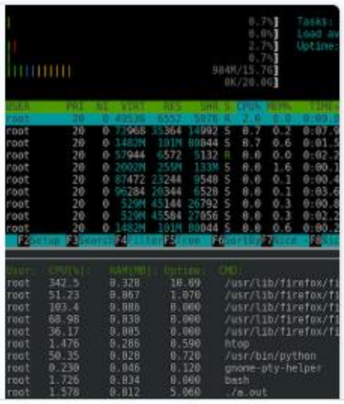
PROJECT

Project: System Monitor

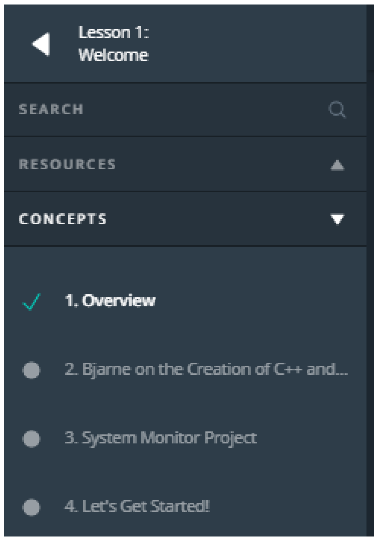
Time to build the project! In this lesson, you'll get the tools you need to build the project for this course, a system monitor application similar to htop!

CONTINUE →

STARTED — Due in 1 month



```
                                   0.7%]   Tasks:
                                   0.0%]   Load av
                                   2.7%]   Uptime:
                                   0.7%]
||||||||||                   984M/15.7G]
                               0K/20.0G]
USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+
root       20   0 43528  6552  5076 R  2.0  0.0  0:00.0
root       20   0 72968 35364 14992 S  0.7  0.2  0:07.9
root       20   0 1482M  101M 80044 S  0.7  0.6  0:01.5
root       20   0 57944  6572  5132 R  0.0  0.0  0:02.2
root       20   0 2002M  255M  133M S  0.0  1.6  0:00.1
root       20   0 87472 23244  9548 S  0.0  0.1  0:00.4
root       20   0 96284 20344  6528 S  0.0  0.1  0:03.6
root       20   0  529M 45144 26792 S  0.0  0.3  0:00.8
root       20   0  529M 45504 27056 S  0.0  0.3  0:02.2
root       20   0 1482M  101M 80044 S  0.0  0.6  0:00.2
 F2Setup F3Search F4Filter F5Tree  F6SortBy F7Nice  F8

User:  CPU[%]:   RAM[MB]:  Uptime:   CMD:
root   342.5     0.328     10.09     /usr/lib/firefox/fi
root   51.23     0.067     1.070     /usr/lib/firefox/fi
root   103.4     0.086     0.000     /usr/lib/firefox/fi
root   68.98     0.038     0.000     /usr/lib/firefox/fi
root   36.17     0.005     0.000     /usr/lib/firefox/fi
root   1.476     0.286     0.590     htop
root   50.35     0.028     0.720     /usr/bin/python
root   0.230     0.046     0.120     gnome-pty-helper
root   1.726     0.034     0.000     bash
root   1.578     0.012     5.060     ./a.out
```


Lesson 1:
Welcome

SEARCH

RESOURCES

CONCEPTS

✓ 1. Overview

● 2. Bjarne on the Creation of C++ and...

● 3. System Monitor Project

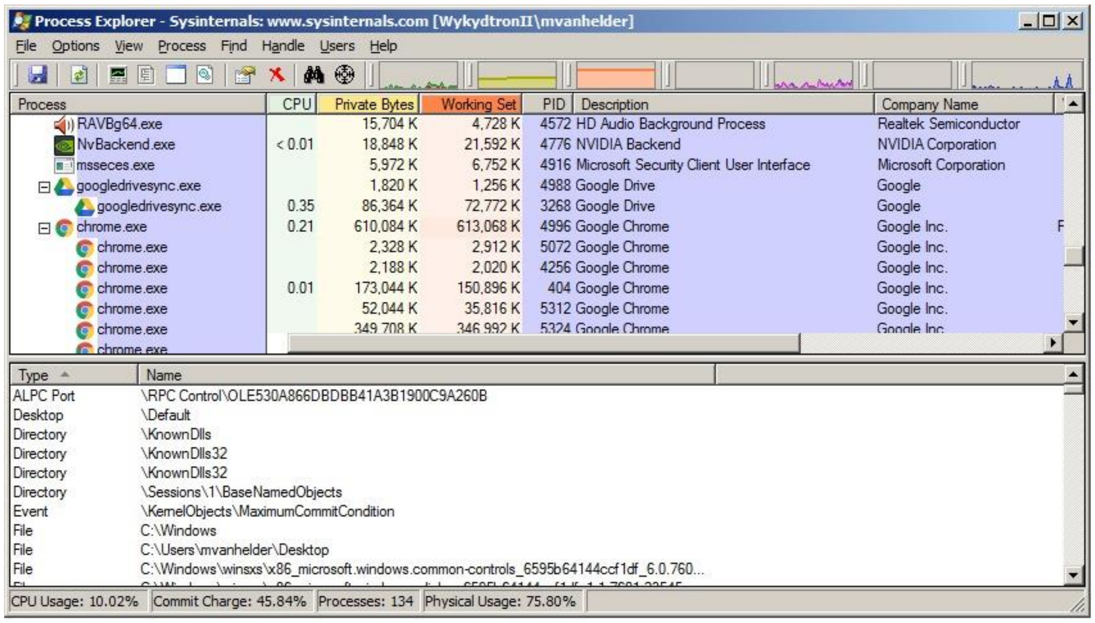
● 4. Let's Get Started!

1) Overview
   https://youtu.be/8V271hq1dfE

2) Bjarne on the Creation of C++ and classes
   https://youtu.be/pqPvz33zVkE

3) System Monitor Project
   https://youtu.be/pUoA2pFOxbI

If you use Windows, then you might be familiar with the **Process Explorer** or **Process Monitor** applications:


Process Explorer - Sysinternals: www.sysinternals.com [WykydtronII\mvanhelder]
File Options View Process Find Handle Users Help

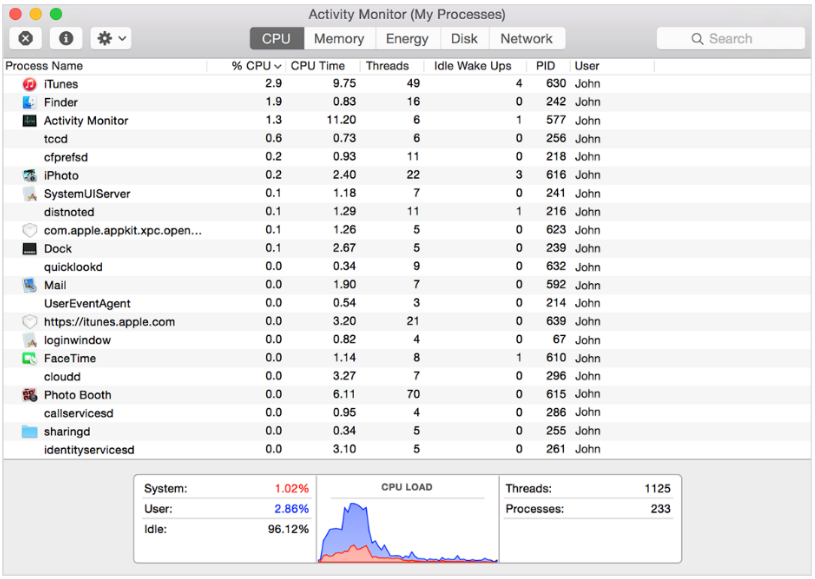
Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
RAVBg64.exe		15,704 K	4,728 K	4572	HD Audio Background Process	Realtek Semiconductor
NvBackend.exe	< 0.01	18,848 K	21,592 K	4776	NVIDIA Backend	NVIDIA Corporation
msseces.exe		5,972 K	6,752 K	4916	Microsoft Security Client User Interface	Microsoft Corporation
googledrivesync.exe		1,820 K	1,256 K	4988	Google Drive	Google
googledrivesync.exe	0.35	86,364 K	72,772 K	3268	Google Drive	Google
chrome.exe	0.21	610,084 K	613,068 K	4996	Google Chrome	Google Inc.
chrome.exe		2,328 K	2,912 K	5072	Google Chrome	Google Inc.
chrome.exe		2,188 K	2,020 K	4256	Google Chrome	Google Inc.
chrome.exe	0.01	173,044 K	150,896 K	404	Google Chrome	Google Inc.
chrome.exe		52,044 K	35,816 K	5312	Google Chrome	Google Inc.
chrome.exe		349,708 K	346,992 K	5324	Google Chrome	Google Inc.
chrome.exe						

Type	Name
ALPC Port	\RPC Control\OLE530A866DBDBB41A3B1900C9A260B
Desktop	\Default
Directory	\KnownDlls
Directory	\KnownDlls32
Directory	\KnownDlls32
Directory	\Sessions\1\BaseNamedObjects
Event	\KernelObjects\MaximumCommitCondition
File	C:\Windows
File	C:\Users\mvanhelder\Desktop
File	C:\Windows\winsxs\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.760...

CPU Usage: 10.02% | Commit Charge: 45.84% | Processes: 134 | Physical Usage: 75.80%

Source: PC World

On the other hand, if you are a Mac user, you've probably seen the **Activity Monitor** before:



Activity Monitor (My Processes)

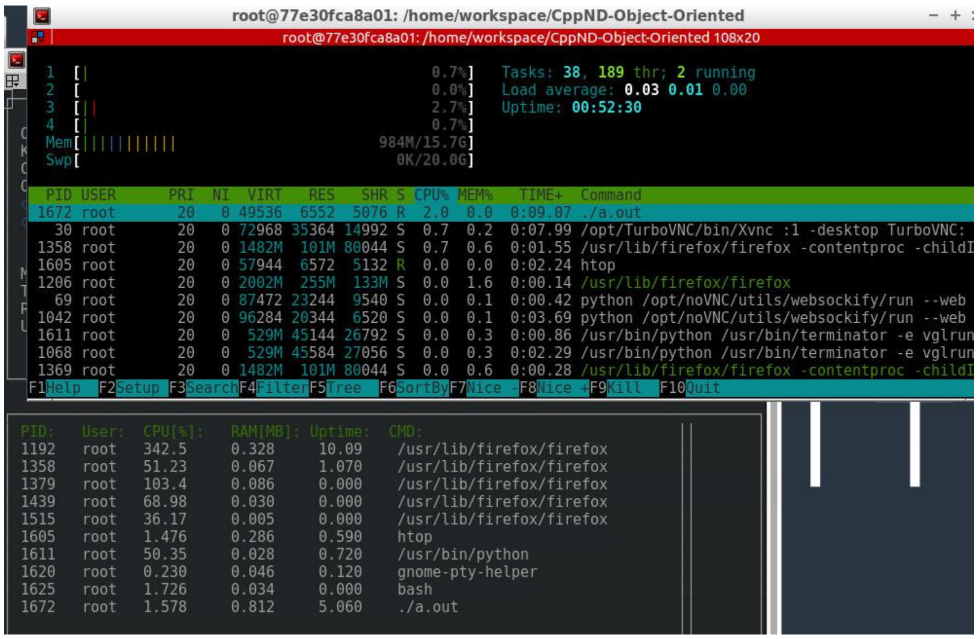
Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
iTunes	2.9	9.75	49	4	630	John
Finder	1.9	0.83	16	0	242	John
Activity Monitor	1.3	11.20	6	1	577	John
tccd	0.6	0.73	6	0	256	John
cfprefsd	0.2	0.93	11	0	218	John
iPhoto	0.2	2.40	22	3	616	John
SystemUIServer	0.1	1.18	7	0	241	John
distnoted	0.1	1.29	11	1	216	John
com.apple.appkit.xpc.open...	0.1	1.26	5	0	623	John
Dock	0.1	2.67	5	0	239	John
quicklookd	0.0	0.34	9	0	632	John
Mail	0.0	1.90	7	0	592	John
UserEventAgent	0.0	0.54	3	0	214	John
https://itunes.apple.com	0.0	3.20	21	0	639	John
loginwindow	0.0	0.82	4	0	67	John
FaceTime	0.0	1.14	8	1	610	John
cloudd	0.0	3.27	7	0	296	John
Photo Booth	0.0	6.11	70	0	615	John
callservicesd	0.0	0.95	4	0	286	John
sharingd	0.0	0.34	5	0	255	John
identityservicesd	0.0	3.10	5	0	261	John

System:	1.02%
User:	2.86%
Idle:	96.12%

CPU LOAD

Threads:	1125
Processes:	233

And finally, if you are a Linux user, you may have used **top** or **htop** 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:

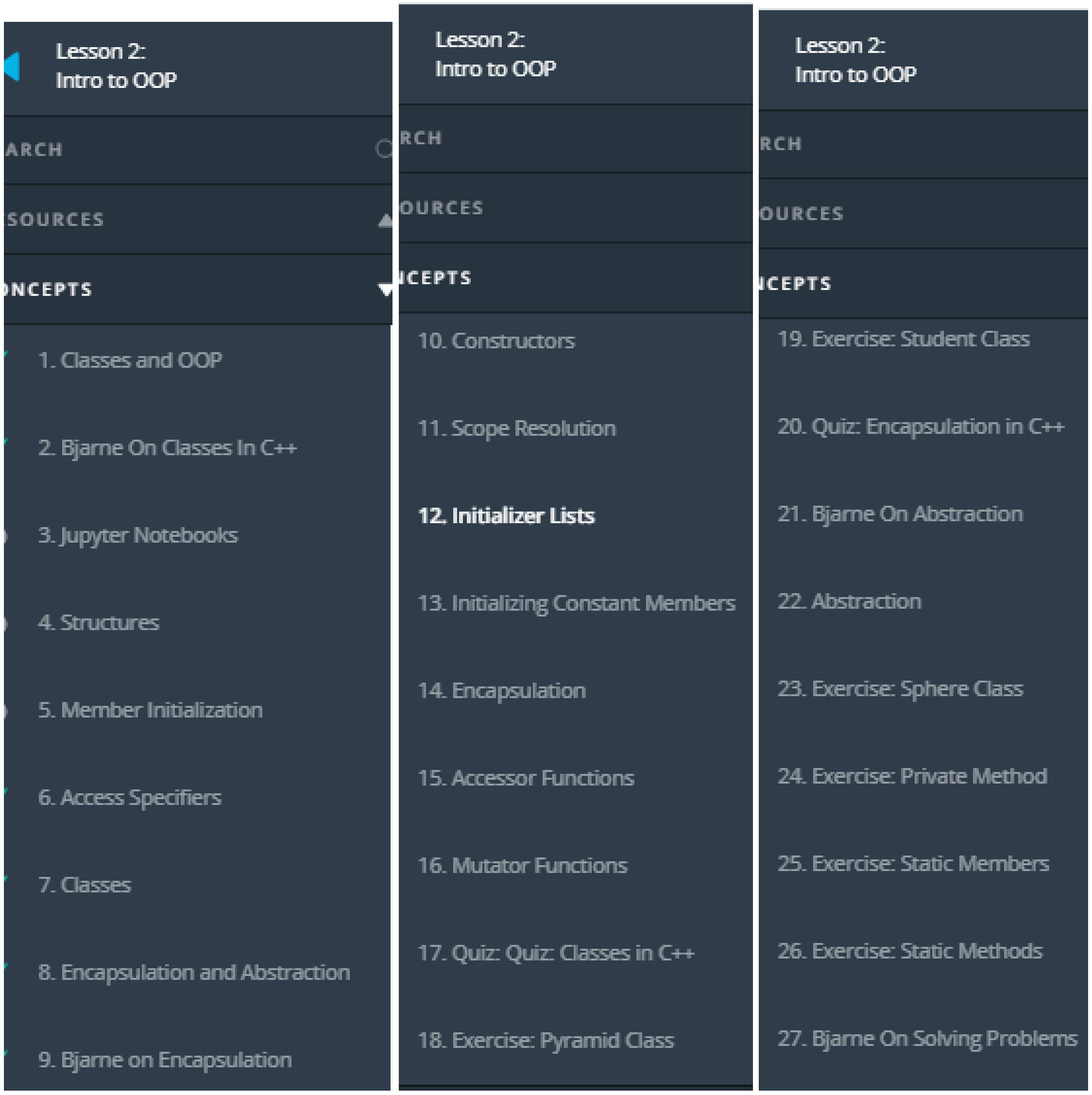


root@77e30fca8a01: /home/workspace/CppND-Object-Oriented

```
root@77e30fca8a01: /home/workspace/CppND-Object-Oriented 108x20
  1  [|                                    0.7%]   Tasks: 38, 189 thr; 2 running
  2  [                                     0.0%]   Load average: 0.03 0.01 0.00
  3  [||                                   2.7%]   Uptime: 00:52:30
  4  [|                                    0.7%]
  Mem[||||||||||||                     984M/15.7G]
  Swp[                                   0K/20.0G]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 1672 root       20   0 49536  6552  5076 R  2.0  0.0  0:09.07 ./a.out
   30 root       20   0 72968 35364 14992 S  0.7  0.2  0:07.99 /opt/TurboVNC/bin/Xvnc :1 -desktop TurboVNC:
 1358 root       20   0 1482M  101M 80044 S  0.7  0.6  0:01.55 /usr/lib/firefox/firefox -contentproc -childI
 1605 root       20   0 57944  6572  5132 R  0.0  0.0  0:02.24 htop
 1206 root       20   0 2002M  255M  133M S  0.0  1.6  0:00.14 /usr/lib/firefox/firefox
   69 root       20   0 87472 23244  9540 S  0.0  0.1  0:00.42 python /opt/noVNC/utils/websockify/run --web
 1042 root       20   0 96284 20344  6520 S  0.0  0.1  0:03.69 python /opt/noVNC/utils/websockify/run --web
 1611 root       20   0  529M 45144 26792 S  0.0  0.3  0:00.86 /usr/bin/python /usr/bin/terminator -e vglrun
 1068 root       20   0  529M 45584 27056 S  0.0  0.3  0:02.29 /usr/bin/python /usr/bin/terminator -e vglrun
 1369 root       20   0 1482M  101M 80044 S  0.0  0.6  0:00.28 /usr/lib/firefox/firefox -contentproc -childI
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

```
PID:    User:   CPU[%]:    RAM[MB]: Uptime:   CMD:
1192    root    342.5      0.328     10.09    /usr/lib/firefox/firefox
1358    root    51.23      0.067     1.070    /usr/lib/firefox/firefox
1379    root    103.4      0.086     0.000    /usr/lib/firefox/firefox
1439    root    68.98      0.030     0.000    /usr/lib/firefox/firefox
1515    root    36.17      0.005     0.000    /usr/lib/firefox/firefox
1605    root    1.476      0.286     0.590    htop
1611    root    50.35      0.028     0.720    /usr/bin/python
1620    root    0.230      0.046     0.120    gnome-pty-helper
1625    root    1.726      0.034     0.000    bash
1672    root    1.578      0.812     5.060    ./a.out
```

4) Let's Get started
   https://youtu.be/_uuC7FphqcA



Lesson 2:
Intro to OOP

ARCH

SOURCES

NCEPTS

1. Classes and OOP
2. Bjarne On Classes In C++
3. Jupyter Notebooks
4. Structures
5. Member Initialization
6. Access Specifiers
7. Classes
8. Encapsulation and Abstraction
9. Bjarne on Encapsulation

Lesson 2:
Intro to OOP

RCH

OURCES

CEPTS

10. Constructors
11. Scope Resolution
12. Initializer Lists
13. Initializing Constant Members
14. Encapsulation
15. Accessor Functions
16. Mutator Functions
17. Quiz: Quiz: Classes in C++
18. Exercise: Pyramid Class

Lesson 2:
Intro to OOP

RCH

OURCES

CEPTS

19. Exercise: Student Class
20. Quiz: Encapsulation in C++
21. Bjarne On Abstraction
22. Abstraction
23. Exercise: Sphere Class
24. Exercise: Private Method
25. Exercise: Static Members
26. Exercise: Static Methods
27. Bjarne On Solving Problems

1) Classes and OOP
   https://youtu.be/0Z9vdXNyEVc

2) Bjarne on classes in c++
   https://youtu.be/leLjGXFcycE

3) Jupyter Notebooks

Jupyter Notebooks

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

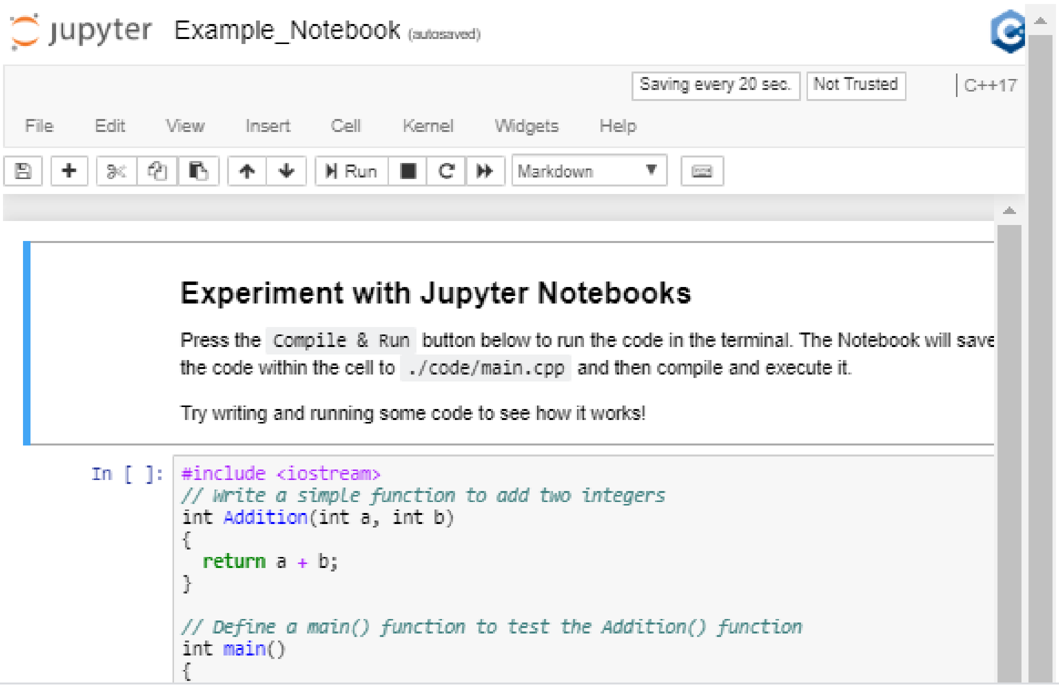
In this lesson, you will save, compile, and run executables over and over. To make your life simpler, we've set things up so you can save, compile and run with the click of a single `Compile & Run` button in the Jupyter Notebooks.

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.



Experiment with Jupyter Notebooks

Press the `Compile & Run` button below to run the code in the terminal. The Notebook will save the code within the cell to `./code/main.cpp` and then compile and execute it.

Try writing and running some code to see how it works!

In [ ]:
```
#include <iostream>
// Write a simple function to add two integers
int Addition(int a, int b)
{
  return a + b;
}

// Define a main() function to test the Addition() function
int main()
{
```

4) 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](#).

```
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**, such as `int` and `float`. These fundamental types are sometimes called **"primitives"**.
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.

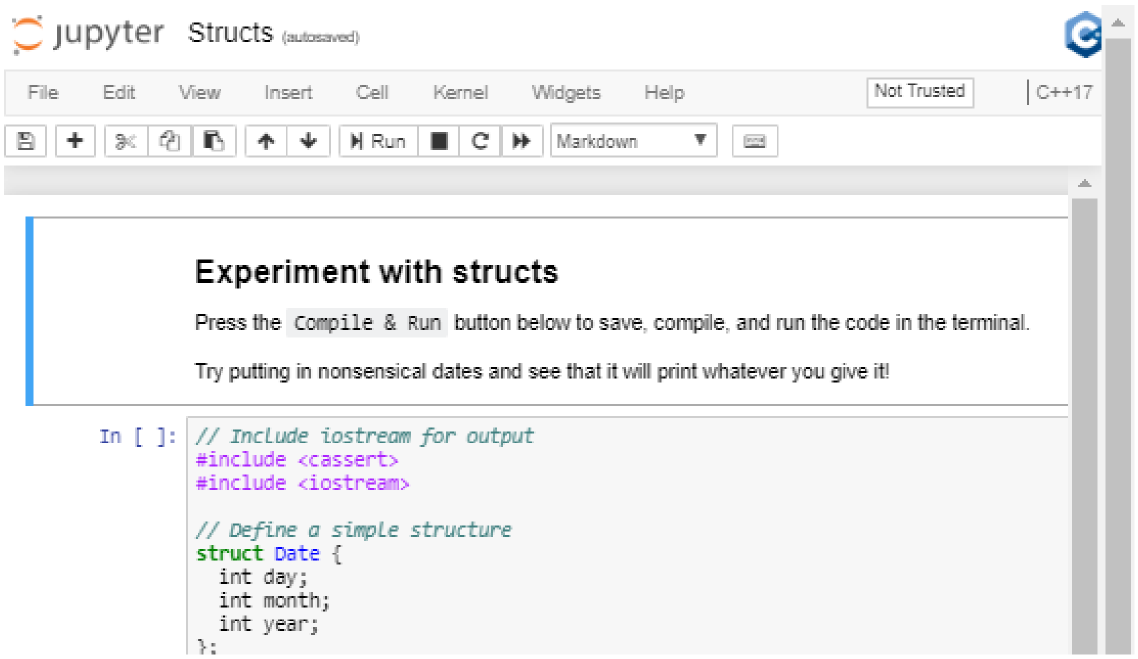
For example, C++ does not have a fundamental type for storing a date. (The Standard Library does include types related to **time**, which can be converted to dates.)
A programmer might desire to create a type to store a date.

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;
```


jupyter Structs (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
C++17
Run
Markdown

Experiment with structs

Press the `Compile & Run` button below to save, compile, and run the code in the terminal.

Try putting in nonsensical dates and see that it will print whatever you give it!

In [ ]:
```
// Include iostream for output
#include <cassert>
#include <iostream>

// Define a simple structure
struct Date {
  int day;
  int month;
  int year;
};
```

5) Member Initialization

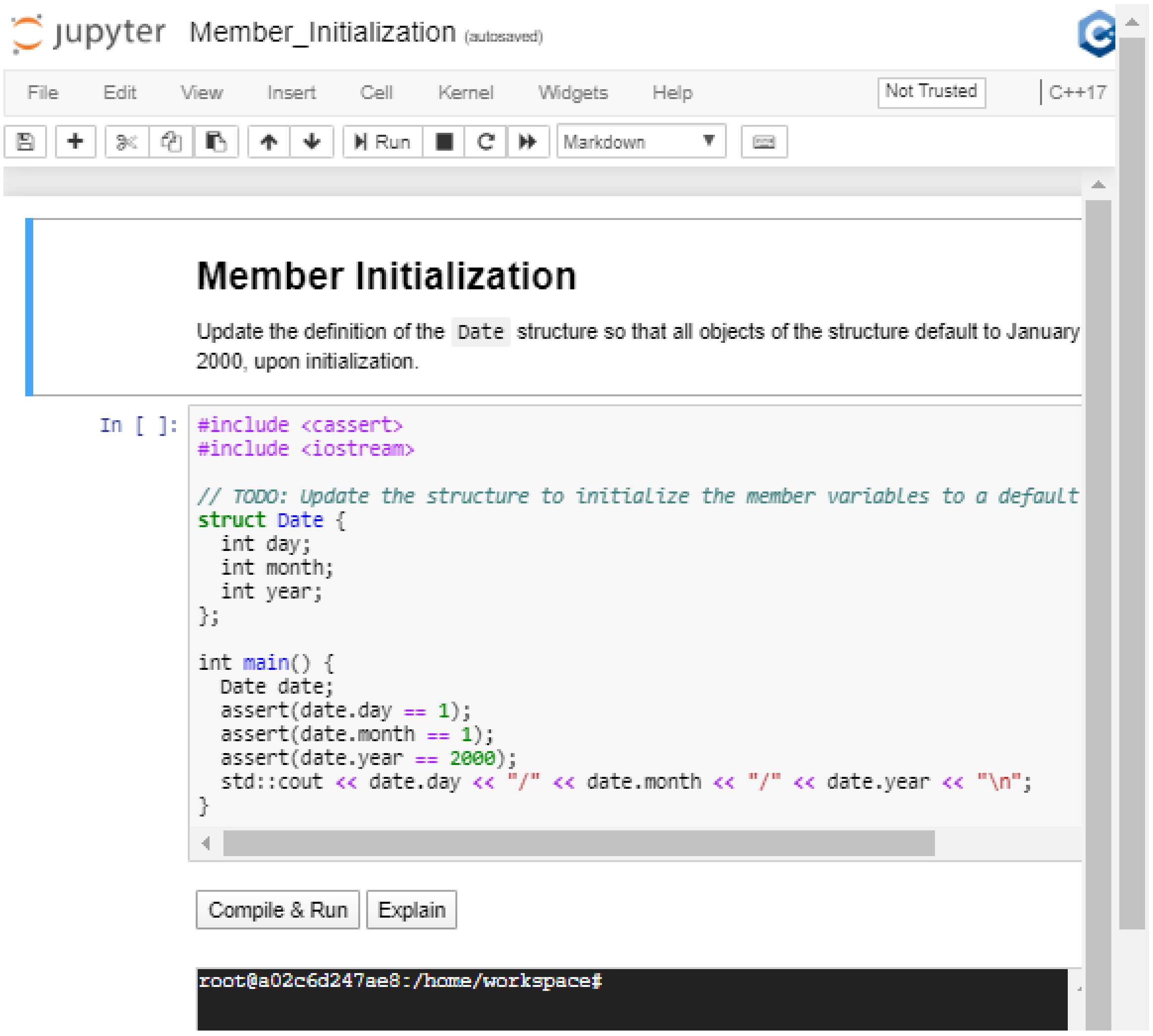
Member Initialization

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.


jupyter Member_Initialization (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
C++17
Run
Markdown

Member Initialization

Update the definition of the `Date` structure so that all objects of the structure default to January 2000, upon initialization.

In [ ]:
```
#include <cassert>
#include <iostream>

// TODO: Update the structure to initialize the member variables to a default
struct Date {
  int day;
  int month;
  int year;
};

int main() {
  Date date;
  assert(date.day == 1);
  assert(date.month == 1);
  assert(date.year == 2000);
  std::cout << date.day << "/" << date.month << "/" << date.year << "\n";
}
```

```
root@a02c6d247ae8:/home/workspace#
```

6) 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).

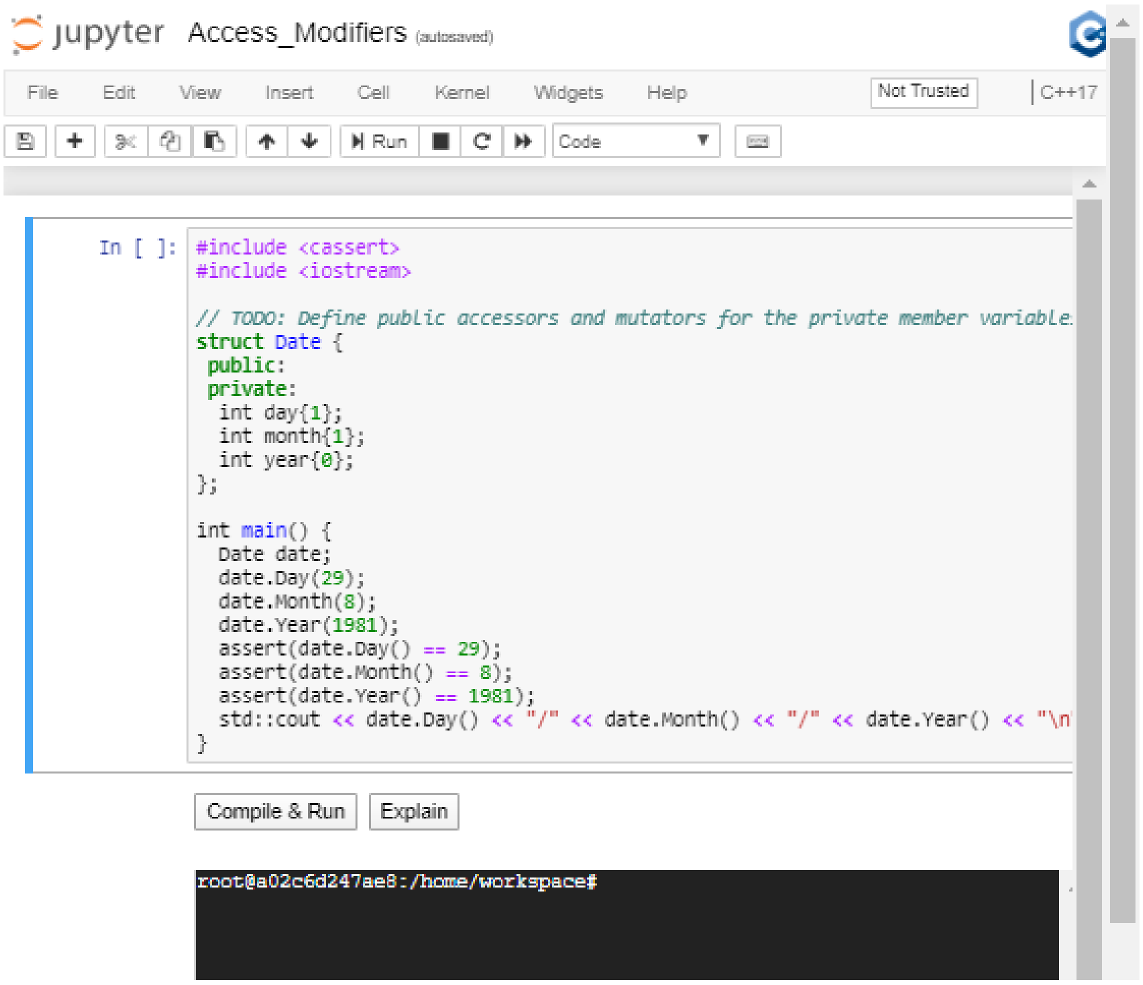
There is a third access modifier called `protected`, which implies that members are accessible from other member functions of the same class (or from their "friends"), and also from members of their derived classes. We'll also discuss about derived classes later, when we learn about inheritance.

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!


jupyter Access_Modifiers (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
C++17
Run
Code
Compile & Run
Explain

```
In [ ]: #include <cassert>
        #include <iostream>

        // TODO: Define public accessors and mutators for the private member variable
        struct Date {
         public:
         private:
          int day{1};
          int month{1};
          int year{0};
        };

        int main() {
          Date date;
          date.Day(29);
          date.Month(8);
          date.Year(1981);
          assert(date.Day() == 29);
          assert(date.Month() == 8);
          assert(date.Year() == 1981);
          std::cout << date.Day() << "/" << date.Month() << "/" << date.Year() << "\n
        }
```

```
root@a02c6d247ae8:/home/workspace#
```

Avoid Trivial Getters And Setters

Sometimes accessors are not necessary, or even advisable. The **C++ Core Guidelines** 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
};
```

7) 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"**.

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};
};
```

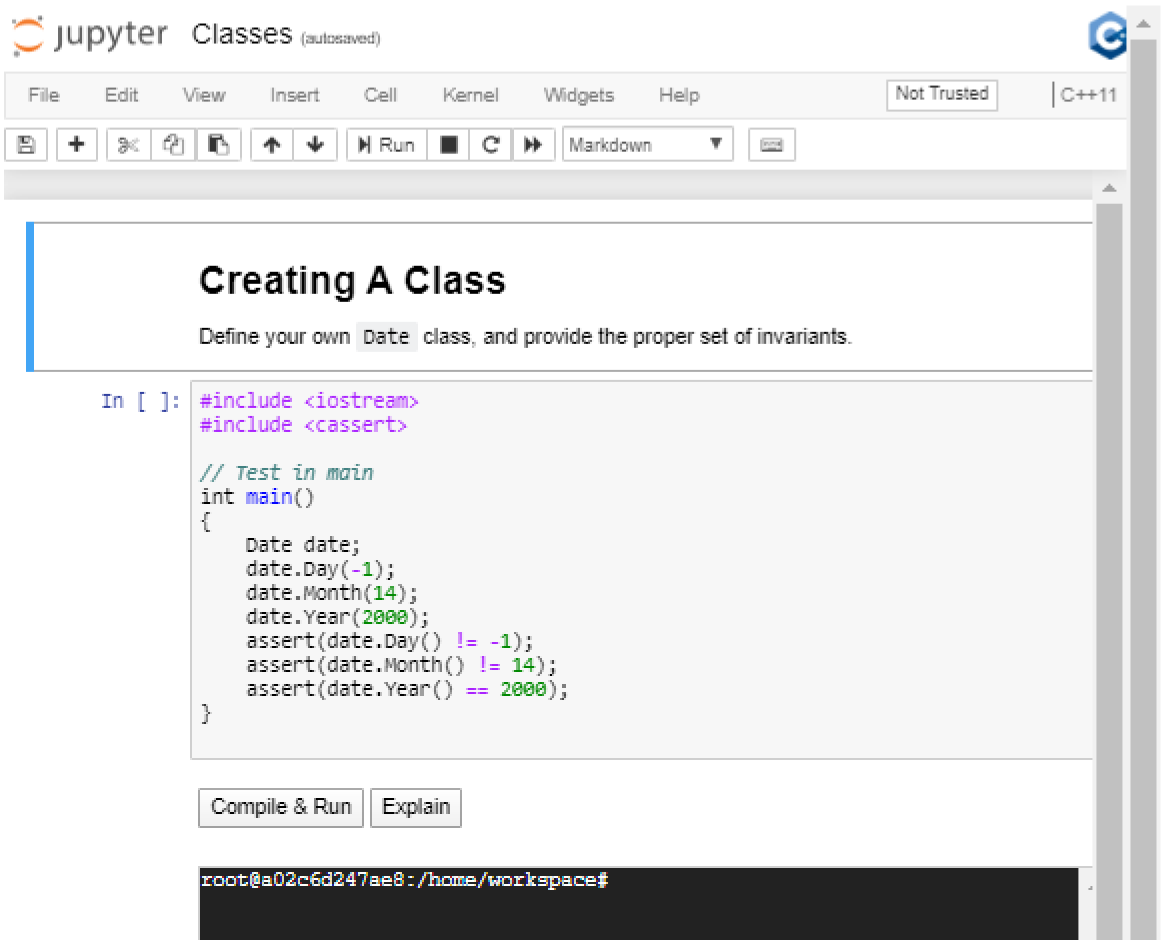
`Date` Invariants

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.


jupyter Classes (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
C++11
Run
Markdown
Compile & Run
Explain

Creating A Class

Define your own `Date` class, and provide the proper set of invariants.

In [ ]:
```
#include <iostream>
#include <cassert>

// Test in main
int main()
{
    Date date;
    date.Day(-1);
    date.Month(14);
    date.Year(2000);
    assert(date.Day() != -1);
    assert(date.Month() != 14);
    assert(date.Year() == 2000);
}
```

```
root@a02c6d247ae8:/home/workspace#
```

8) Encapsulation and Abstraction
   https://youtu.be/KXojXgyW-O8

9) Bjarne on Encapsulation
   https://youtu.be/M27atfK73_s

10) Constructors
    https://youtu.be/zw5fqCmxD8o

Constructors

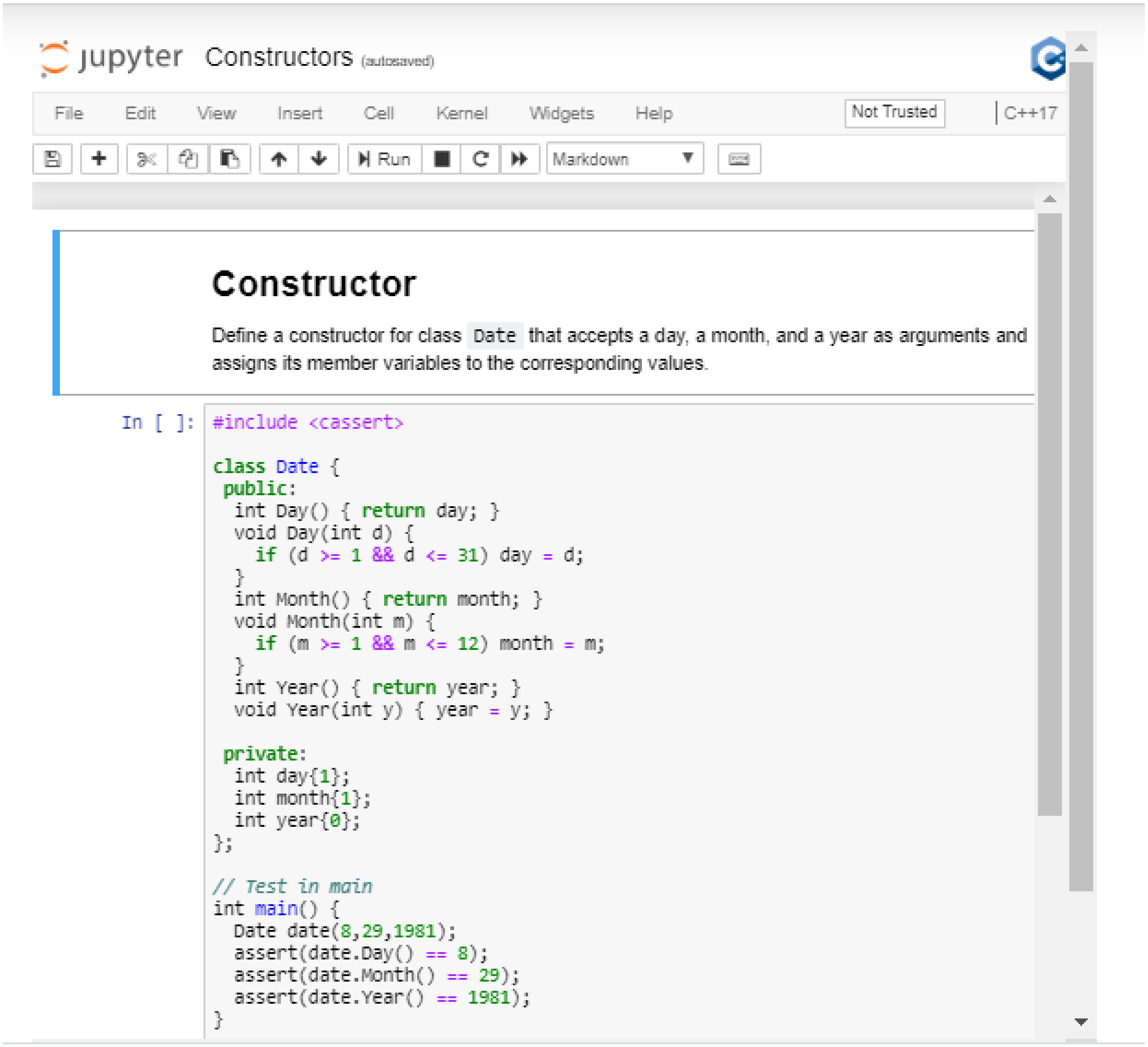
Constructors are member functions of a class or struct that initialize an object. The Core Guidelines **define a constructor**) as:

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)`.


jupyter Constructors (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
C++17
Run
Markdown

Constructor

Define a constructor for class `Date` that accepts a day, a month, and a year as arguments and assigns its member variables to the corresponding values.

In [ ]:

```cpp
#include <cassert>

class Date {
 public:
  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};
};

// Test in main
int main() {
  Date date(8,29,1981);
  assert(date.Day() == 8);
  assert(date.Month() == 29);
  assert(date.Year() == 1981);
}
```

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:

```cpp
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**. **The compiler will define a default constructor**, which accepts no arguments, for any class or structure that does not contain an explicitly-defined constructor.

11) Scope Resolution
    https://youtu.be/U2ItvJEwuHQ

Scope Resolution

C++ allows different **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"**.

Scope Resultion Operator

`::` is the **scope resolution operator**. 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

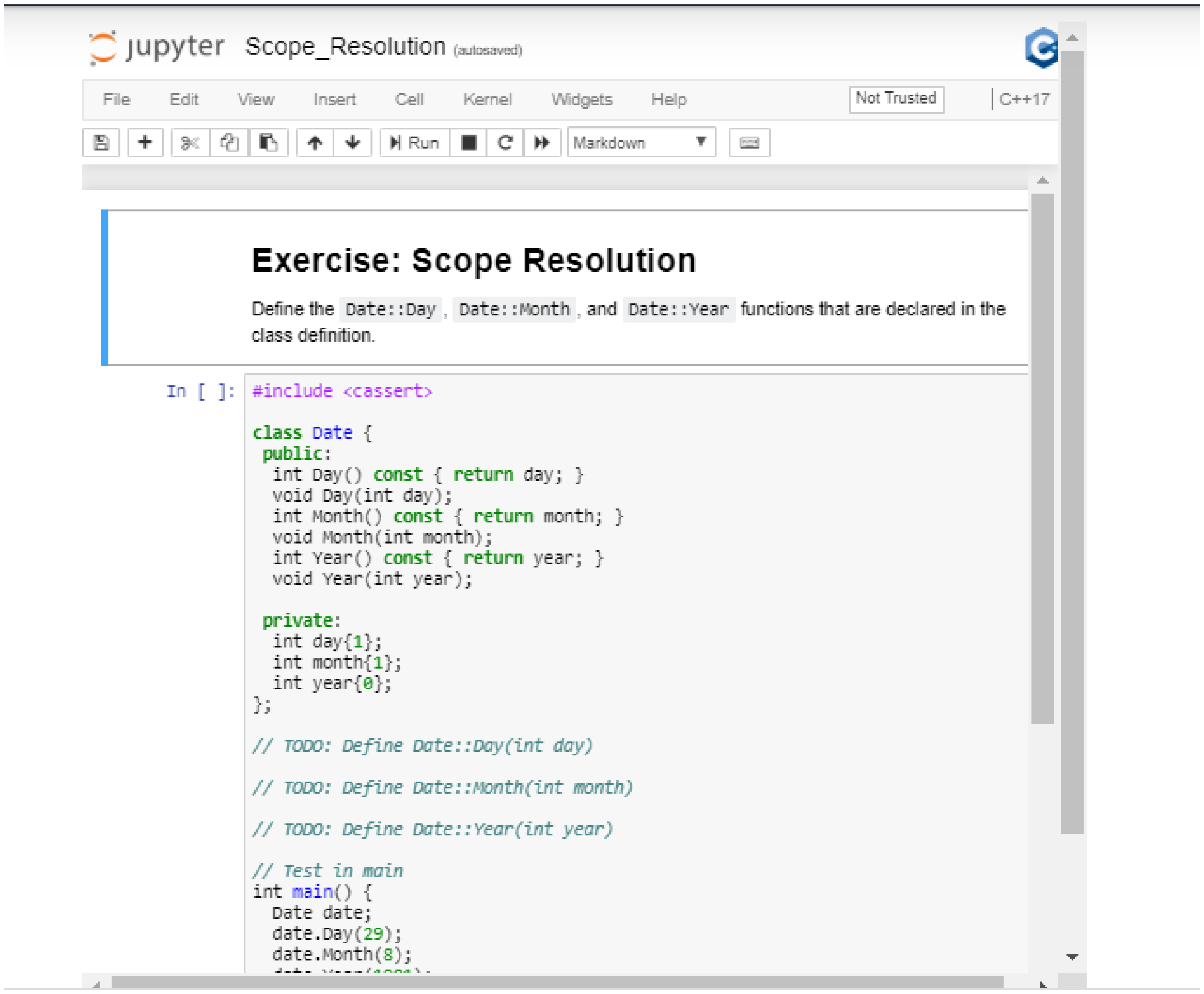
Namespaces allow programmers to group logically related variables and functions together. Namespaces also help to avoid conflicts between to variables that have the same name in different parts of a program.

```
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.
Classes like **`std::vector`** and functions like **`std::sort`** are defined within the `std` namespace


jupyter Scope_Resolution (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
C++17
Run
Markdown

Exercise: Scope Resolution

Define the `Date::Day`, `Date::Month`, and `Date::Year` functions that are declared in the class definition.

In [ ]:
```
#include <cassert>

class Date {
 public:
  int Day() const { return day; }
  void Day(int day);
  int Month() const { return month; }
  void Month(int month);
  int Year() const { return year; }
  void Year(int year);

 private:
  int day{1};
  int month{1};
  int year{0};
};

// TODO: Define Date::Day(int day)

// TODO: Define Date::Month(int month)

// TODO: Define Date::Year(int year)

// Test in main
int main() {
  Date date;
  date.Day(29);
  date.Month(8);
```

12) Initializer Lists
https://youtu.be/cqKuYu1oiow

Initializer Lists

Initializer lists 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.

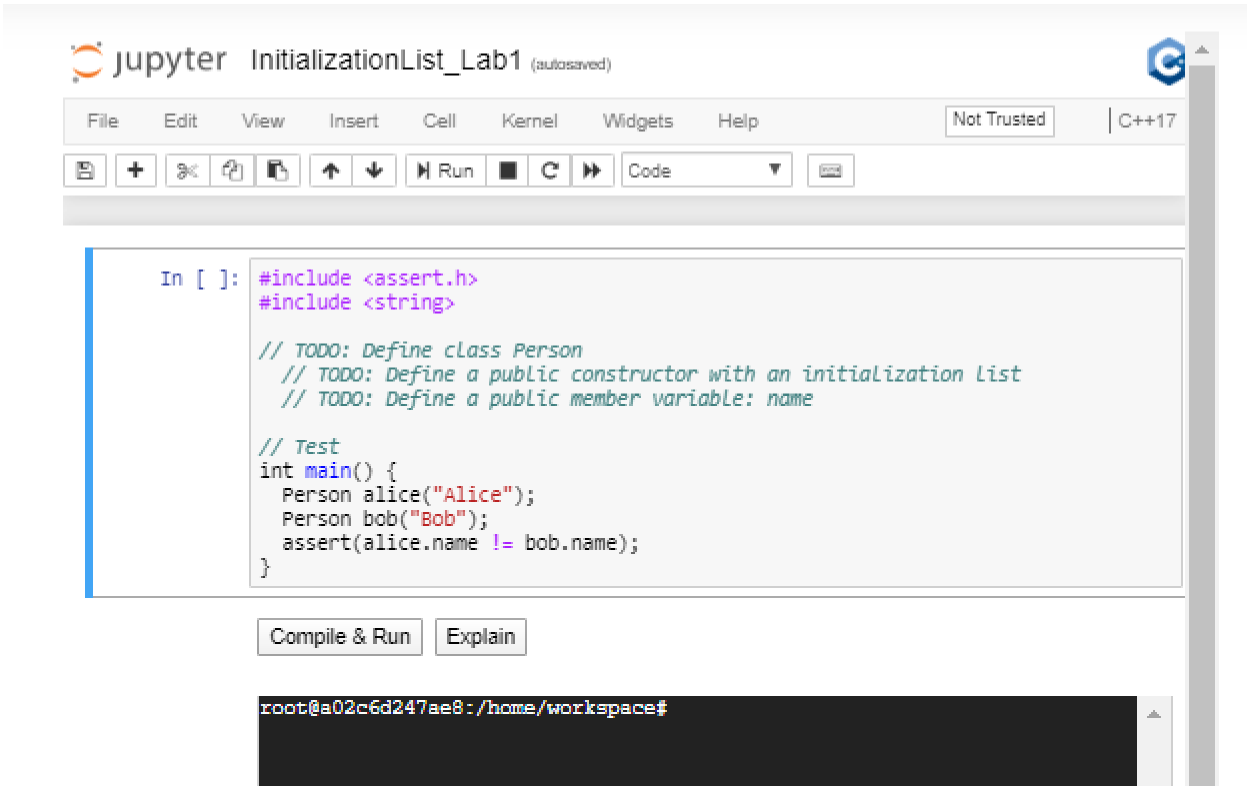
```
Date::Date(int day, int month, int year) : year_(y) {
  Day(day);
  Month(month);
}
```

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**. 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.


jupyter InitializationList_Lab1 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Not Trusted C++17
Run Code
Compile & Run Explain

```
In [ ]: #include <assert.h>
        #include <string>

        // TODO: Define class Person
          // TODO: Define a public constructor with an initialization list
          // TODO: Define a public member variable: name

        // Test
        int main() {
          Person alice("Alice");
          Person bob("Bob");
          assert(alice.name != bob.name);
        }
```

```
root@a02c6d247ae8:/home/workspace#
```

13) Initializing Constant Members
https://youtu.be/Ms4Li58ZvwA

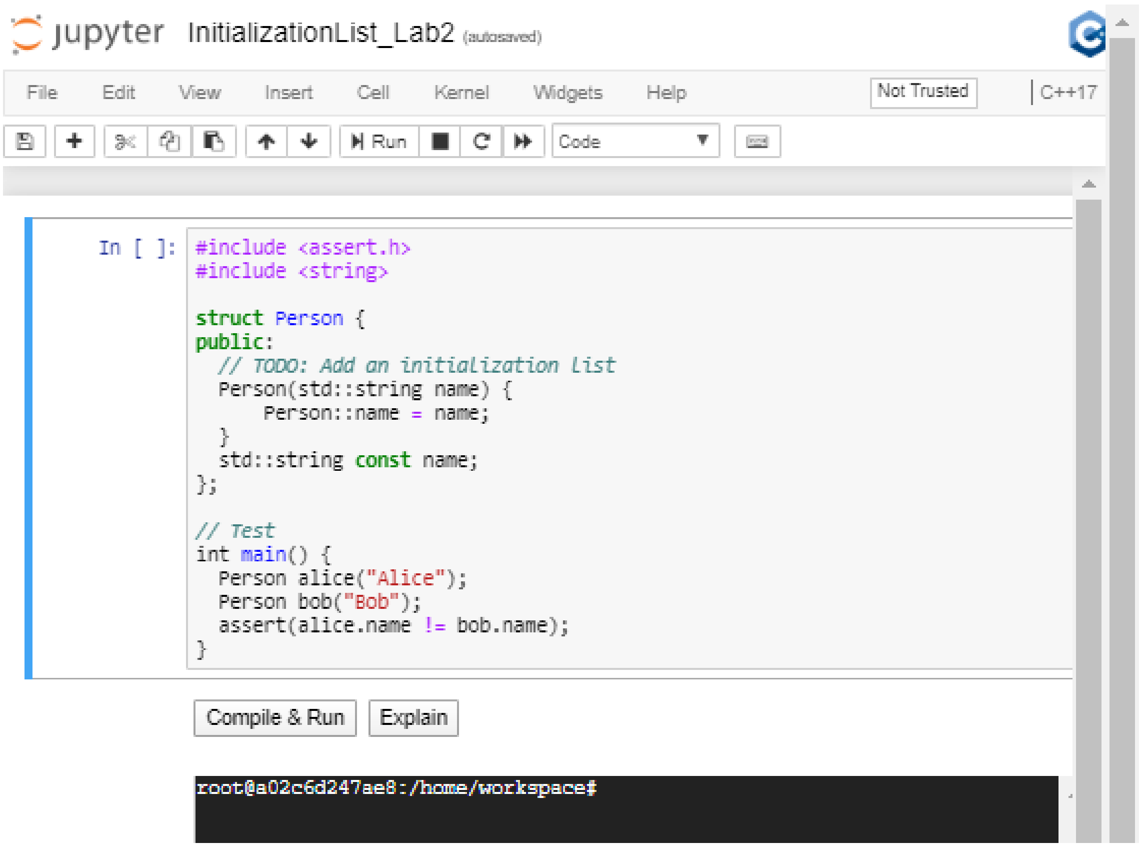
Exercise: Constructor Syntax

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** 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.


jupyter InitializationList_Lab2 (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
C++17
Run
Code
Compile & Run
Explain

```
In [ ]: #include <assert.h>
        #include <string>

        struct Person {
        public:
          // TODO: Add an initialization list
          Person(std::string name) {
              Person::name = name;
          }
          std::string const name;
        };

        // Test
        int main() {
          Person alice("Alice");
          Person bob("Bob");
          assert(alice.name != bob.name);
        }
```

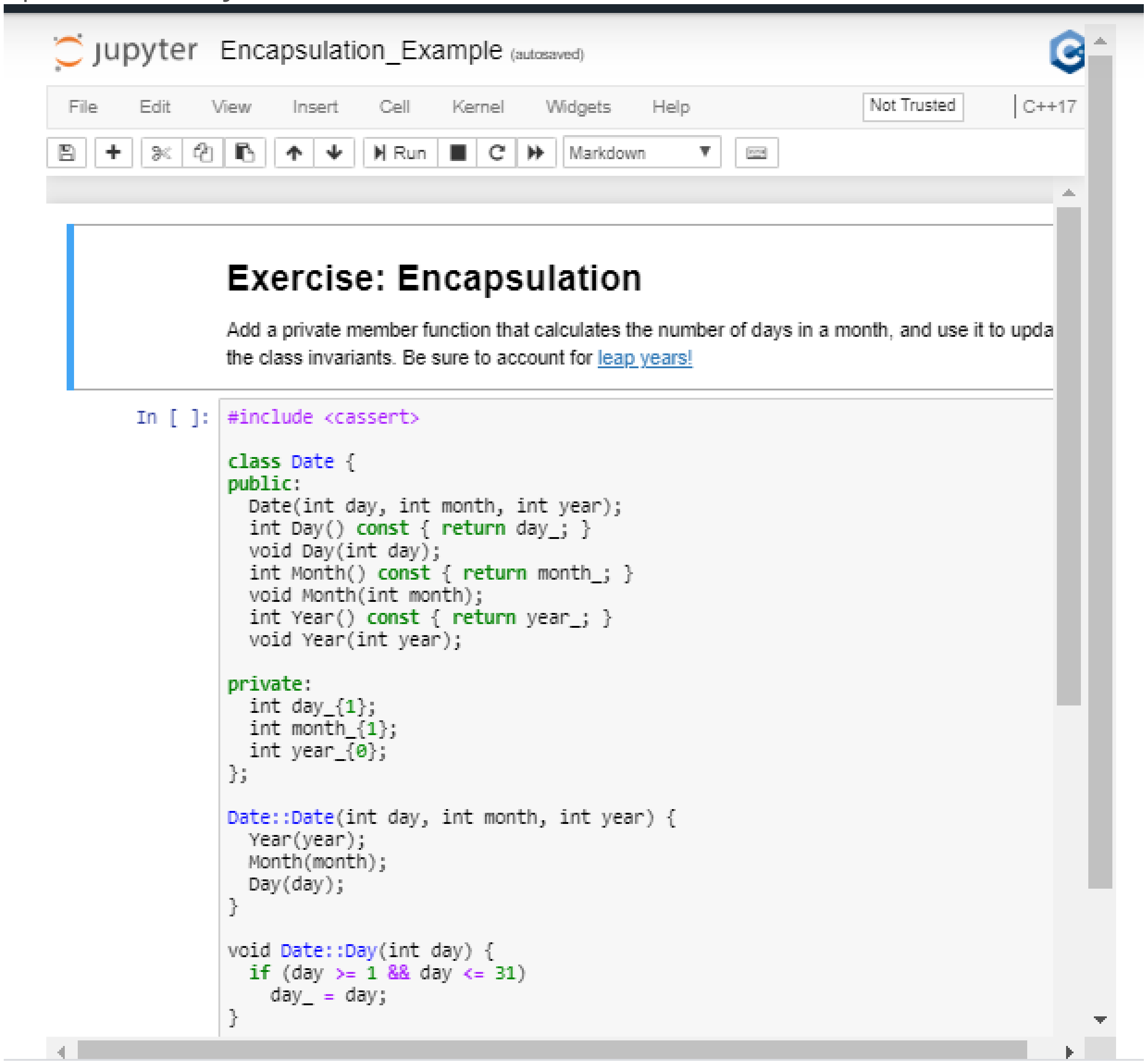
```
root@a02c6d247ae8:/home/workspace#
```

14) Encapsulation
https://youtu.be/1-j6AxEoK3c

Encapsulation

Encapsulation 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**.
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.


jupyter Encapsulation_Example (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
C++17
Run
Markdown

Exercise: Encapsulation

Add a private member function that calculates the number of days in a month, and use it to upda
the class invariants. Be sure to account for leap years!

In [ ]:
```
#include <cassert>

class Date {
public:
  Date(int day, int month, int year);
  int Day() const { return day_; }
  void Day(int day);
  int Month() const { return month_; }
  void Month(int month);
  int Year() const { return year_; }
  void Year(int year);

private:
  int day_{1};
  int month_{1};
  int year_{0};
};

Date::Date(int day, int month, int year) {
  Year(year);
  Month(month);
  Day(day);
}

void Date::Day(int day) {
  if (day >= 1 && day <= 31)
    day_ = day;
}
```

15) 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`

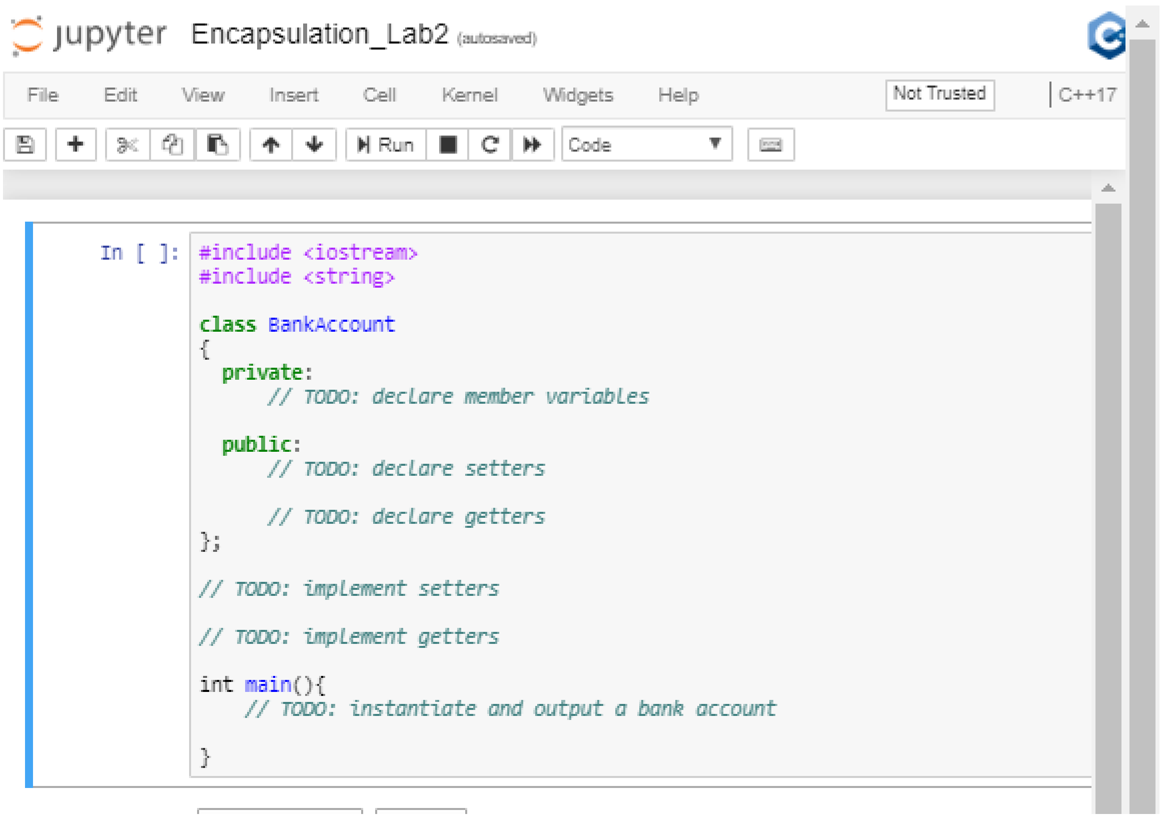
Accessors should only retrieve data. They should not change the data stored in the object.

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.


jupyter Encapsulation_Lab2 (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
C++17
Run
Code
In [ ]:

```
#include <iostream>
#include <string>

class BankAccount
{
  private:
      // TODO: declare member variables

  public:
      // TODO: declare setters

      // TODO: declare getters
};

// TODO: implement setters

// TODO: implement getters

int main(){
    // TODO: instantiate and output a bank account

}
```

16) Mutator Functions
    https://youtu.be/-WRtCE3MZB8

Mutator Functions

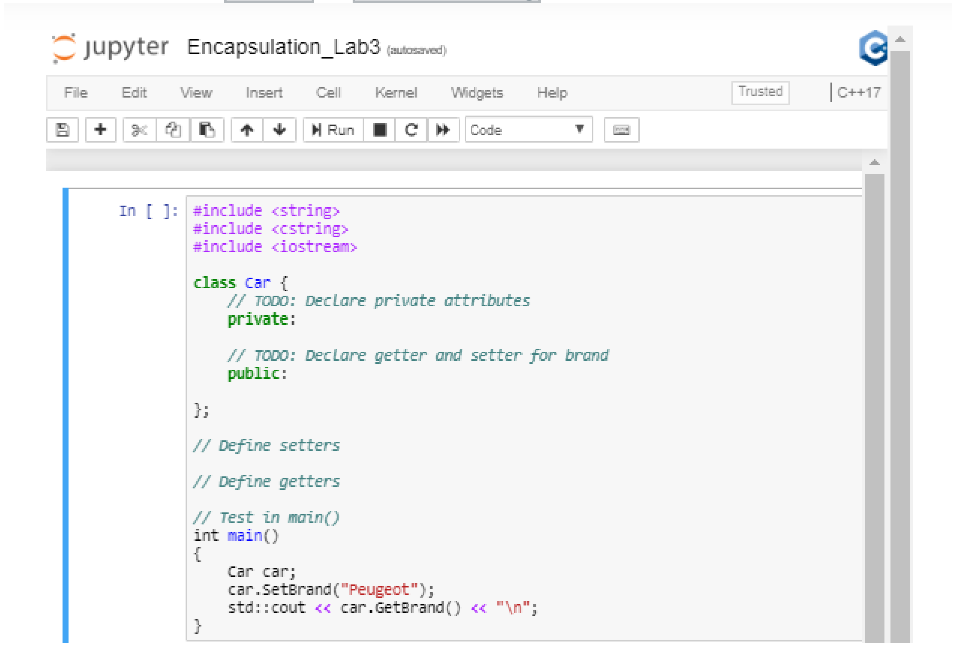
A mutator ("setter") function can apply logic ("invariants") when updating member data.

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** (a character array ending in null character) to set the value of `brand`.

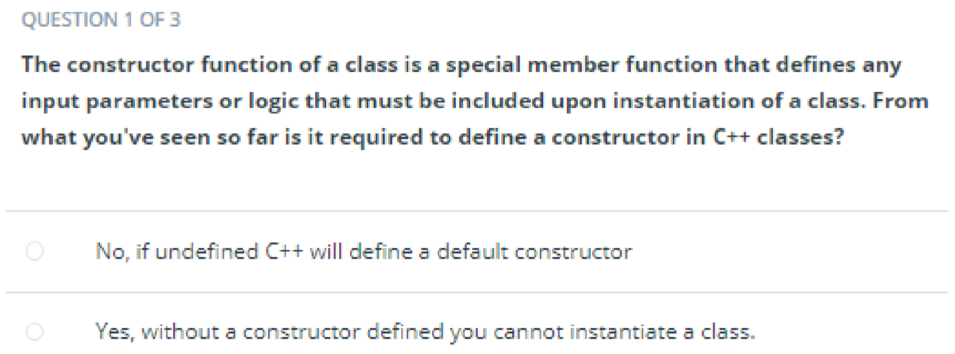
4. The accessor function for the `brand` must return a string, so in this function you first will need to convert `brand` to `std::string`, and then return it.



jupyter Encapsulation_Lab3 (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Trusted
C++17
Run
Code
In [ ]:

```
#include <string>
#include <cstring>
#include <iostream>

class Car {
    // TODO: Declare private attributes
    private:

    // TODO: Declare getter and setter for brand
    public:

};

// Define setters

// Define getters

// Test in main()
int main()
{
    Car car;
    car.SetBrand("Peugeot");
    std::cout << car.GetBrand() << "\n";
}
```

17) Quiz: Classes in C++

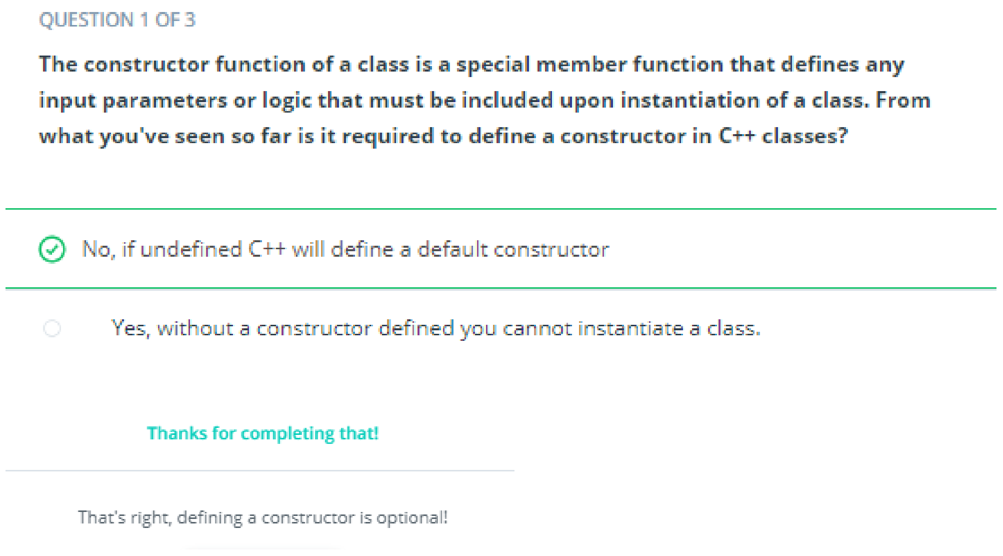


QUESTION 1 OF 3

The constructor function of a class is a special member function that defines any input parameters or logic that must be included upon instantiation of a class. From what you've seen so far is it required to define a constructor in C++ classes?

○ No, if undefined C++ will define a default constructor

○ Yes, without a constructor defined you cannot instantiate a class.



QUESTION 1 OF 3

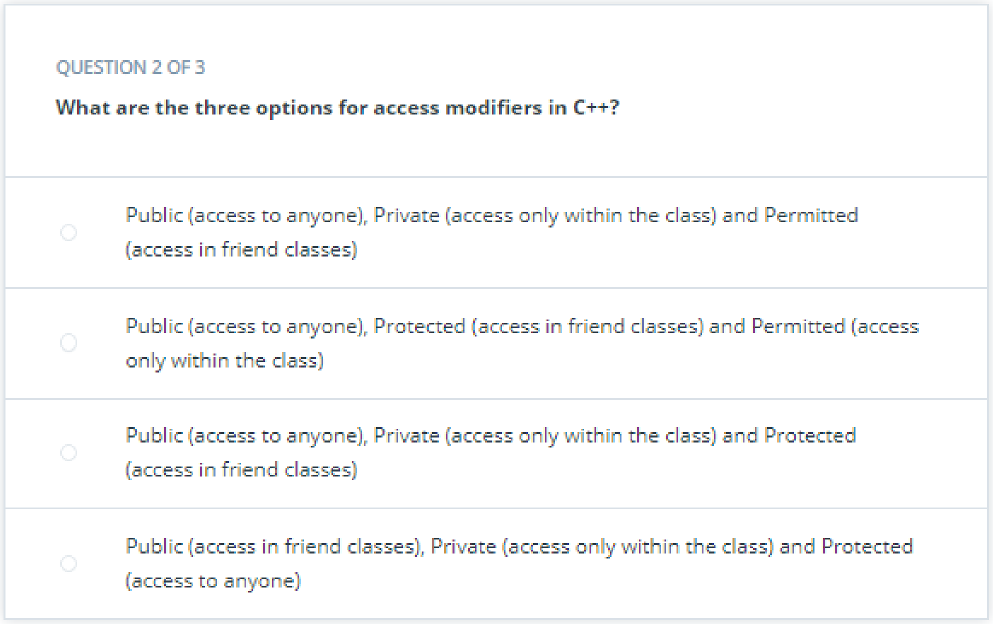
The constructor function of a class is a special member function that defines any input parameters or logic that must be included upon instantiation of a class. From what you've seen so far is it required to define a constructor in C++ classes?

[✓ No, if undefined C++ will define a default constructor]

○ Yes, without a constructor defined you cannot instantiate a class.

Thanks for completing that!

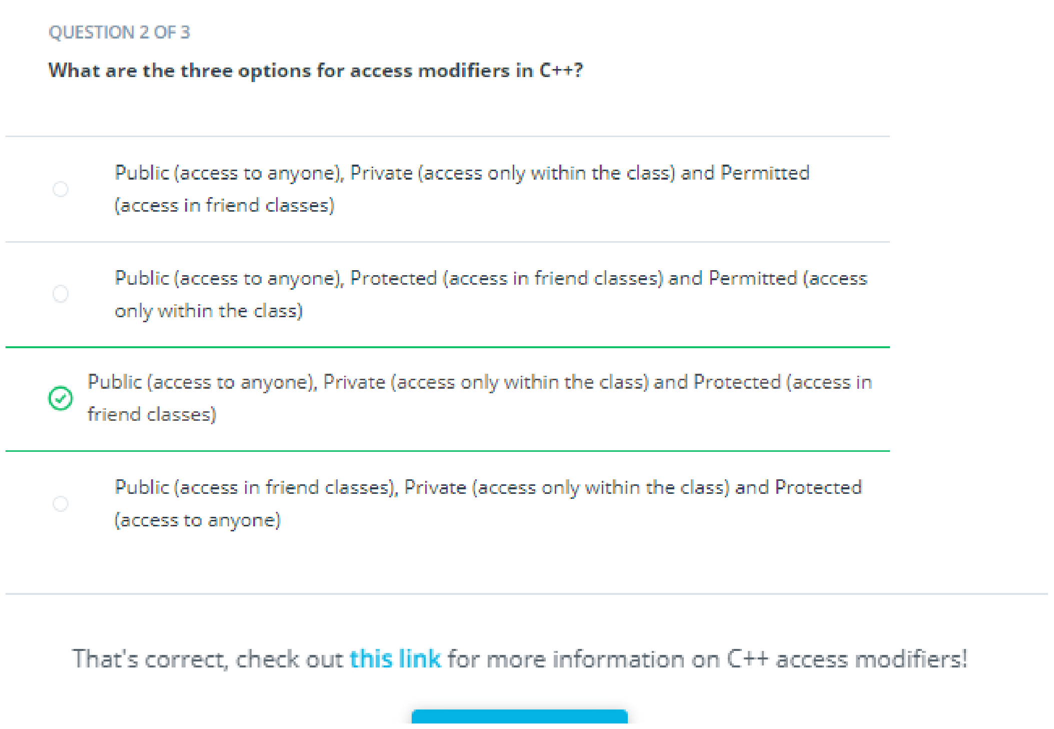
That's right, defining a constructor is optional!



QUESTION 2 OF 3

What are the three options for access modifiers in C++?

- Public (access to anyone), Private (access only within the class) and Permitted (access in friend classes)
- Public (access to anyone), Protected (access in friend classes) and Permitted (access only within the class)
- Public (access to anyone), Private (access only within the class) and Protected (access in friend classes)
- Public (access in friend classes), Private (access only within the class) and Protected (access to anyone)



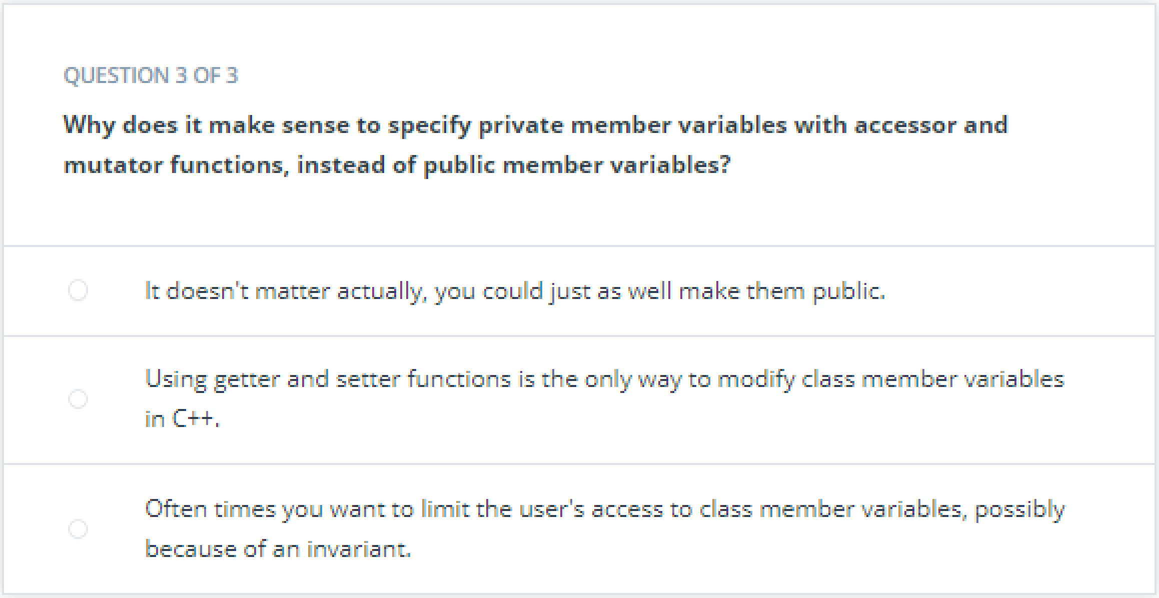
QUESTION 2 OF 3

What are the three options for access modifiers in C++?

- Public (access to anyone), Private (access only within the class) and Permitted (access in friend classes)
- Public (access to anyone), Protected (access in friend classes) and Permitted (access only within the class)
- [✓ Public (access to anyone), Private (access only within the class) and Protected (access in friend classes)]
- Public (access in friend classes), Private (access only within the class) and Protected (access to anyone)

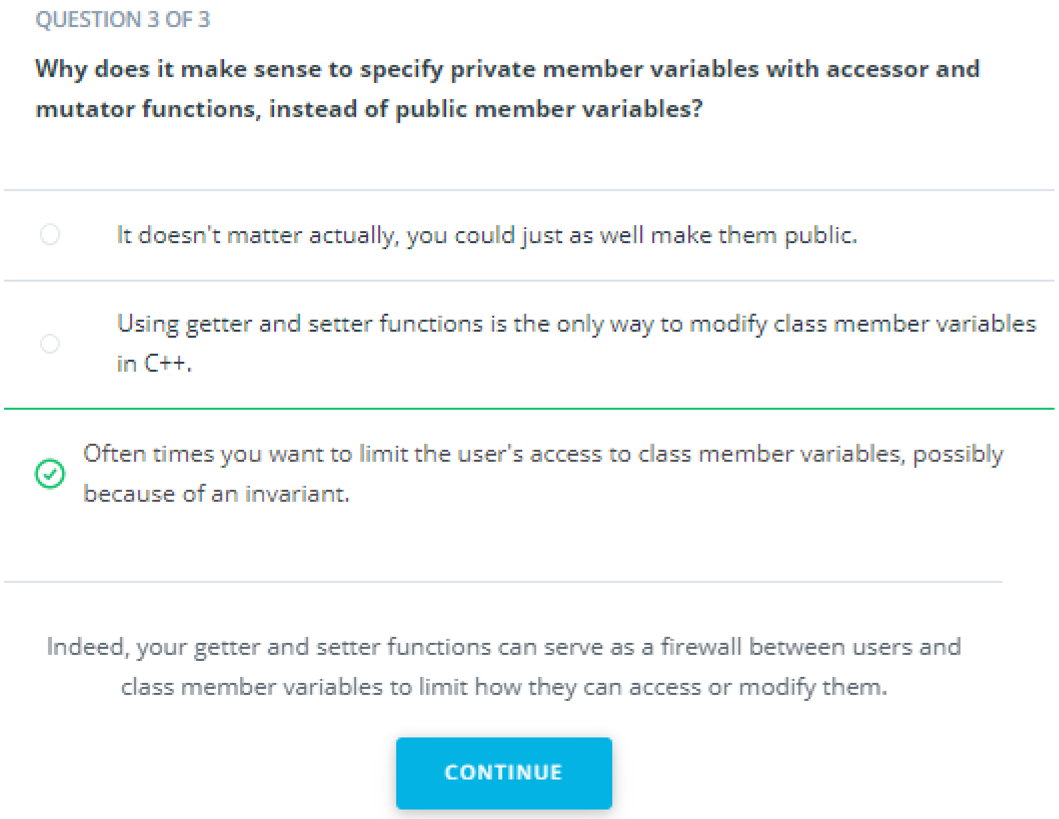
That's correct, check out this link for more information on C++ access modifiers!

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


QUESTION 3 OF 3

Why does it make sense to specify private member variables with accessor and mutator functions, instead of public member variables?

- It doesn't matter actually, you could just as well make them public.
- Using getter and setter functions is the only way to modify class member variables in C++.
- Often times you want to limit the user's access to class member variables, possibly because of an invariant.


QUESTION 3 OF 3

Why does it make sense to specify private member variables with accessor and mutator functions, instead of public member variables?

- It doesn't matter actually, you could just as well make them public.
- Using getter and setter functions is the only way to modify class member variables in C++.
- ✓ Often times you want to limit the user's access to class member variables, possibly because of an invariant.

Indeed, your getter and setter functions can serve as a firewall between users and class member variables to limit how they can access or modify them.

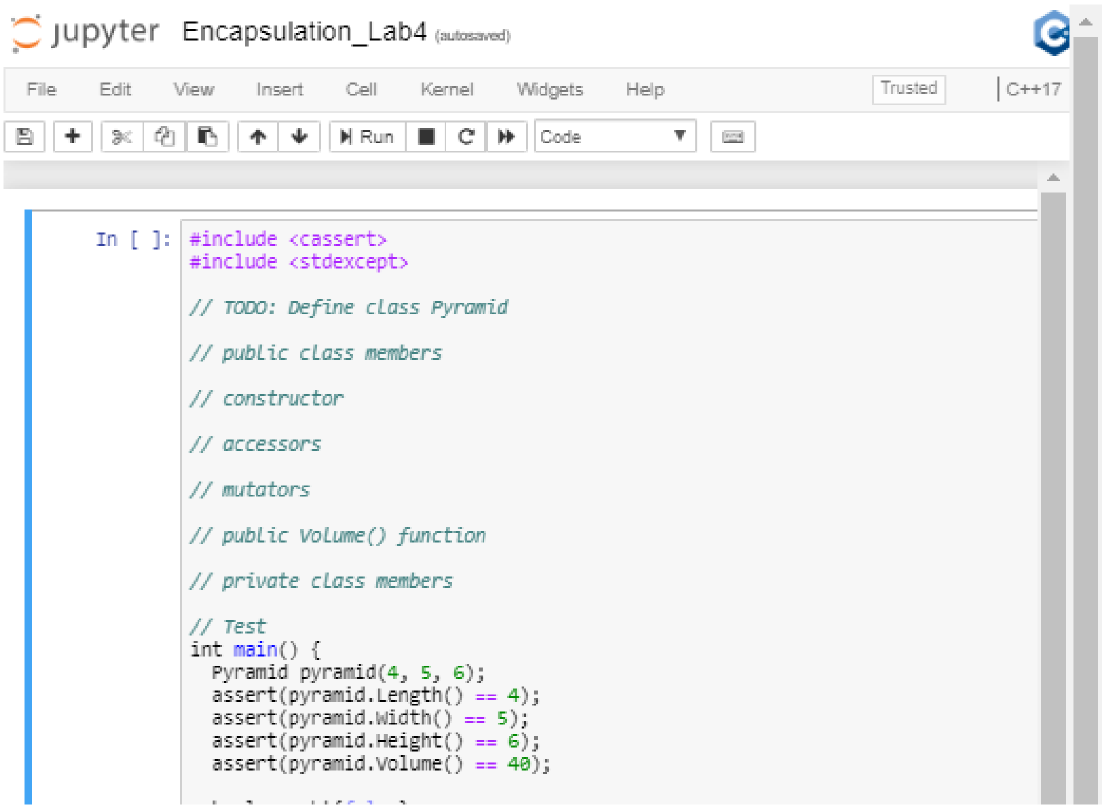
18) Exercise: Pyramid Class
https://youtu.be/g3CM02Bnamk

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**.



```
#include <cassert>
#include <stdexcept>

// TODO: Define class Pyramid

// public class members

// constructor

// accessors

// mutators

// public Volume() function

// private class members

// Test
int main() {
  Pyramid pyramid(4, 5, 6);
  assert(pyramid.Length() == 4);
  assert(pyramid.Width() == 5);
  assert(pyramid.Height() == 6);
  assert(pyramid.Volume() == 40);
```

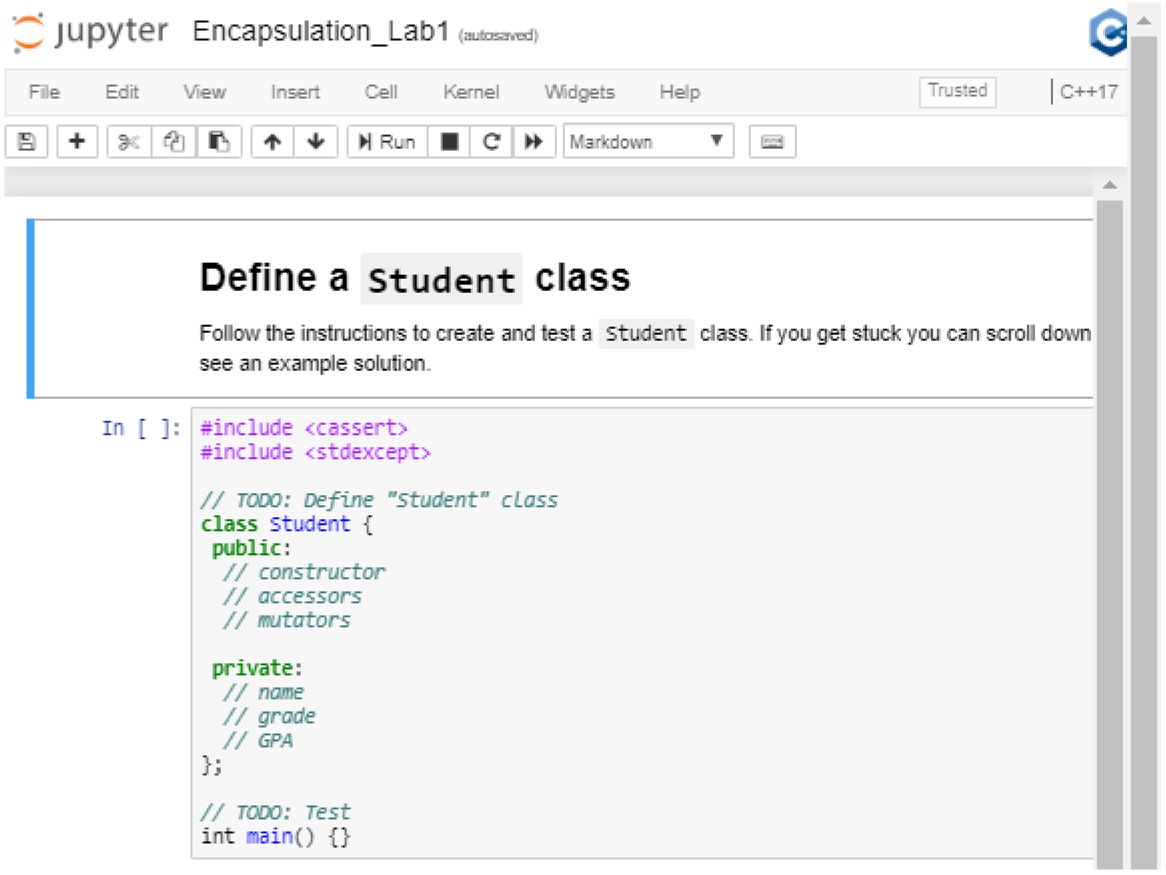
19) Exercise: Student Class
https://youtu.be/--WHwiY1Z1k

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.

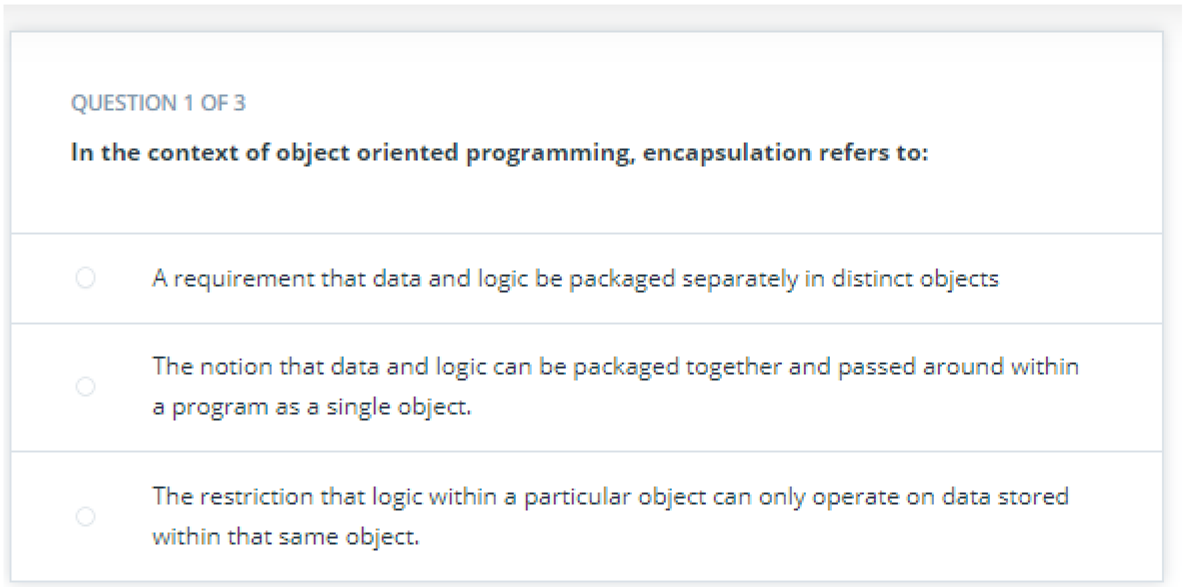

jupyter Encapsulation_Lab1 (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Trusted
C++17
Run
Markdown

Define a `Student` class

Follow the instructions to create and test a `Student` class. If you get stuck you can scroll down see an example solution.

In [ ]:
```
#include <cassert>
#include <stdexcept>

// TODO: Define "Student" class
class Student {
 public:
  // constructor
  // accessors
  // mutators

 private:
  // name
  // grade
  // GPA
};

// TODO: Test
int main() {}
```

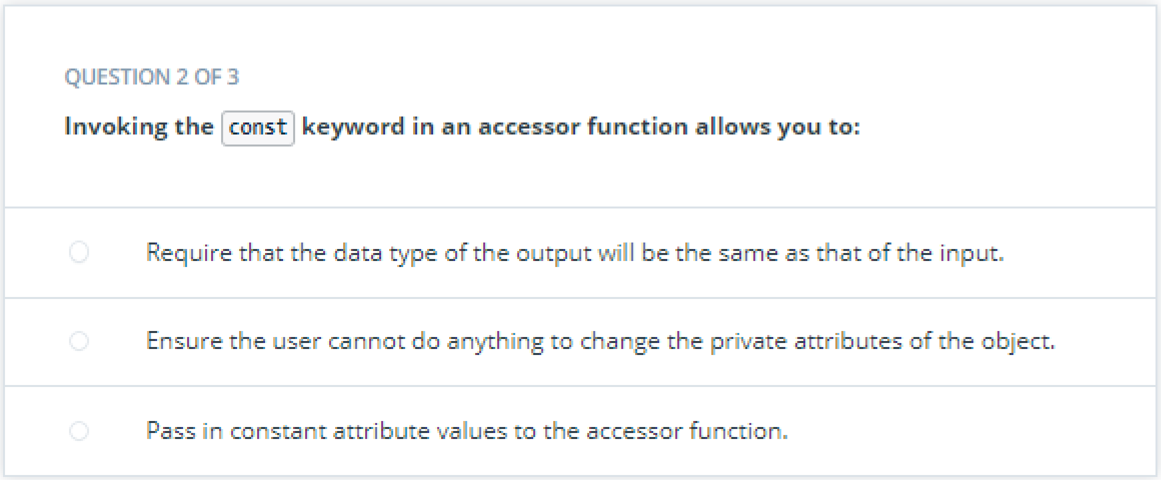
20) QUIZ: Encapsulation in c++



QUESTION 1 OF 3

In the context of object oriented programming, encapsulation refers to:

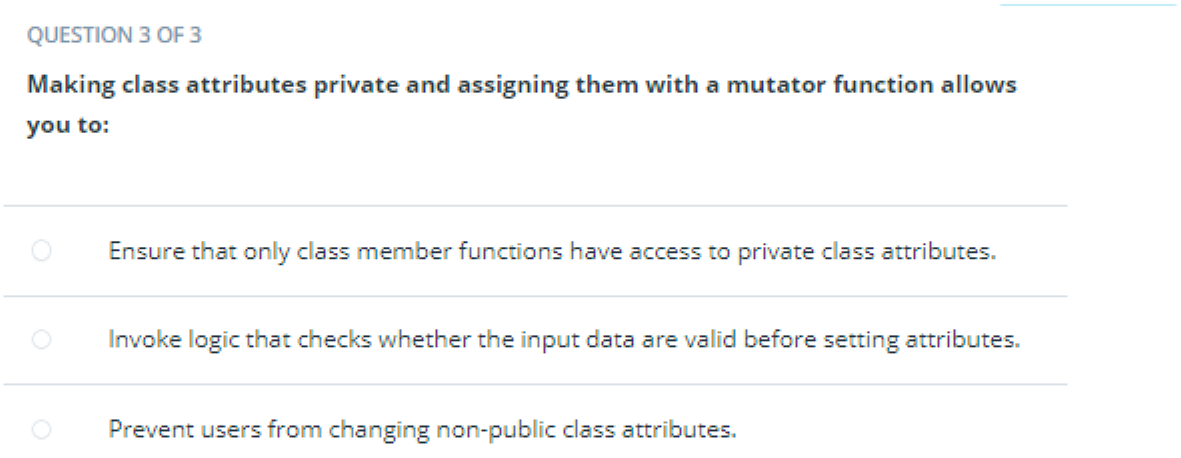
- A requirement that data and logic be packaged separately in distinct objects
- The notion that data and logic can be packaged together and passed around within a program as a single object.
- The restriction that logic within a particular object can only operate on data stored within that same object.



QUESTION 2 OF 3

Invoking the `const` keyword in an accessor function allows you to:

- Require that the data type of the output will be the same as that of the input.
- Ensure the user cannot do anything to change the private attributes of the object.
- Pass in constant attribute values to the accessor function.



QUESTION 3 OF 3

Making class attributes private and assigning them with a mutator function allows you to:

- Ensure that only class member functions have access to private class attributes.
- Invoke logic that checks whether the input data are valid before setting attributes.
- Prevent users from changing non-public class attributes.

QUESTION 1 OF 3

In the context of object oriented programming, encapsulation refers to:

- A requirement that data and logic be packaged separately in distinct objects
- [✓ The notion that data and logic can be packaged together and passed around within a program as a single object.]
- The restriction that logic within a particular object can only operate on data stored within that same object.

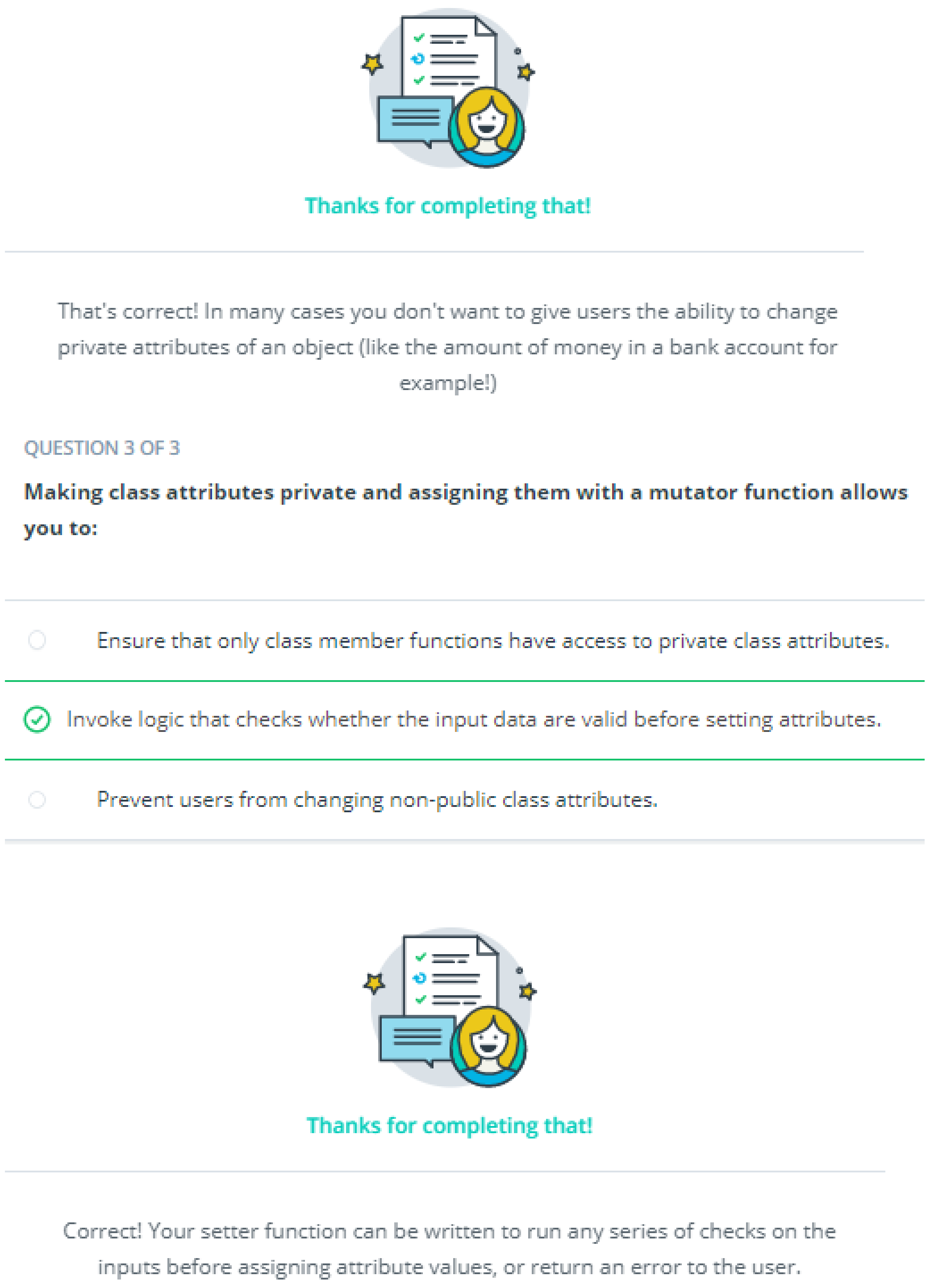
Thanks for completing that!

Indeed! The notion of encapsulation as a way of passing data and logic bundled together in a single object is at the very core of OOP.

QUESTION 2 OF 3

Invoking the `const` keyword in an accessor function allows you to:

- Require that the data type of the output will be the same as that of the input.
- [✓ Ensure the user cannot do anything to change the private attributes of the object.]
- Pass in constant attribute values to the accessor function.



Thanks for completing that!

That's correct! In many cases you don't want to give users the ability to change private attributes of an object (like the amount of money in a bank account for example!)

QUESTION 3 OF 3

Making class attributes private and assigning them with a mutator function allows you to:

○ Ensure that only class member functions have access to private class attributes.

[✓ Invoke logic that checks whether the input data are valid before setting attributes.]

○ Prevent users from changing non-public class attributes.

Thanks for completing that!

Correct! Your setter function can be written to run any series of checks on the inputs before assigning attribute values, or return an error to the user.

21) Bjarne on Abstraction
https://youtu.be/eUf7QBJNIFc

22) 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

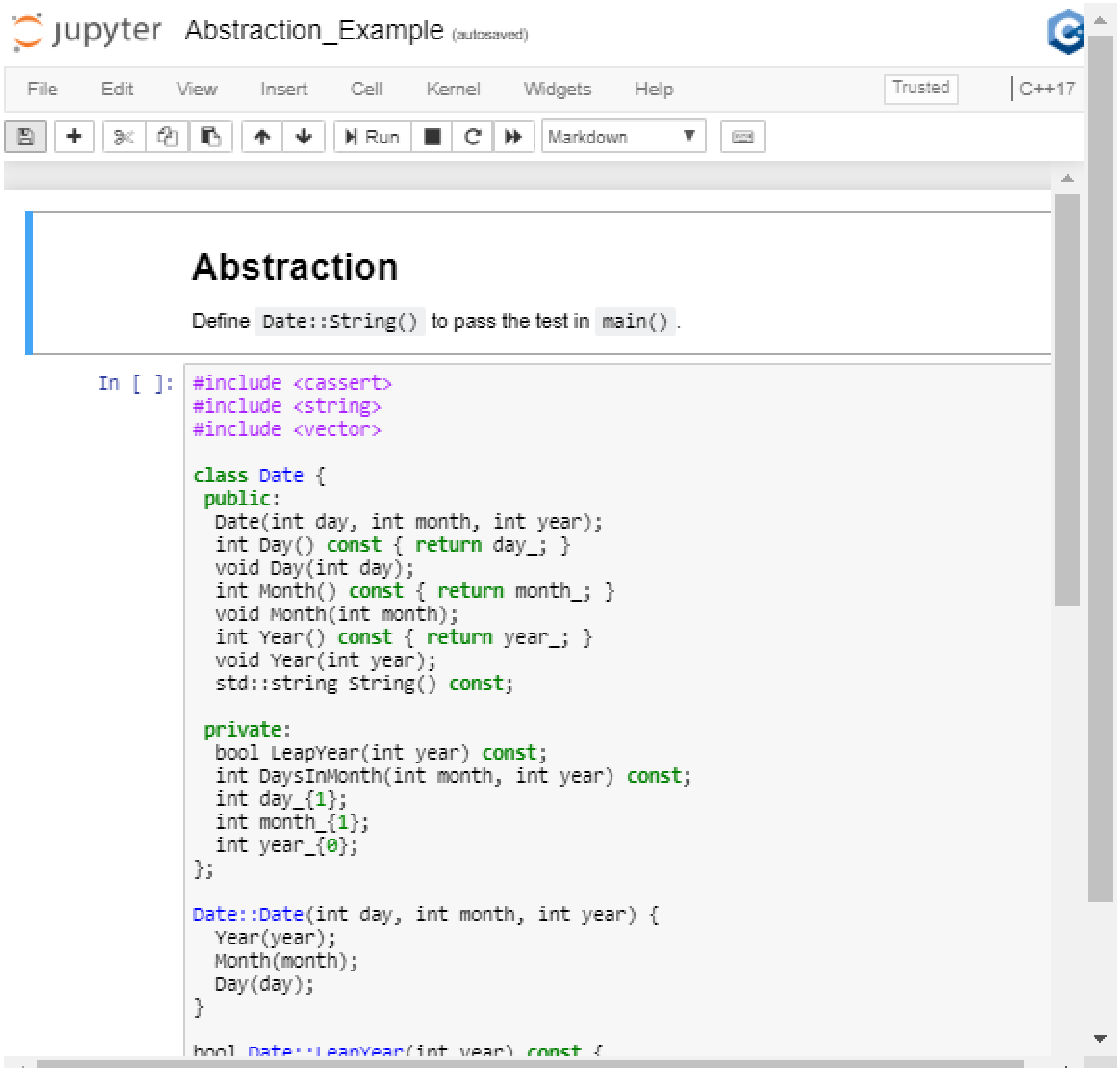
The `String()` function within this `Date` class is an example of abstraction.

```
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 `int`s instead of three separate `int`s, for example -- the user of the `Date` class will not need to know.


jupyter Abstraction_Example (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Trusted
C++17
Run
Markdown

Abstraction

Define `Date::String()` to pass the test in `main()`.

In [ ]:
```
#include <cassert>
#include <string>
#include <vector>

class Date {
 public:
  Date(int day, int month, int year);
  int Day() const { return day_; }
  void Day(int day);
  int Month() const { return month_; }
  void Month(int month);
  int Year() const { return year_; }
  void Year(int year);
  std::string String() const;

 private:
  bool LeapYear(int year) const;
  int DaysInMonth(int month, int year) const;
  int day_{1};
  int month_{1};
  int year_{0};
};

Date::Date(int day, int month, int year) {
  Year(year);
  Month(month);
  Day(day);
}

bool Date::LeapYear(int year) const {
```

23) Exercise: Sphere Class
https://youtu.be/S5iKMpA-gFE

Exercise: Sphere Class

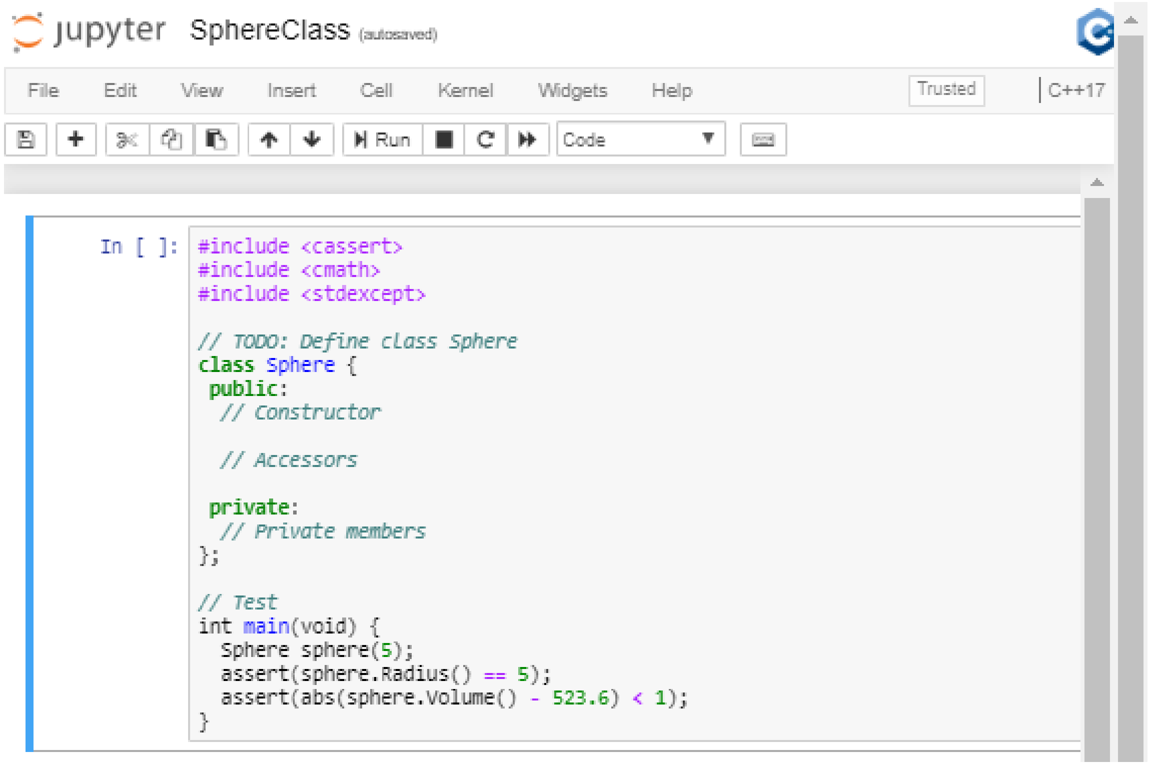
In this exercise you will practice abstraction by creating a class which represents a sphere.

Declare:

1. A constructor function that takes the radius as an argument
2. A member function that returns the **volume**

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`.


jupyter SphereClass (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Trusted
C++17
Run
Code
In [ ]:

```cpp
#include <cassert>
#include <cmath>
#include <stdexcept>

// TODO: Define class Sphere
class Sphere {
 public:
  // Constructor

  // Accessors

 private:
  // Private members
};

// Test
int main(void) {
  Sphere sphere(5);
  assert(sphere.Radius() == 5);
  assert(abs(sphere.Volume() - 523.6) < 1);
}
```

24) Exercise: Private Method
    https://youtu.be/Z75_sEnapD4

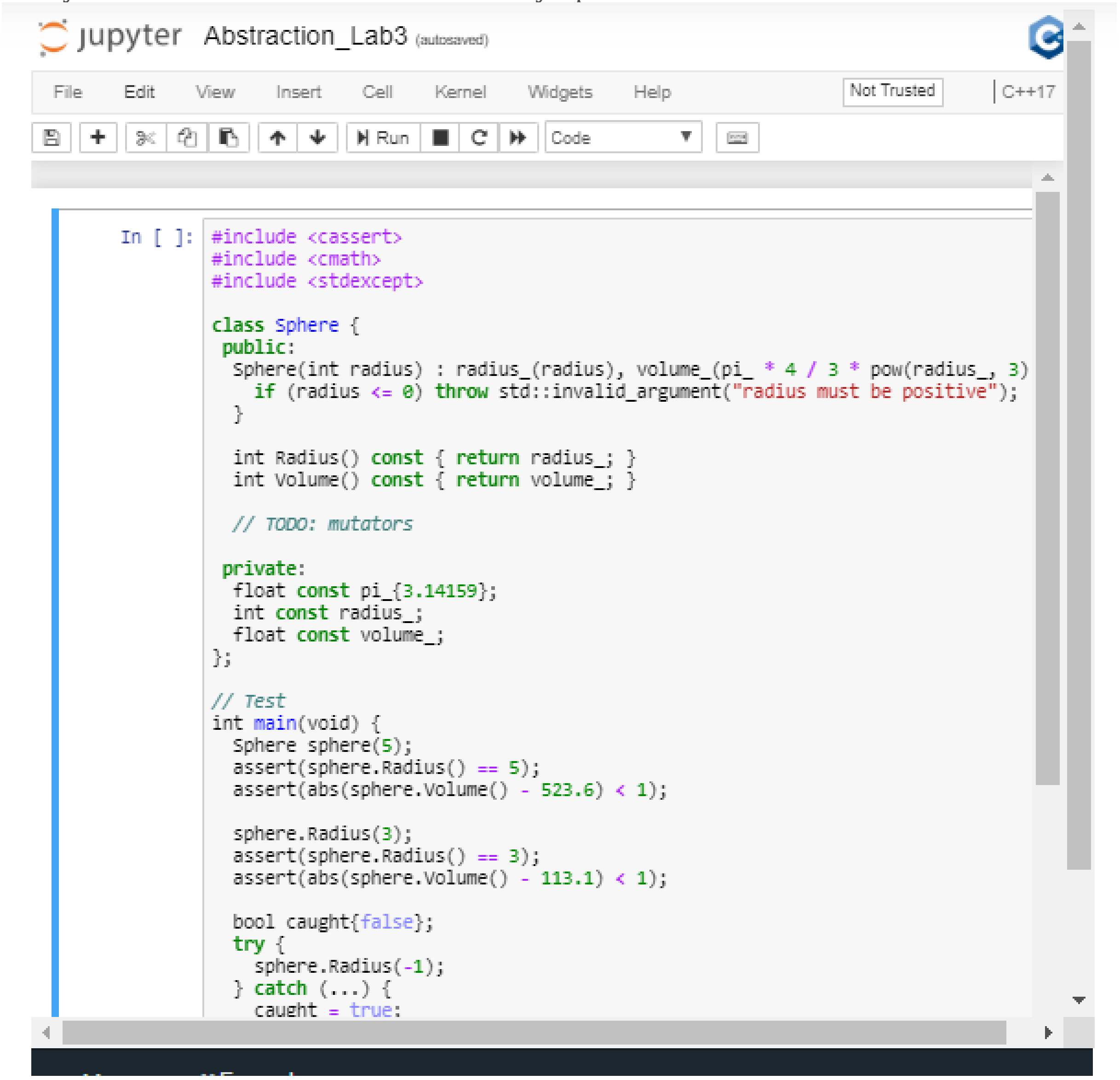
Exercise: Private Method

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** 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.

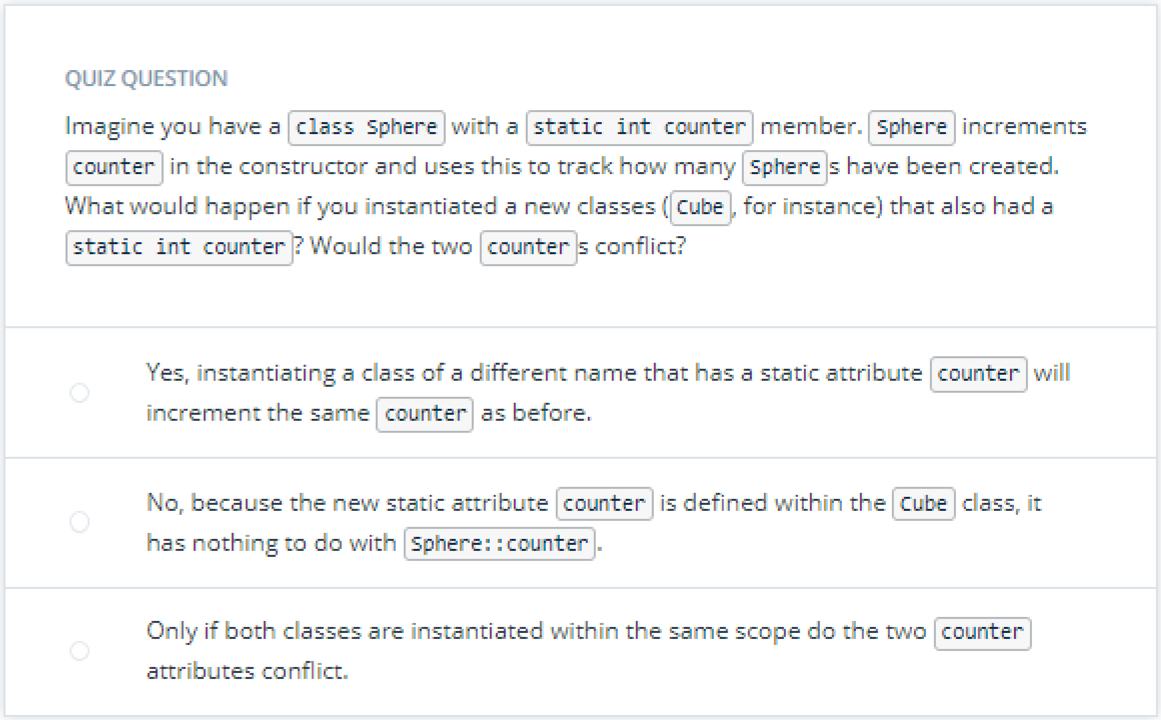

jupyter Abstraction_Lab3 (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
C++17
Run
Code

```
In [ ]: #include <cassert>
        #include <cmath>
        #include <stdexcept>

        class Sphere {
         public:
          Sphere(int radius) : radius_(radius), volume_(pi_ * 4 / 3 * pow(radius_, 3)
            if (radius <= 0) throw std::invalid_argument("radius must be positive");
          }

          int Radius() const { return radius_; }
          int Volume() const { return volume_; }

          // TODO: mutators

         private:
          float const pi_{3.14159};
          int const radius_;
          float const volume_;
        };

        // Test
        int main(void) {
          Sphere sphere(5);
          assert(sphere.Radius() == 5);
          assert(abs(sphere.Volume() - 523.6) < 1);

          sphere.Radius(3);
          assert(sphere.Radius() == 3);
          assert(abs(sphere.Volume() - 113.1) < 1);

          bool caught{false};
          try {
            sphere.Radius(-1);
          } catch (...) {
            caught = true;
```

25) Exercise: Static Members
https://youtu.be/7fBkcIL6d8k

Static Members

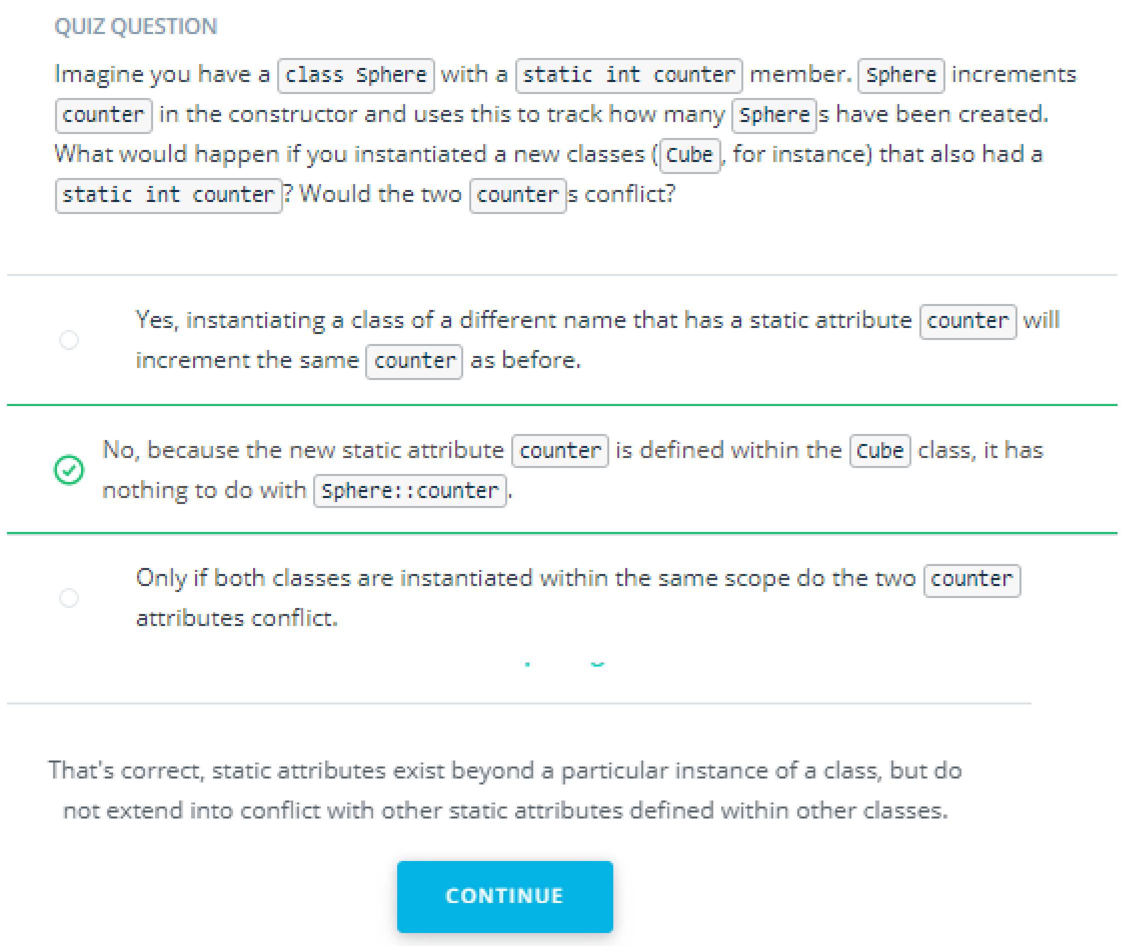
Class members can be declared `static`, which means that the member belongs to the entire class, instead of to a specific instance of the class. More specifically, a `static` member is created only once and then shared by all instances (i.e. objects) of the class. That means that if the `static` member 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.



QUIZ QUESTION

Imagine you have a `class Sphere` with a `static int counter` member. `Sphere` increments `counter` in the constructor and uses this to track how many `Sphere`s have been created. What would happen if you instantiated a new classes (`Cube`, for instance) that also had a `static int counter`? Would the two `counter`s conflict?

- Yes, instantiating a class of a different name that has a static attribute `counter` will increment the same `counter` as before.
- No, because the new static attribute `counter` is defined within the `Cube` class, it has nothing to do with `Sphere::counter`.
- Only if both classes are instantiated within the same scope do the two `counter` attributes conflict.



QUIZ QUESTION

Imagine you have a `class Sphere` with a `static int counter` member. `Sphere` increments `counter` in the constructor and uses this to track how many `Sphere`s have been created. What would happen if you instantiated a new classes (`Cube`, for instance) that also had a `static int counter`? Would the two `counter`s conflict?

- Yes, instantiating a class of a different name that has a static attribute `counter` will increment the same `counter` as before.
- [✓ No, because the new static attribute `counter` is defined within the `Cube` class, it has nothing to do with `Sphere::counter`.]
- Only if both classes are instantiated within the same scope do the two `counter` attributes conflict.

That's correct, static attributes exist beyond a particular instance of a class, but do not extend into conflict with other static attributes defined within other classes.

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:

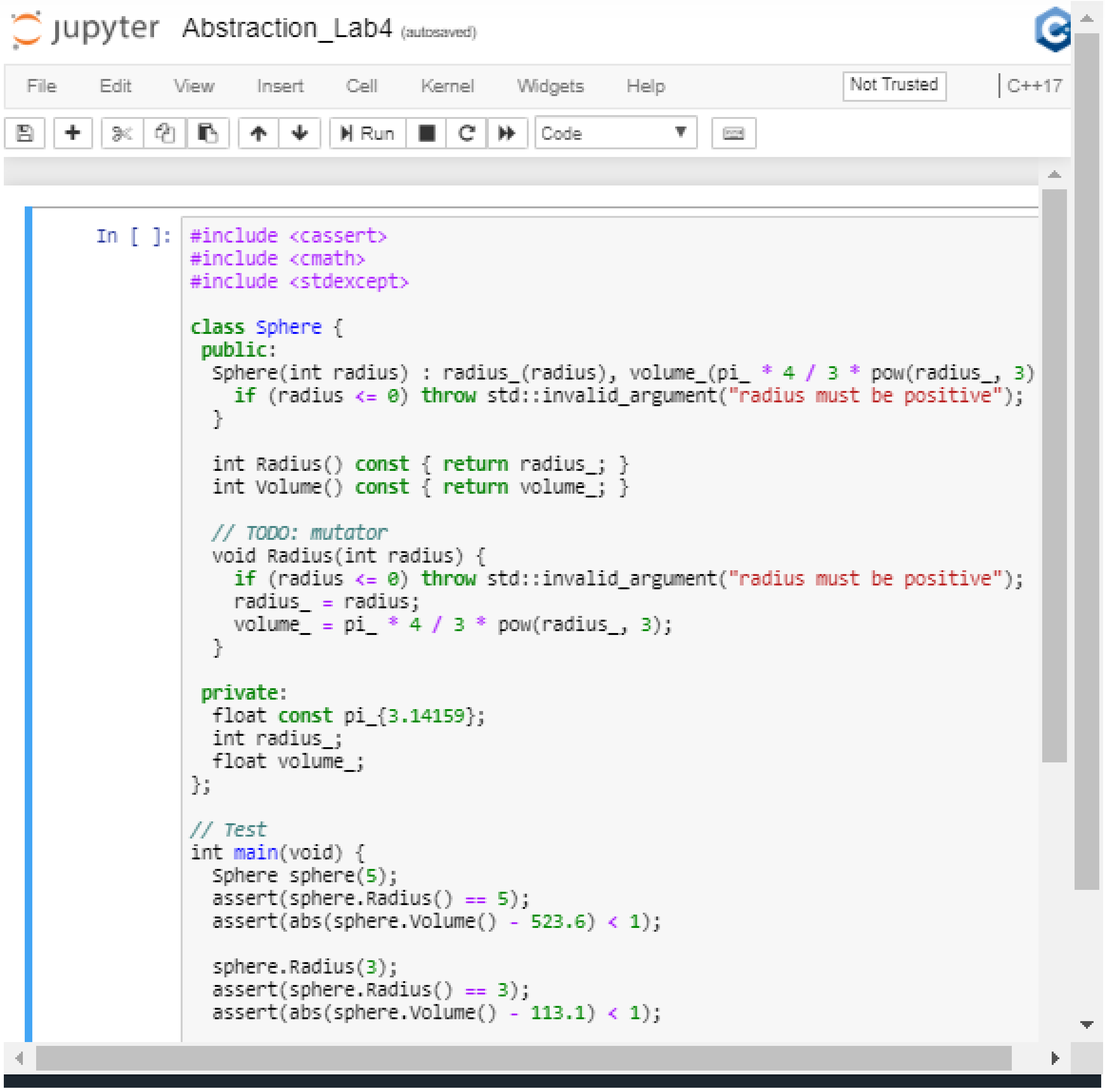
```
#include <cassert>

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`. 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?



```
#include <cassert>
#include <cmath>
#include <stdexcept>

class Sphere {
 public:
  Sphere(int radius) : radius_(radius), volume_(pi_ * 4 / 3 * pow(radius_, 3)
    if (radius <= 0) throw std::invalid_argument("radius must be positive");
  }

  int Radius() const { return radius_; }
  int Volume() const { return volume_; }

  // TODO: mutator
  void Radius(int radius) {
    if (radius <= 0) throw std::invalid_argument("radius must be positive");
    radius_ = radius;
    volume_ = pi_ * 4 / 3 * pow(radius_, 3);
  }

 private:
  float const pi_{3.14159};
  int radius_;
  float volume_;
};

// Test
int main(void) {
  Sphere sphere(5);
  assert(sphere.Radius() == 5);
  assert(abs(sphere.Volume() - 523.6) < 1);

  sphere.Radius(3);
  assert(sphere.Radius() == 3);
  assert(abs(sphere.Volume() - 113.1) < 1);
```

26) Exercise: Static Methods

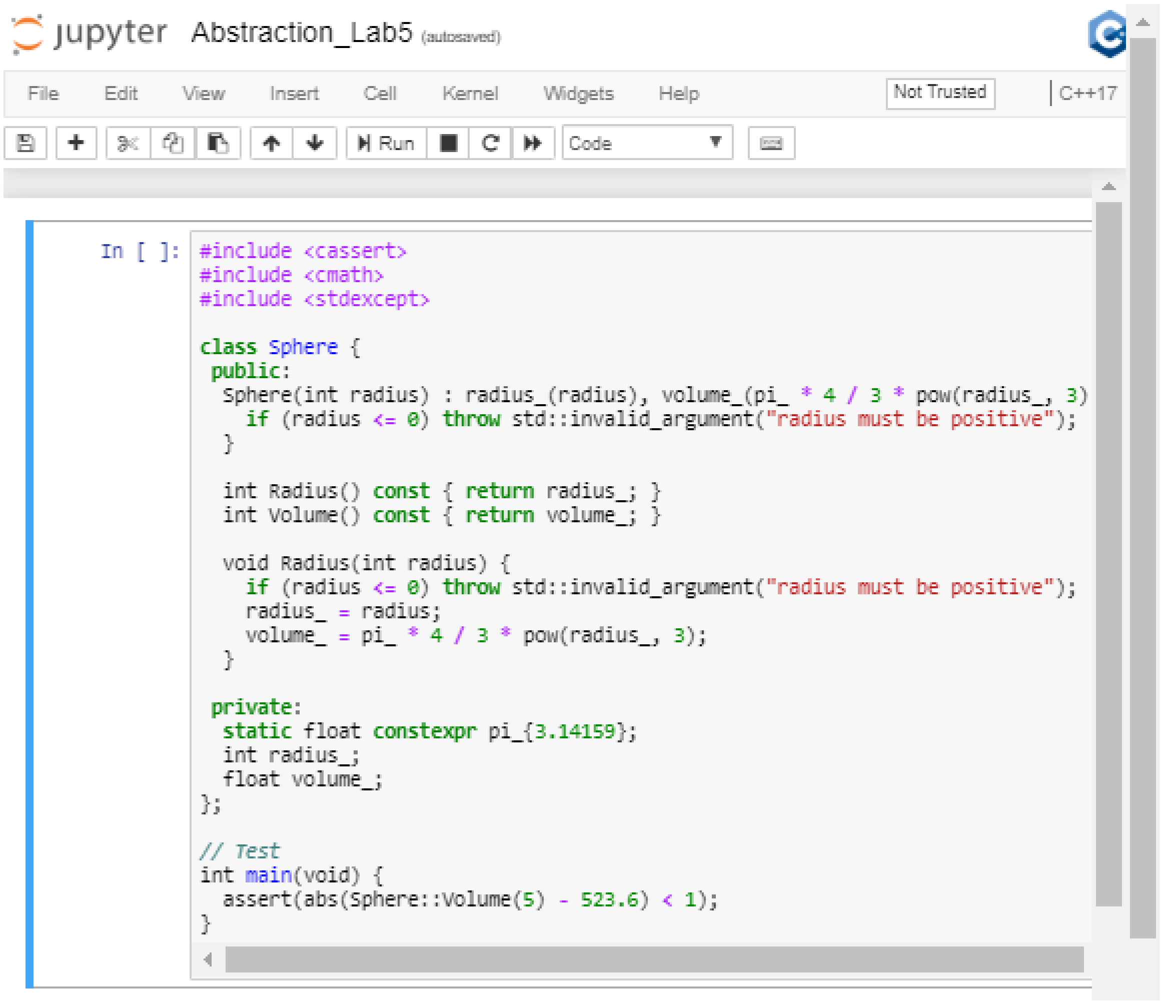
Exercise: Static Method

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



```
#include <cassert>
#include <cmath>
#include <stdexcept>

class Sphere {
 public:
  Sphere(int radius) : radius_(radius), volume_(pi_ * 4 / 3 * pow(radius_, 3)
    if (radius <= 0) throw std::invalid_argument("radius must be positive");
  }

  int Radius() const { return radius_; }
  int Volume() const { return volume_; }

  void Radius(int radius) {
    if (radius <= 0) throw std::invalid_argument("radius must be positive");
    radius_ = radius;
    volume_ = pi_ * 4 / 3 * pow(radius_, 3);
  }

 private:
  static float constexpr pi_{3.14159};
  int radius_;
  float volume_;
};

// Test
int main(void) {
  assert(abs(Sphere::Volume(5) - 523.6) < 1);
}
```

27) Bjarne on Solving Methods
https://youtu.be/Jo1NTa-krEE