

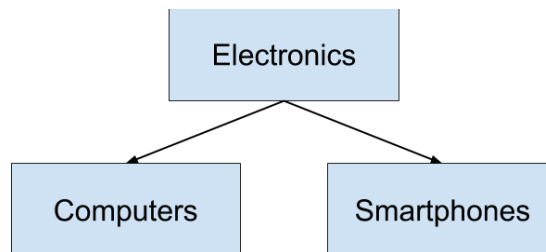
Lesson 3: Advanced OOP	Lesson 3: Advanced OOP	
ARCH	ARCH	
SOURCES	SOURCES	
CONCEPTS	CONCEPTS	
1. Polymorphism and Inheritance	11. Virtual Functions	
2. Bjarne on Inheritance	12. Polymorphism: Overriding	
3. Inheritance	13. Override	
4. Access Specifiers	14. Multiple Inheritance	
5. Exercise: Animal Class	15. Generic Programming	
6. Composition	16. Bjarne on Generic Program...	
7. Exercise: Class Hierarchy	17. Templates	
8. Exercise: Friends	18. Bjarne on Templates	
9. Polymorphism: Overloading	19. Exercise: Comparison Operat...	
10. Polymorphism: Operator Ov...	20. Deduction	
		21. Exercise: Class Template
		22. Summary
		23. Bjarne on Best Practices wit...

- 1) Polymorphism and Inheritance
<https://youtu.be/91JxGNiQdSE>
- 2) Bjarne on Inheritance
<https://youtu.be/pxDZ7VuyaHI>
- 3) Inheritance
<https://youtu.be/qu4dDc-xARM>

Inheritance

In our everyday life, we tend to divide things into groups, based on their shared characteristics. Here are some groups that you have probably used yourself: electronics, tools, vehicles, or plants.

Sometimes these groups have hierarchies. For example, computers and smartphones are both types of electronics, but computers and smartphones are also groups in and of themselves. You can imagine a tree with "electronics" at the top, and "computers" and "smartphones" each as children of the "electronics" node.



Object-oriented programming uses the same principles! For instance, imagine a `Vehicle` class:

```
class Vehicle {  
public:  
    int wheels = 0;  
    string color = "blue";  
  
    void Print() const  
    {  
        std::cout << "This " << color << " vehicle has " << wheels << " wheels!\n";  
    }  
};
```

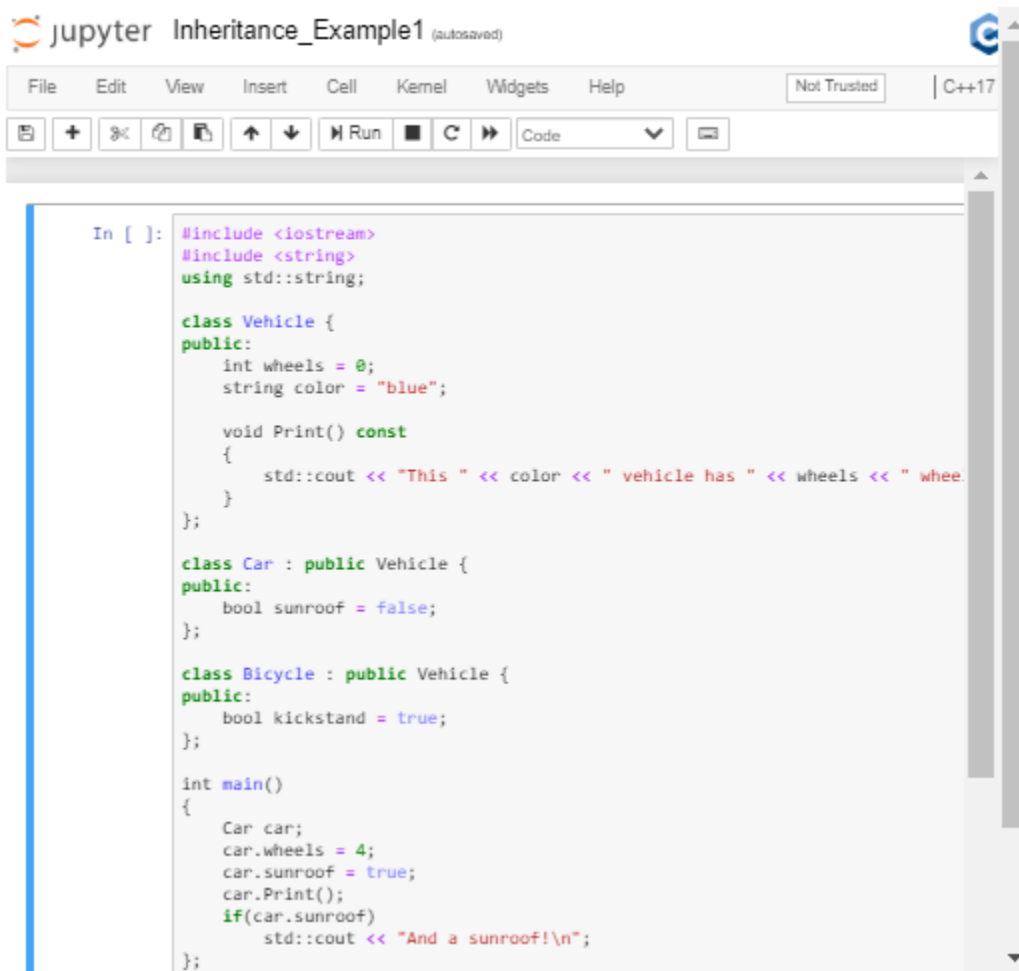
We can derive other classes from `Vehicle`, such as `Car` or `Bicycle`. One advantage is that this saves us from having to re-define all of the common member variables - in this case, `wheels` and `color` - in each derived class.

Another benefit is that derived classes, for example `Car` and `Bicycle`, can have distinct member variables, such as `sunroof` or `kickstand`. Different derived classes will have different member variables:

```
class Car : public Vehicle {  
public:  
    bool sunroof = false;  
};  
  
class Bicycle : public Vehicle {  
public:  
    bool kickstand = true;  
};
```

Instructions

1. Add a new member variable to `class Vehicle`.
2. Output that new member in `main()`.
3. Derive a new class from `Vehicle`, alongside `Car` and `Bicycle`.
4. Instantiate an object of that new class.
5. Print the object.



The image shows a Jupyter Notebook window titled "Inheritance_Example1 (autosaved)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a status bar indicating "Not Trusted" and "C++17". The code is written in a C++ cell and defines a base class `Vehicle` with attributes `wheels` and `color`, and a method `Print()`. Two derived classes, `Car` and `Bicycle`, inherit from `Vehicle`. `Car` has an additional attribute `sunroof`, and `Bicycle` has `kickstand`. The `main()` function creates a `Car` object, sets its attributes, calls `Print()`, and prints an additional message about the sunroof.

```
In [ ]: #include <iostream>
#include <string>
using std::string;

class Vehicle {
public:
    int wheels = 0;
    string color = "blue";

    void Print() const
    {
        std::cout << "This " << color << " vehicle has " << wheels << " whee.
    }
};

class Car : public Vehicle {
public:
    bool sunroof = false;
};

class Bicycle : public Vehicle {
public:
    bool kickstand = true;
};

int main()
{
    Car car;
    car.wheels = 4;
    car.sunroof = true;
    car.Print();
    if(car.sunroof)
        std::cout << "And a sunroof!\n";
};
```

4) Access Specifiers

<https://youtu.be/LVWK1aJiN40>

Inherited Access Specifiers

Just as access specifiers (i.e. `public`, `protected`, and `private`) define which class members *users* can access, the same access modifiers also define which class members *users of a derived classes* can access.

Public inheritance: the public and protected members of the base class listed after the specifier keep their member access in the derived class

Protected inheritance: the public and protected members of the base class listed after the specifier are protected members of the derived class

Private inheritance: the public and protected members of the base class listed after the specifier are private members of the derived class

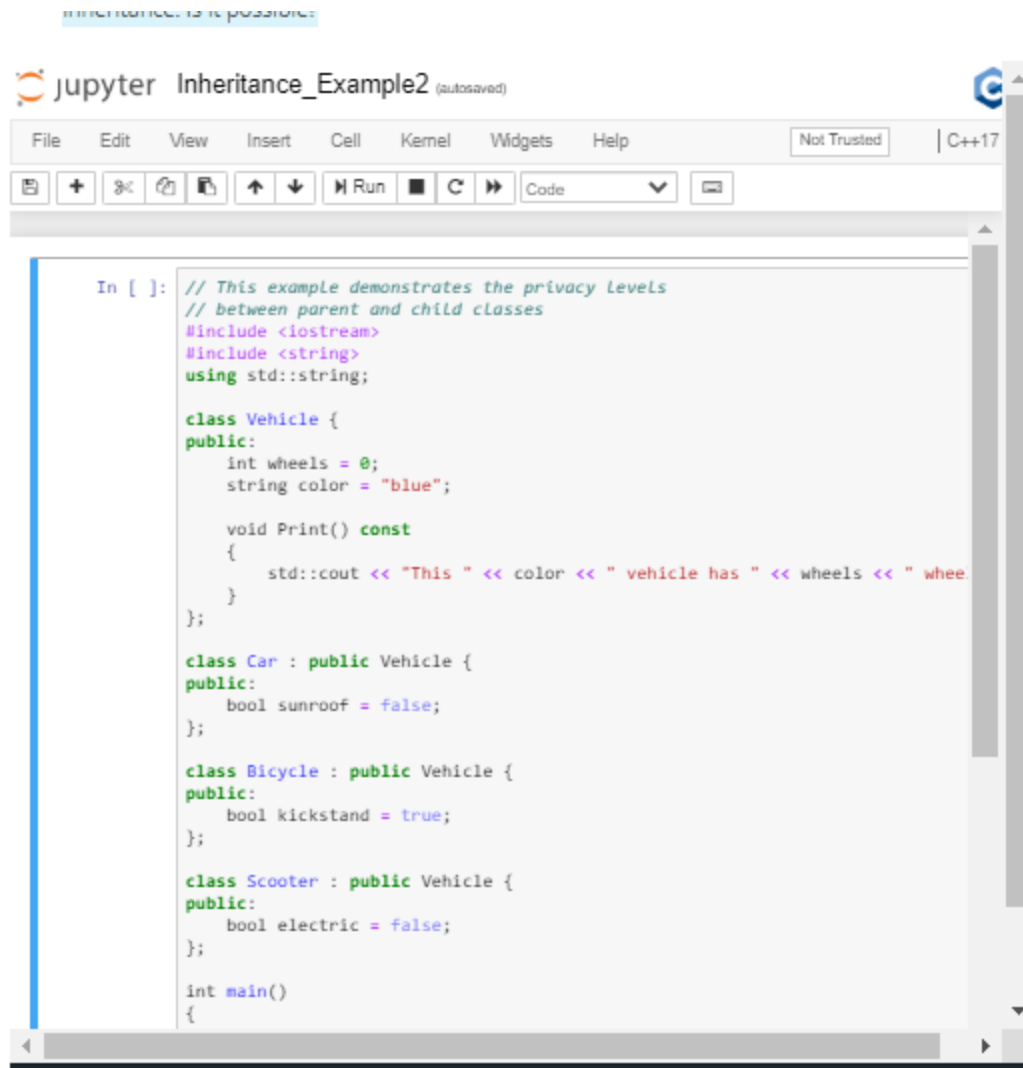
Source: [C++ reference](#)

In the exercise below, you'll experiment with access modifiers.

Instructions

1. Update the derived classes so that one has `protected` inheritance and one has `private` inheritance.
2. Try to access a `protected` member from `main()`. Is it possible?
3. Try to access a `private` member from `main()`. Is it possible?
4. Try to access a member of the base class from within the derived class that has `protected` inheritance. Is it possible?
5. Try to access a member of the base class from within the derived class that has `private` inheritance. Is it possible?

```
inheritance, is it possible;
```



```
In [ ]: // This example demonstrates the privacy levels
// between parent and child classes
#include <iostream>
#include <string>
using std::string;

class Vehicle {
public:
    int wheels = 0;
    string color = "blue";

    void Print() const
    {
        std::cout << "This " << color << " vehicle has " << wheels << " whee.
    }
};

class Car : public Vehicle {
public:
    bool sunroof = false;
};

class Bicycle : public Vehicle {
public:
    bool kickstand = true;
};

class Scooter : public Vehicle {
public:
    bool electric = false;
};

int main()
{
```

5) Exercise: Animal Class

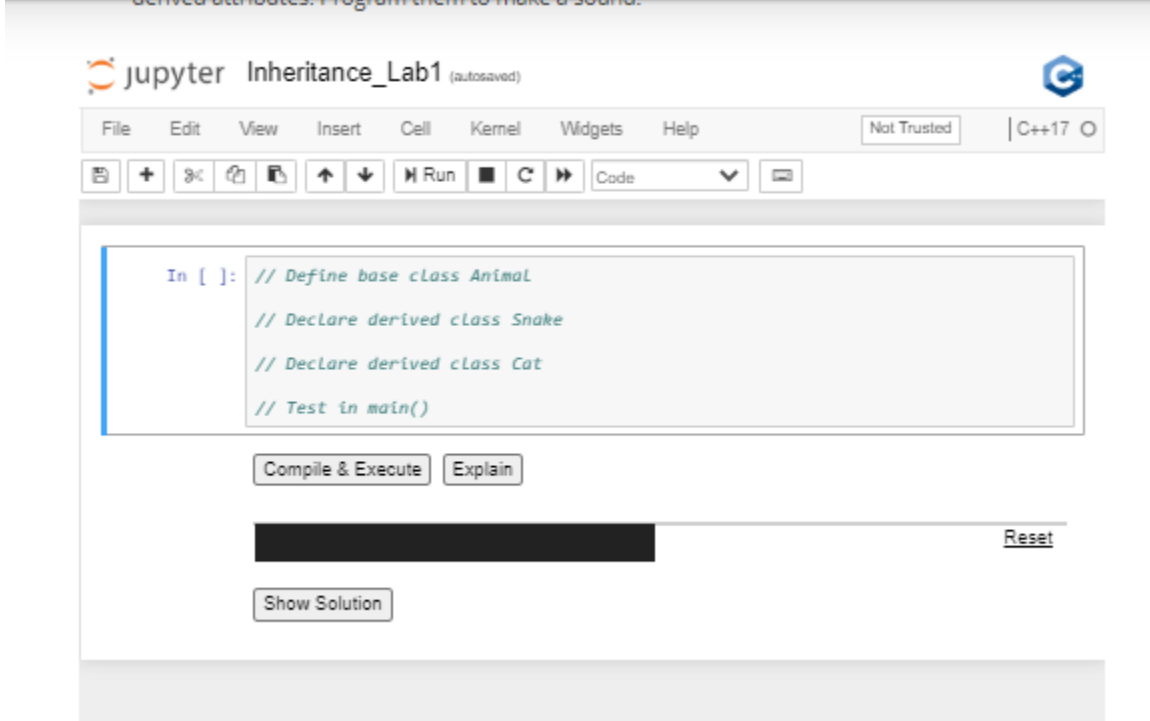
Inheritance

In this exercise you will practice building an inheritance hierarchy.

Instructions

1. Define a class `Animal`.
2. Define 3 member variables: `color`, `name`, `age`.
3. Define a derived class `Snake` that inherits from the base class `Animal`.
4. Create a member variable `length` for the `Snake` class.
5. Create a derived class `Cat` that inherits from the base class `Animal`.
6. Create a member variable `height` for the `Cat` class.
7. Create `MakeSound()` member functions for each of the derived classes.

8. In the `main()` function instantiate `Snake` and `Cat` objects. Initialize both their unique and derived attributes. Program them to make a sound.



6) Composition

<https://youtu.be/iUkRGy6kK4A>

Composition

Composition is a closely related alternative to inheritance. Composition involves constructing ("composing") classes from other classes, instead of inheriting traits from a parent class.

A common way to distinguish "composition" from "inheritance" is to think about what an object can do, rather than what it is. This is often expressed as "**has a**" versus "**is a**".

From the standpoint of composition, a cat "has a" head and "has a" set of paws and "has a" tail.

From the standpoint of inheritance, a cat "is a" mammal.

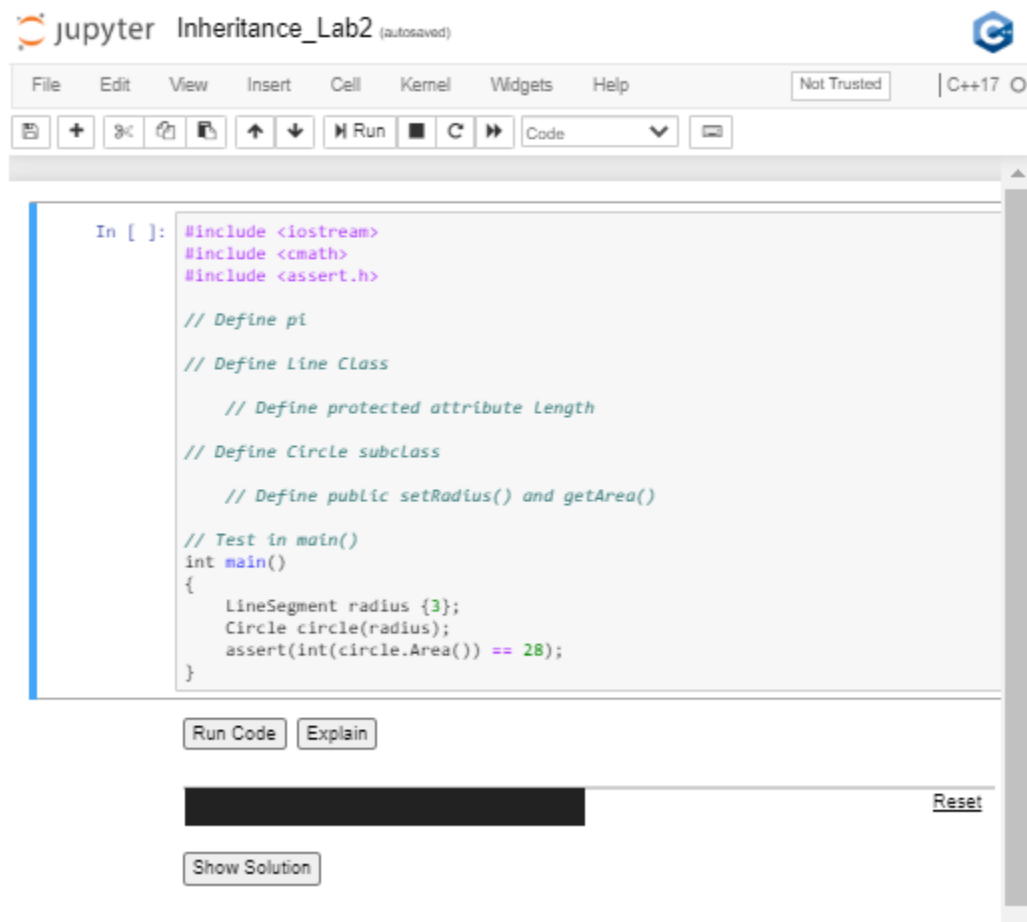
There is **no hard and fast rule** about when to prefer composition over inheritance. In general, if a class needs only extend a small amount of functionality beyond what is already offered by another class, it makes sense to **inherit** from that other class. However, if a class needs to contain functionality from a variety of otherwise unrelated classes, it makes sense to **compose** the class from those other classes.

In this example, you'll practice working with composition in C++.

Instructions

In this exercise, you will start with a `LineSegment` class and create a `Circle` class. Note that you will compose `Circle` from `LineSegment`, instead of inheriting `Circle` from `LineSegment`. Specifically, the `length` attribute from `LineSegment` will become the circle's radius.

1. Create a class `LineSegment`.
2. Declare an attribute `length` in class `LineSegment`.
3. Define pi (3.14159) with a `macro`.
4. Create a class `Circle`, composed of a `LineSegment` that represents the circle's radius. Use this radius to calculate the area of the circle (area of a circle = πr^2).
5. Verify the behavior of `Circle` in `main()`.



The image shows a Jupyter Notebook interface titled "Inheritance_Lab2 (autosaved)". The code is written in C++ and defines two classes: `LineSegment` and `Circle`. The `LineSegment` class has a protected attribute `length`. The `Circle` class is composed of a `LineSegment` object and has a public method `Area()` that calculates the area of the circle using the formula πr^2 . The `main()` function tests the `Circle` class by creating a `Circle` object with a radius of 3 and asserting that its area is 28.

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

// Define pi
#define pi 3.14159

// Define Line Class
class LineSegment {
    // Define protected attribute length
protected:
    int length;
};

// Define Circle subclass
class Circle {
    // Define public setRadius() and getArea()
public:
    Circle(int radius) : radius(radius) {}
    int Area() const { return (int)(pi * radius * radius); }
};

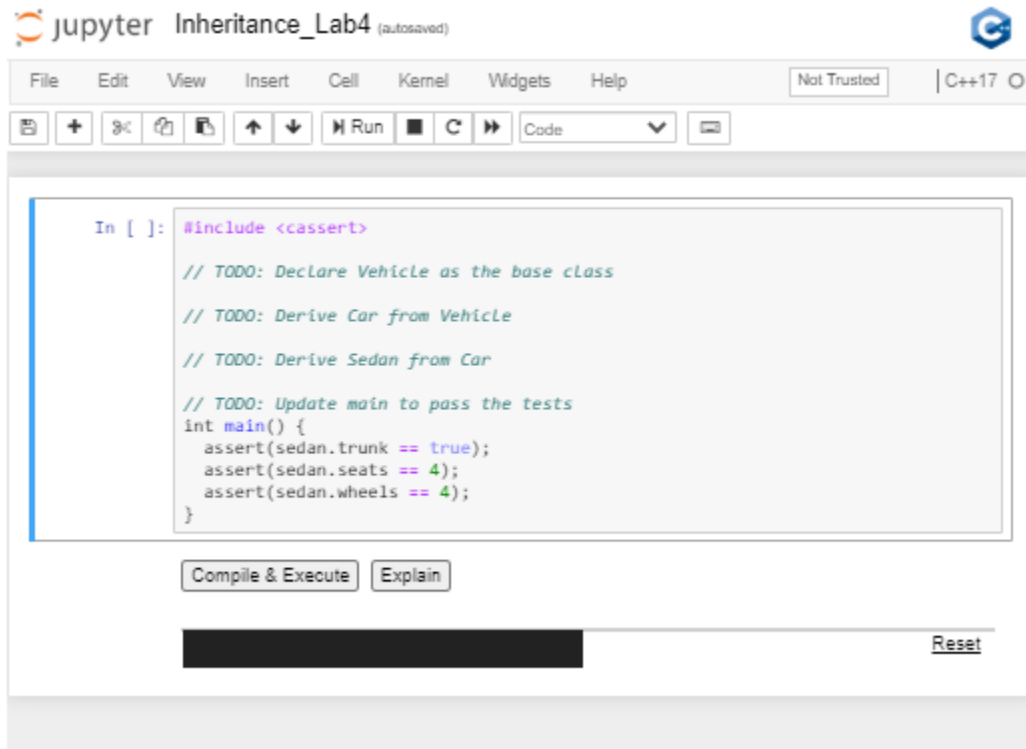
// Test in main()
int main()
{
    LineSegment radius {3};
    Circle circle(radius);
    assert(int(circle.Area()) == 28);
}
```

Buttons: Run Code, Explain, Show Solution, Reset

7) Exercise: Class Hierarchy

Exercise: Class Hierarchy

Multi-level inheritance is term used for chained classes in an inheritance tree. Have a look at the example in the notebook below to get a feel for multi-level inheritance.



The image shows a Jupyter Notebook titled "Inheritance_Lab4 (autosaved)". The notebook interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a status bar (Not Trusted, C++17), and a toolbar with icons for file operations, code execution, and output viewing. The code cell contains the following C++ code:

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

// TODO: Declare Vehicle as the base class

// TODO: Derive Car from Vehicle

// TODO: Derive Sedan from Car

// TODO: Update main to pass the tests
int main() {
    assert(sedan.trunk == true);
    assert(sedan.seats == 4);
    assert(sedan.wheels == 4);
}
```

Below the code cell are buttons for "Compile & Execute" and "Explain". At the bottom right of the notebook interface is a "Reset" button.

8) Exercise: Friends

<https://youtu.be/GxdPV4mz7wg>

Friends

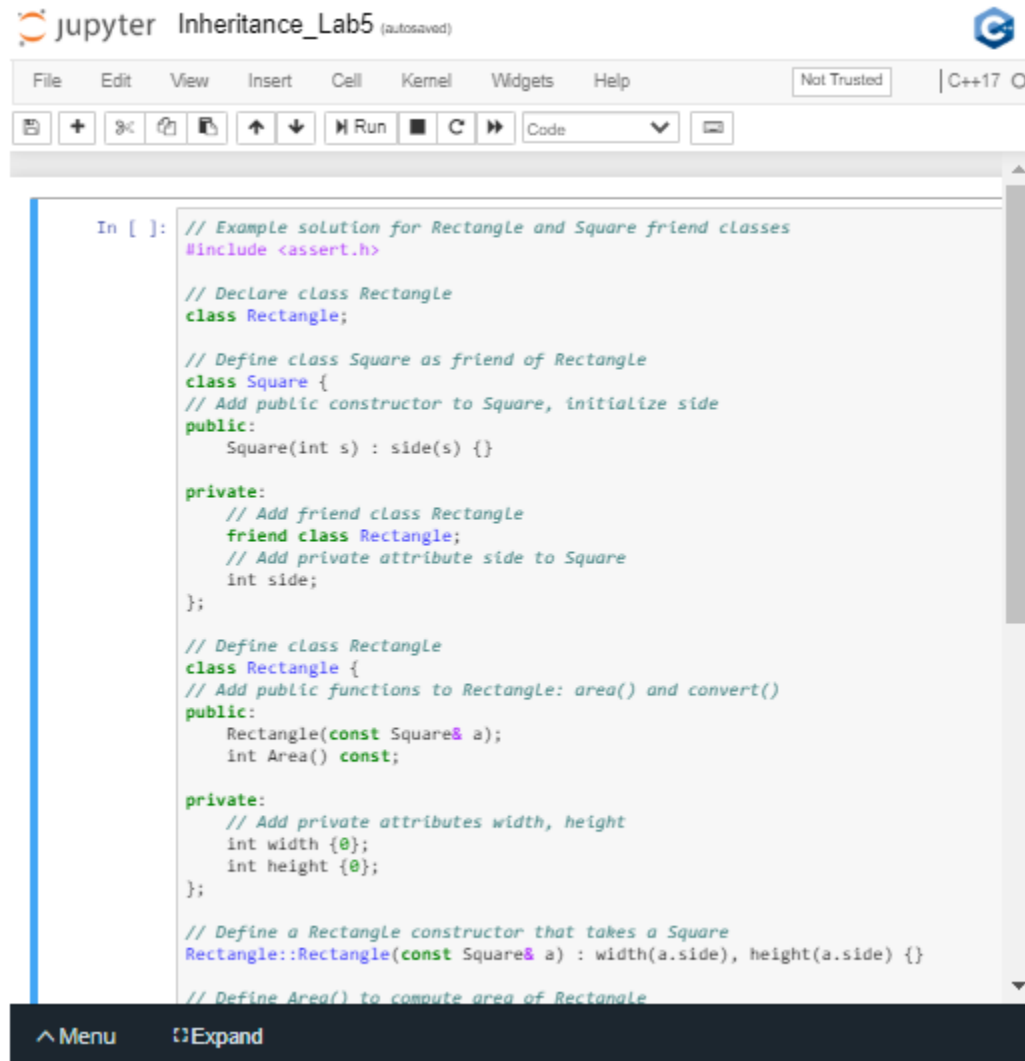
In C++, `friend` classes provide an alternative inheritance mechanism to derived classes. The main difference between classical inheritance and friend inheritance is that a `friend` class can access private members of the base class, which isn't the case for classical inheritance. In classical inheritance, a derived class can only access public and protected members of the base class.

Instructions

In this exercise you will experiment with friend classes. In the notebook below, implement the following steps:

1. Declare a class `Rectangle`.
2. Define a class `Square`.
3. Add class `Rectangle` as a friend of the class `Square`.

4. Add a private attribute `side` to class `Square`.
5. Create a public constructor in class `Square` that initializes the `side` attribute.
6. Add private members `width` and `height` to class `Rectangle`.
7. Add a `Rectangle()` constructor that takes a `Square` as an argument.
8. Add an `Area()` function to class `Rectangle`.



The screenshot shows a Jupyter Notebook window titled "Inheritance_Lab5 (autosaved)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running, and code execution. The code editor displays the following C++ code:

```
In [ ]: // Example solution for Rectangle and Square friend classes
#include <assert.h>

// Declare class Rectangle
class Rectangle;

// Define class Square as friend of Rectangle
class Square {
// Add public constructor to Square, initialize side
public:
    Square(int s) : side(s) {}

private:
    // Add friend class Rectangle
    friend class Rectangle;
    // Add private attribute side to Square
    int side;
};

// Define class Rectangle
class Rectangle {
// Add public functions to Rectangle: area() and convert()
public:
    Rectangle(const Square& a);
    int Area() const;

private:
    // Add private attributes width, height
    int width {0};
    int height {0};
};

// Define a Rectangle constructor that takes a Square
Rectangle::Rectangle(const Square& a) : width(a.side), height(a.side) {}

// Define Area() to compute area of Rectangle
```

9) Polymorphism: Overloading

<https://youtu.be/Y-SSHBtvPHo>

Polymorphism

Polymorphism is means "assuming many forms".

In the context of object-oriented programming, **polymorphism** describes a paradigm in which a function may behave differently depending on how it is called. In particular, the function will perform differently based on its inputs.

Polymorphism can be achieved in two ways in C++: overloading and overriding. In this exercise we will focus on overloading.

Overloading

In C++, you can write two (or more) versions of a function with the same name. This is called "[overloading](#)". Overloading requires that we leave the function name the same, but we modify the function signature. For example, we might define the same function name with multiple different configurations of input arguments.

This example of `class Date` overloads:

```
#include <ctime>

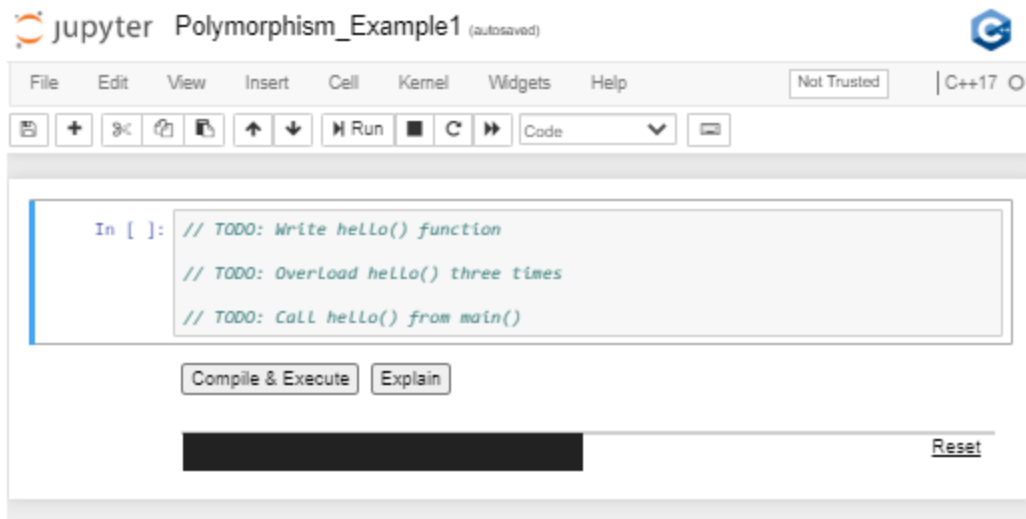
class Date {
public:
    Date(int day, int month, int year) : day_(day), month_(month), year_(year) {}
    Date(int day, int month) : day_(day), month_(month) // automatically sets the Date to the c
urrent year
    {
        time_t t = time(NULL);
        tm* timePtr = localtime(&t);
        year_ = timePtr->tm_year;
    }

private:
    int day_;
    int month_;
    int year_;
};
```

Instructions

Overloading can happen outside of an object-oriented context, too. In this exercise, you will practice overloading a normal function that is not a class member.

1. Create a function `hello()` that outputs, "Hello, World!"
2. Create a `class Human`.
3. Overload `hello()` by creating a function `hello(Human human)`. This function should output, "Hello, Human!"
4. Create 2 more classes and use those classes to further overload the `hello()` function.



10) Polymorphism: Operator Overloading

<https://youtu.be/ej8uoPtFoo>

Operator Overloading

. In this exercise you'll see how to achieve polymorphism with **operator overloading**. You can choose any operator from the ASCII table and give it your own set of rules!

Operator overloading can be useful for many things. Consider the `+` operator. We can use it to add `ints`, `doubles`, `floats`, or even `std::strings`.

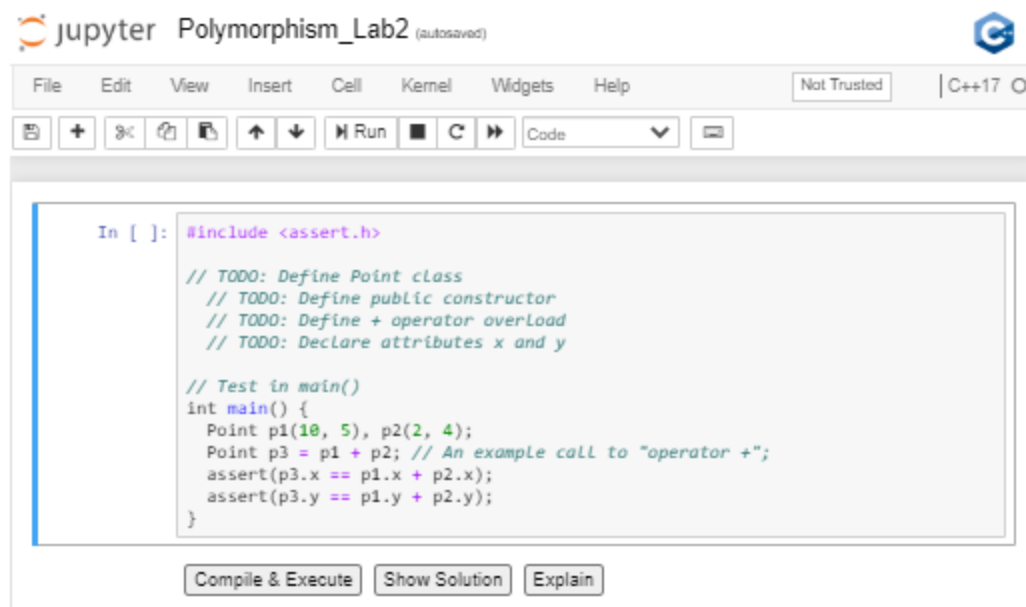
In order to overload an operator, use the `operator` keyword in the function signature:

```
Complex operator+(const Complex& addend) {  
    //...logic to add complex numbers  
}
```

Imagine vector addition. You might want to perform vector addition on a pair of points to add their x and y components. The compiler won't recognize this type of operation on its own, because this data is user defined. However, you can overload the `+` operator so it performs the action that you want to implement.

Instructions

1. Define class `Point`.
2. Declare a prototype of overload method for `+` operator.
3. Confirm the tests pass.



The image shows a Jupyter Lab2 interface with a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running, and code execution. The main area contains a code editor with the following C++ code:

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

// TODO: Define Point class
// TODO: Define public constructor
// TODO: Define + operator overload
// TODO: Declare attributes x and y

// Test in main()
int main() {
    Point p1(10, 5), p2(2, 4);
    Point p3 = p1 + p2; // An example call to "operator +"
    assert(p3.x == p1.x + p2.x);
    assert(p3.y == p1.y + p2.y);
}
```

Below the code editor are three buttons: "Compile & Execute", "Show Solution", and "Explain".

11) Virtual Functions
<https://youtu.be/2krvZ3-INUk>

Virtual Functions

Virtual functions are a polymorphic feature. These functions are declared (and possibly defined) in a base class, and can be overridden by derived classes.

This approach declares an **interface** at the base level, but delegates the implementation of the interface to the derived classes.

In this exercise, `class Shape` is the base class. Geometrical shapes possess both an area and a perimeter. `Area()` and `Perimeter()` should be virtual functions of the base class interface. Append `= 0` to each of these functions in order to declare them to be "pure" virtual functions.

A **pure virtual function** is a **virtual function** that the base class **declares** but does not **define**.

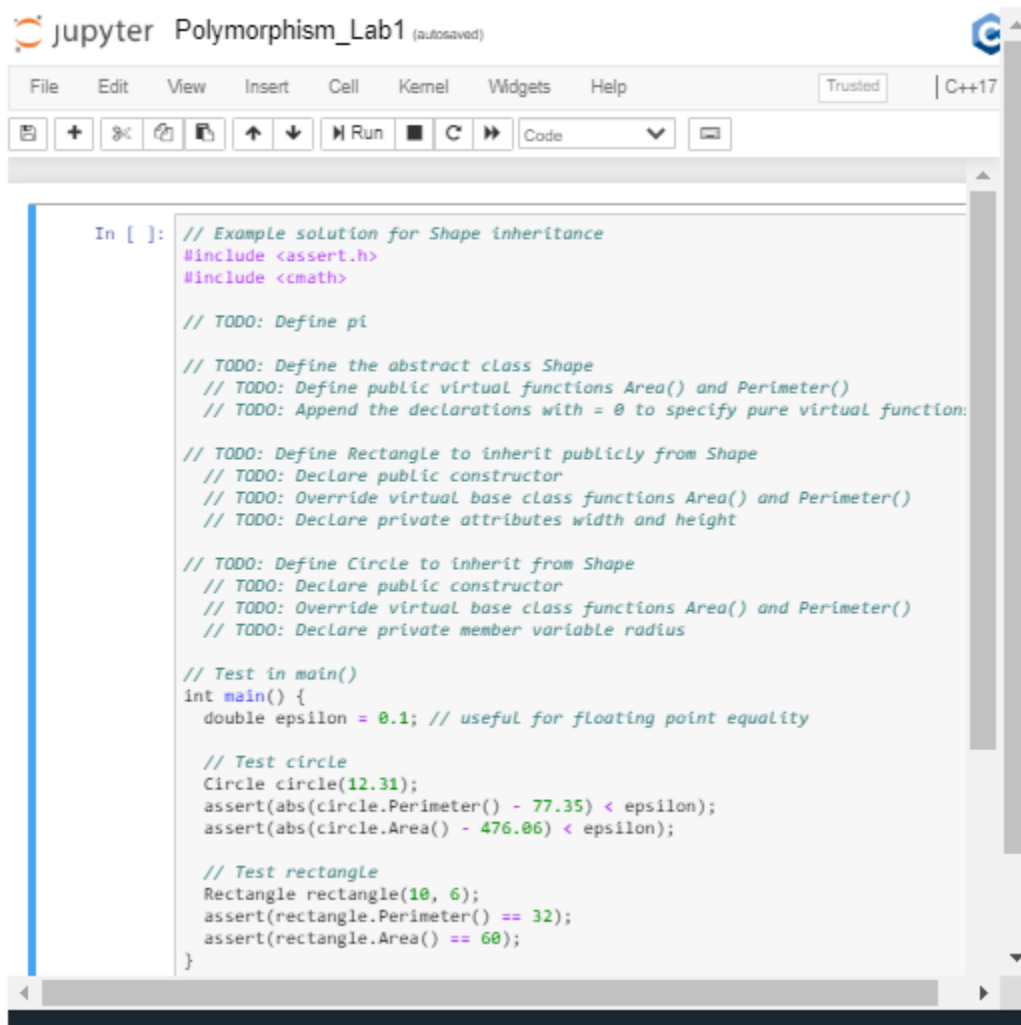
A pure virtual function has the side effect of making its class **abstract**. This means that the class cannot be instantiated. Instead, only classes that derive from the abstract class and override the pure virtual function can be instantiated.

```
class Shape {
public:
    Shape() {}
    virtual double Area() const = 0;
    virtual double Perimeter() const = 0;
};
```

Virtual functions can be defined by derived classes, but this is not required. However, if we mark the virtual function with `= 0` in the base class, then we are declaring the function to be a pure virtual function. This means that the base class does not define this function. A derived class must define this function, or else the derived class will be abstract.

Instructions

1. Create base class called `Shape`.
2. Define pure virtual functions (`= 0`) for the base class.
3. Write the derived classes.
 - Inherit from `class Shape`.
 - Override the pure virtual functions from the base class.
4. Test in `main()`



```
In [ ]: // Example solution for Shape inheritance
#include <assert.h>
#include <cmath>

// TODO: Define pi

// TODO: Define the abstract class Shape
// TODO: Define public virtual functions Area() and Perimeter()
// TODO: Append the declarations with = 0 to specify pure virtual function:

// TODO: Define Rectangle to inherit publicly from Shape
// TODO: Declare public constructor
// TODO: Override virtual base class functions Area() and Perimeter()
// TODO: Declare private attributes width and height

// TODO: Define Circle to inherit from Shape
// TODO: Declare public constructor
// TODO: Override virtual base class functions Area() and Perimeter()
// TODO: Declare private member variable radius

// Test in main()
int main() {
    double epsilon = 0.1; // useful for floating point equality

    // Test circle
    Circle circle(12.31);
    assert(abs(circle.Perimeter() - 77.35) < epsilon);
    assert(abs(circle.Area() - 476.06) < epsilon);

    // Test rectangle
    Rectangle rectangle(10, 6);
    assert(rectangle.Perimeter() == 32);
    assert(rectangle.Area() == 60);
}
```

12) Polymorphism: Overriding

<https://youtu.be/u15HcpiBeRc>

Polymorphism: Overriding

"**Overriding**" a function occurs when:

1. A base class declares a **virtual function**.
2. A derived class *overrides* that virtual function by defining its own implementation with an identical function signature (i.e. the same function name and argument types).

```
class Animal {
public:
    virtual std::string Talk() const = 0;
};

class Cat {
public:
    std::string Talk() const { return std::string("Meow"); }
};
```

In this example, `Animal` exposes a **virtual** function: `Talk()`, but does not define it.

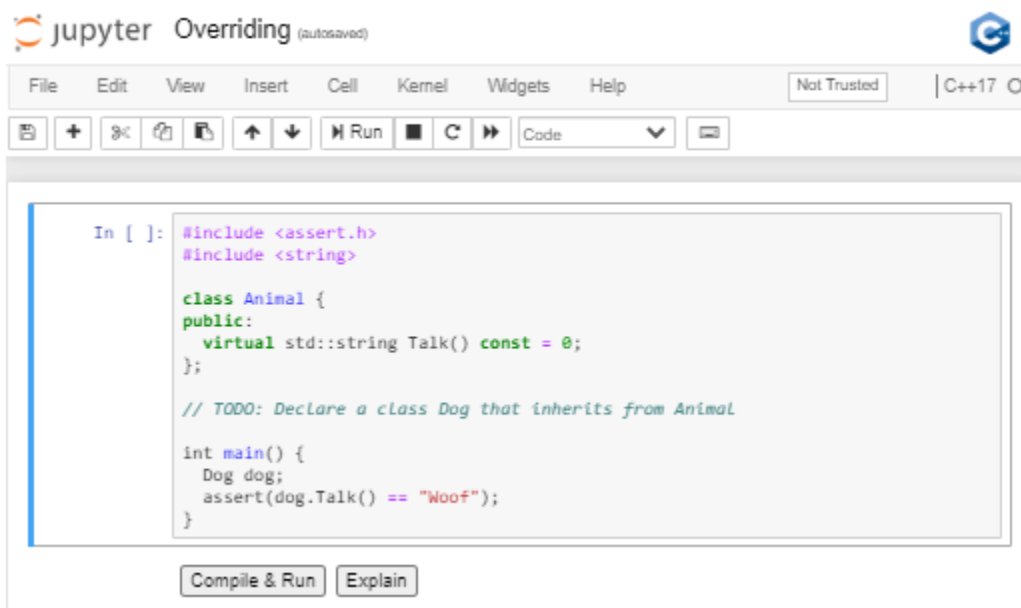
Because `Animal::Talk()` is undefined, it is called a **pure virtual function**, as opposed to an ordinary (impure? 😊) **virtual function**.

Furthermore, because `Animal` contains a pure virtual function, the user cannot instantiate an object of type `Animal`. This makes `Animal` an **abstract class**.

`Cat`, however, inherits from `Animal` and overrides `Animal::Talk()` with `Cat::Talk()`, which is defined. Therefore, it is possible to instantiate an object of type `Cat`.

Instructions

1. Create a class `Dog` to inherit from `Animal`.
2. Define `Dog::Talk()` to override the virtual function `Animal::Talk()`.
3. Confirm that the tests pass.



The image shows a Jupyter Notebook window titled "jupyter Overriding (autosaved)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a toolbar with icons for file operations and execution, and a status bar showing "Not Trusted" and "C++17". The code editor contains the following C++ code:

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

class Animal {
public:
    virtual std::string Talk() const = 0;
};

// TODO: Declare a class Dog that inherits from Animal

int main() {
    Dog dog;
    assert(dog.Talk() == "Woof");
}
```

At the bottom of the code editor, there are two buttons: "Compile & Run" and "Explain".

Function Hiding

Function hiding is **closely related, but distinct from**, overriding.

A derived class hides a base class function, as opposed to overriding it, if the base class function is not specified to be `virtual`.

```
class Cat { // Here, Cat does not derive from a base class
public:
    std::string Talk() const { return std::string("Meow"); }
};

class Lion : public Cat {
public:
    std::string Talk() const { return std::string("Roar"); }
};
```

In this example, `Cat` is the base class and `Lion` is the derived class.

Both `Cat` and `Lion` have `Talk()` member functions.

When an object of type `Lion` calls `Talk()`, the object will run `Lion::Talk()`, not `Cat::Talk()`.

In this situation, `Lion::Talk()` is *hiding* `Cat::Talk()`. If `Cat::Talk()` were `virtual`, then `Lion::Talk()` would *override* `Cat::Talk()`, instead of *hiding* it. *Overriding* requires a `virtual` function in the base class.

The distinction between *overriding* and *hiding* is subtle and not terribly significant, but in certain situations *hiding* **can lead to bizarre errors**, particularly when the two functions have slightly different function signatures.

13) Override

<https://youtu.be/C2DNR0Ao0VM>

Override

"Overriding" a function occurs when a derived class defines the implementation of a `virtual` function that it inherits from a base class.

It is possible, but not required, to specify a function declaration as `override`.

```
class Shape {
public:
    virtual double Area() const = 0;
    virtual double Perimeter() const = 0;
};

class Circle : public Shape {
public:
    Circle(double radius) : radius_(radius) {}
    double Area() const override { return pow(radius_, 2) * PI; } // specified as an override function
    double Perimeter() const override { return 2 * radius_ * PI; } // specified as an override function
};
```



```
private:
    double radius_;
};
```

This specification tells both the compiler and the human programmer that the purpose of this function is to override a virtual function. The compiler will verify that a function specified as `override` does indeed override some other virtual function, or otherwise the compiler will generate an error.

Specifying a function as `override` is [good practice](#), as it empowers the compiler to verify the code, and communicates the intention of the code to future users.

Exercise

In this exercise, you will build two [vehicle motion models](#), and override the `Move()` member function.

The first motion model will be `class ParticleModel`. In this model, the state is `x`, `y`, and `theta` (heading). The `Move(double v, double theta)` function for this model includes instantaneous steering:

```
theta += phi
x += v * cos(theta)
y += v * sin(theta)
```

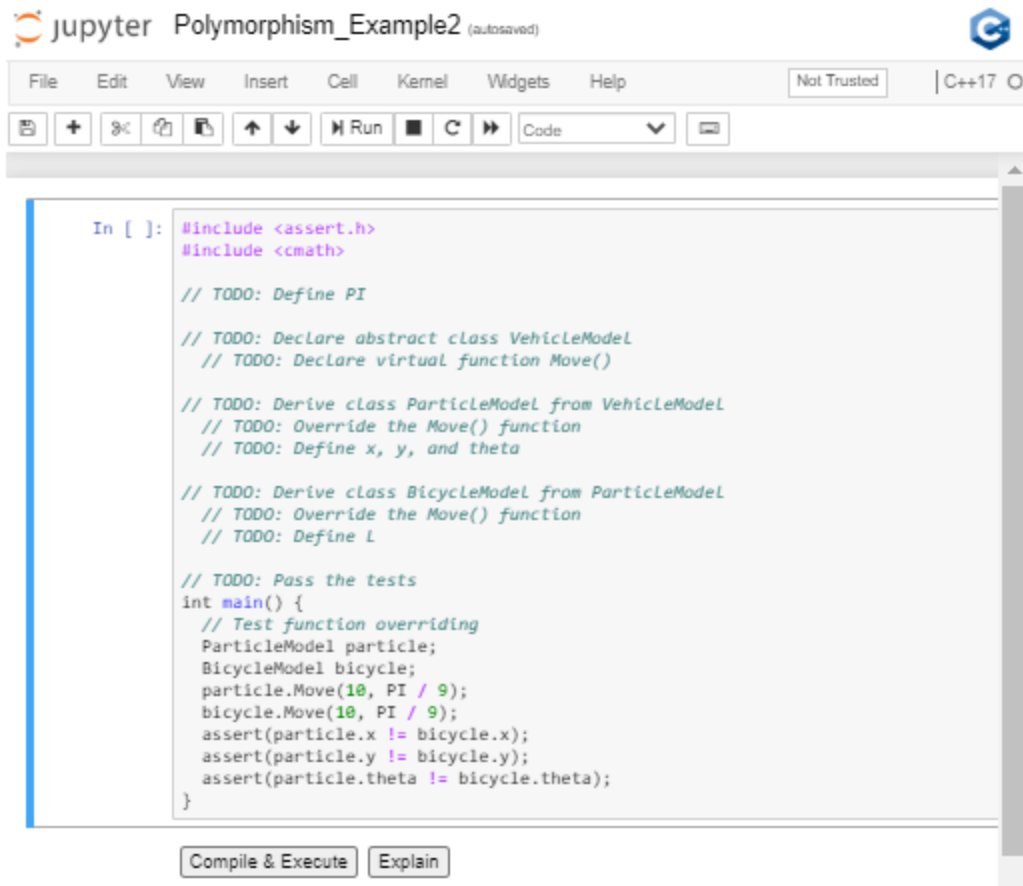
The second motion model will be `class BicycleModel`. In this model, the state is `x`, `y`, `theta` (heading), and `L` (the length of the vehicle). The `Move(double v, double theta)` function for this model is affected by the length of the vehicle:

```
theta += v / L * tan(phi)
x += v * cos(theta)
y += v * sin(theta)
```

You are encouraged to [read more](#) about vehicle motion, but for the purposes of practicing function overriding, the precise motion models are not so important. What is important is that the two models, and thus to the two `Move()` functions, are *different*.

Instructions

1. Define `class ParticleModel`, including its state and `Move()` function.
2. Extend `class BicycleModel` from `class ParticleModel`.
3. Override the `Move()` function within `class BicycleModel`.
4. Specify `BicycleModel::Move()` as `override`.
5. Pass the tests in `main()` by verifying that the two `Move()` functions override each other in different scenarios.



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

// TODO: Define PI

// TODO: Declare abstract class VehicleModel
// TODO: Declare virtual function Move()

// TODO: Derive class ParticleModel from VehicleModel
// TODO: Override the Move() function
// TODO: Define x, y, and theta

// TODO: Derive class BicycleModel from ParticleModel
// TODO: Override the Move() function
// TODO: Define L

// TODO: Pass the tests
int main() {
    // Test function overriding
    ParticleModel particle;
    BicycleModel bicycle;
    particle.Move(10, PI / 9);
    bicycle.Move(10, PI / 9);
    assert(particle.x != bicycle.x);
    assert(particle.y != bicycle.y);
    assert(particle.theta != bicycle.theta);
}
```

Compile & Execute Explain

14) Multiple Inheritance

<https://youtu.be/jEoPLBdLLsw>

Multiple Inheritance

In this exercise, you'll get some practical experience with multiple inheritance. If you have class `Animal` and another class `Pet`, then you can construct a class `Dog`, which inherits from both of these base classes. In doing this, you are able to incorporate attributes of multiple base classes.

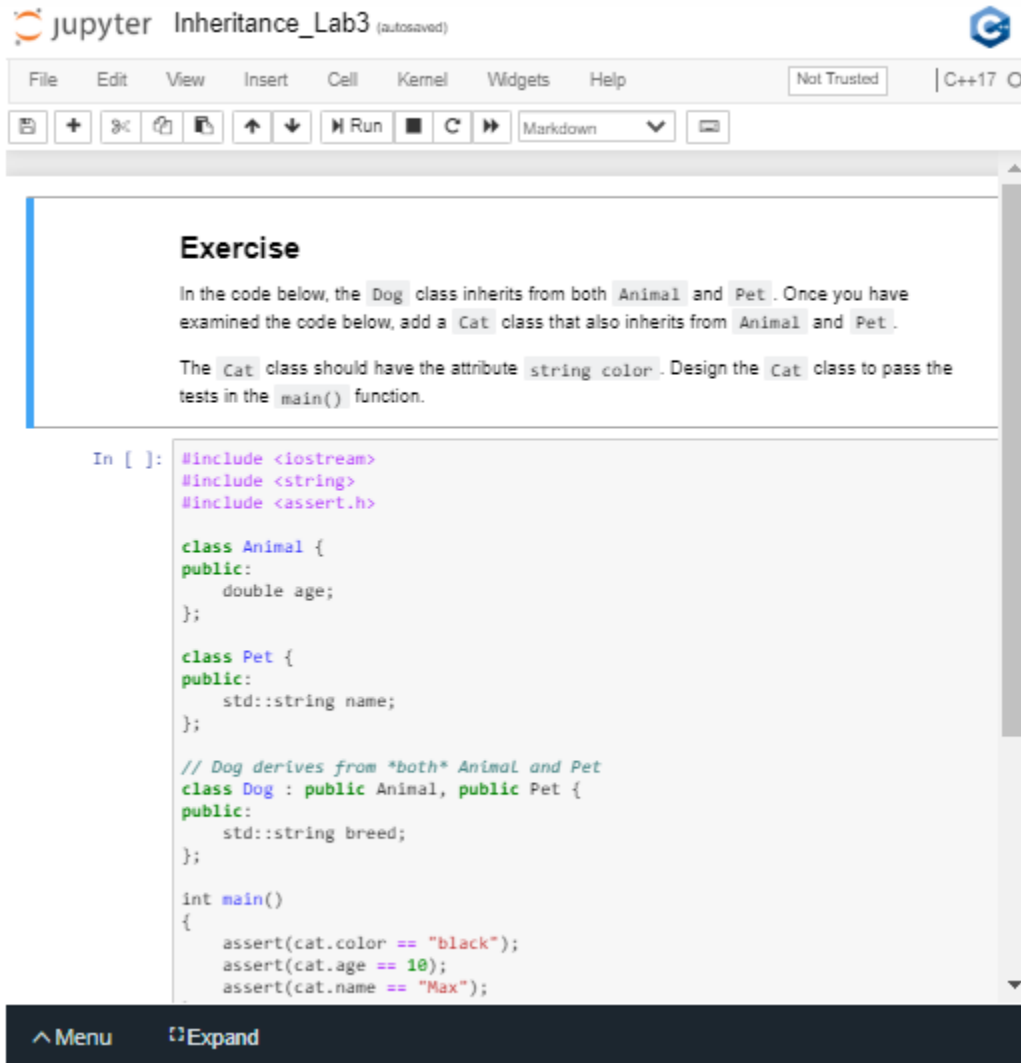
The Core Guidelines have some worthwhile recommendations about how and when to use multiple inheritance:

- "Use multiple inheritance to represent multiple distinct interfaces"
- "Use multiple inheritance to represent the union of implementation attributes"

Instructions

1. Review `class Dog`, which inherits from both `Animal` and `Pet`.

2. Declare a `class Cat`, with a member attribute `color`, that also inherits from both `Animal` and `Pet`.
3. Instantiate an object of `class Cat`.
4. Configure that object to pass the tests in `main()`.



The screenshot shows a JupyterLab window titled "Inheritance_Lab3 (autosaved)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running, and markdown. The main content area is divided into two sections. The top section, titled "Exercise", contains text explaining the task: to add a `Cat` class that inherits from both `Animal` and `Pet`, with a `color` attribute, and to pass tests in the `main()` function. The bottom section, labeled "In []:", contains C++ code. The code defines three classes: `Animal` (with a `double age` attribute), `Pet` (with a `std::string name` attribute), and `Dog` (which inherits from both `Animal` and `Pet`, with a `std::string breed` attribute). The `main()` function contains three assertions: `assert(cat.color == "black");`, `assert(cat.age == 10);`, and `assert(cat.name == "Max");`. The bottom of the window has a dark bar with "Menu" and "Expand" buttons.

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

class Animal {
public:
    double age;
};

class Pet {
public:
    std::string name;
};

// Dog derives from *both* Animal and Pet
class Dog : public Animal, public Pet {
public:
    std::string breed;
};

int main()
{
    assert(cat.color == "black");
    assert(cat.age == 10);
    assert(cat.name == "Max");
}
```

<https://youtu.be/p29phGPfKnQ>

15) Generic Programming

<https://youtu.be/k2Hai5sBemU>

16) Bjarne on Generic Programming

<https://youtu.be/m3a4ojP0dVQ>

17) Templates

<https://youtu.be/bUphr3EuM8A>

Templates

Templates enable generic programming by generalizing a function to apply to any class. Specifically, templates use *types* as parameters so that the same implementation can operate on different data types.

For example, you might need a function to accept many different data types. The function acts on those arguments, perhaps dividing them or sorting them or something else. Rather than writing and maintaining the multiple function declarations, each accepting slightly different arguments, you can write one function and pass the argument types as parameters. At compile time, the compiler then expands the code using the types that are passed as parameters.

```
template <typename Type> Type Sum(Type a, Type b) { return a + b; }  
  
int main() { std::cout << Sum<double>(20.0, 13.7) << "\n"; }
```

Because `Sum()` is defined with a template, when the program calls `Sum()` with `doubles` as parameters, the function expands to become:

```
double Sum(double a, double b) {  
    return a+b;  
}
```

Or in this case:

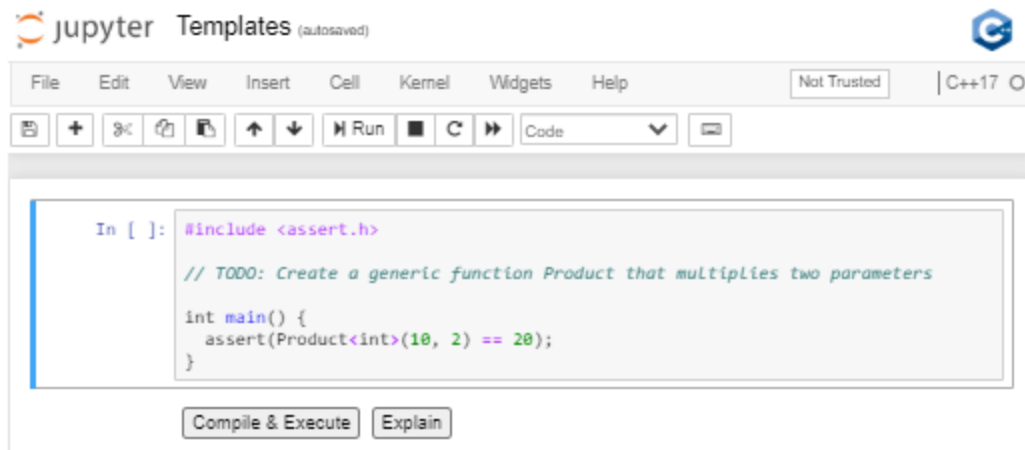
```
std::cout << Sum<char>('Z', 'j') << "\n";
```

The program expands to become:

```
char Sum(char a, char b) {  
    return a+b;  
}
```

We use the keyword `template` to specify which function is generic. Generic code is the term for code that is independent of types. It is mandatory to put the `template<>` tag before the function signature, to specify and mark that the declaration is generic.

Besides `template`, the keyword `typename` (or, alternatively, `class`) specifies the generic type in the function prototype. The parameters that follow `typename` (or `class`) represent generic types in the function declaration. In order to instantiate a templated class, use a templated constructor, for example: `Sum<double>(20.0, 13.7)`. You might recognize this form as the same form used to construct a `vector`. That's because `vector`s are indeed a generic class!



The image shows a Jupyter notebook titled "Templates" with a C++17 kernel. The code cell contains a C++ program that includes `<assert.h>` and defines a generic function `Product` that multiplies two parameters. The `main` function calls `assert(Product<int>(10, 2) == 20);`. Below the code cell are buttons for "Compile & Execute" and "Explain".

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

// TODO: Create a generic function Product that multiplies two parameters

int main() {
    assert(Product<int>(10, 2) == 20);
}
```

18) Bjarne on Templates

<https://youtu.be/tnOsS8JEO0U>

19) Exercise: Comparison Operation

Exercise: Comparison Operator

This exercise demonstrates how a simple comparison between two variables of unknown type can work using templates. In this case, by defining a template that performs a comparison using the `>` operator, you can compare two variables of any type (both variables must be of the same type, though) as long as the operator `>` is defined for that type.

Check out the notebook below to see how that works.



The image shows a Jupyter notebook titled "Templates_Lab1" with a C++17 kernel. The code cell contains a C++ program that includes `<assert.h>` and declares a generic, templated function `Max()`. The `main` function calls `assert(Max(10, 50) == 50);` and `assert(Max(5.7, 1.436246) == 5.7);`. Below the code cell are buttons for "Compile & Execute" and "Explain". At the bottom of the notebook, a terminal window shows the prompt `root@6581fa8d56c3:/home/workspace#`.

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

// TODO: Declare a generic, templated function Max()

int main() {
    assert(Max(10, 50) == 50);
    assert(Max(5.7, 1.436246) == 5.7);
}
```

```
root@6581fa8d56c3:/home/workspace#
```

20) Deduction

Deduction

In this example, you will see the difference between total and partial **deduction**. Deduction occurs when you instantiate an object without explicitly identifying the types. Instead, the compiler "deduces" the types. This can be helpful for writing code that is generic and can handle a variety of inputs.

In this exercise, we will use templates to overload the '#' operator to average two numbers.

Instructions

1. Use a template to overload the # operator.
2. Confirm that the tests pass.

<https://youtu.be/JJLGNIQ1QLk>



21) Exercise: Class Templates

Exercise: Class Template

Classes are the building blocks of object oriented programming in C++. Templates support the creation of generic classes!

Class templates can declare and implement generic attributes for use by generic methods. These templates can be very useful when building classes that will serve multiple purposes.

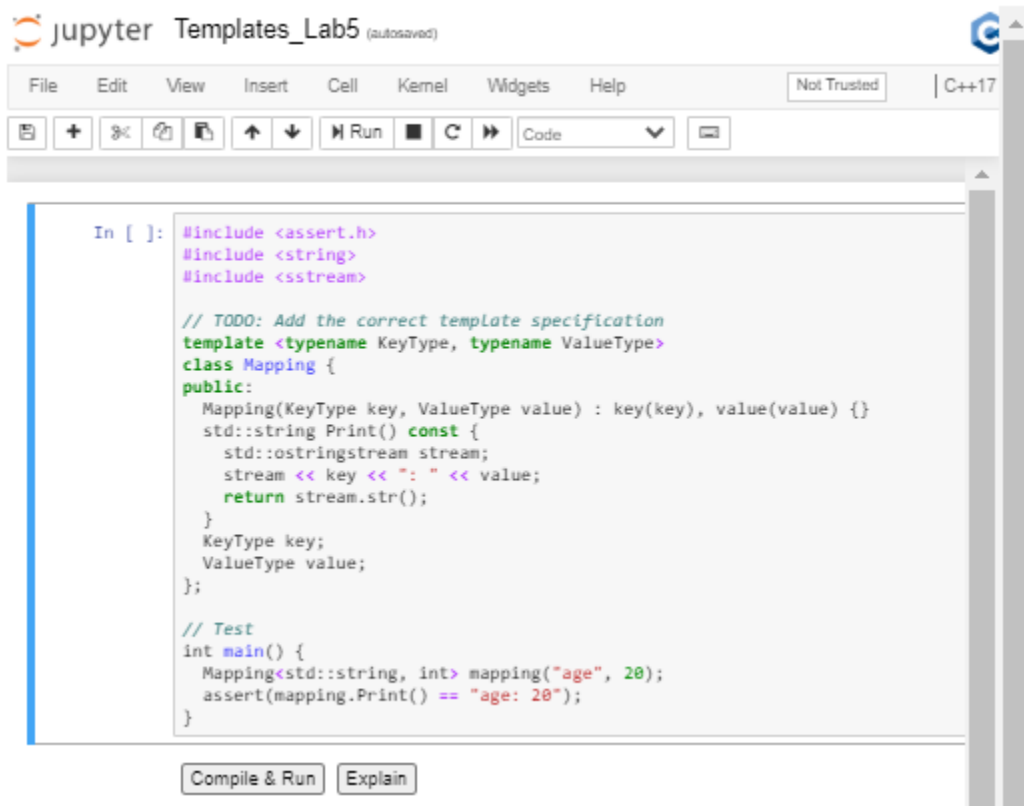
In this exercise you will create a `class Mapping` that maps a generic key to a generic value.

All of the code has been written for you, except the initial template specification.

In order for this template specification to work, you will need to include two generic types: `KeyName` and `ValueName`. Can you imagine how to do that?

Instructions

1. Write the template specification.
2. Verify that the test passes.



The screenshot shows a JupyterLab window titled "jupyter Templates_Lab5 (autosaved)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help), a status bar (Not Trusted, C++17), and a toolbar with icons for file operations, running, and code execution. The main area is a code editor with the following C++ code:

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

// TODO: Add the correct template specification
template <typename KeyType, typename ValueType>
class Mapping {
public:
    Mapping(KeyType key, ValueType value) : key(key), value(value) {}
    std::string Print() const {
        std::ostringstream stream;
        stream << key << ": " << value;
        return stream.str();
    }
    KeyType key;
    ValueType value;
};

// Test
int main() {
    Mapping<std::string, int> mapping("age", 20);
    assert(mapping.Print() == "age: 20");
}
```

At the bottom of the code editor, there are two buttons: "Compile & Run" and "Explain".

22) Summary

<https://youtu.be/QR68Vcr-XTw>

23) Bjarne on Best Practices with Classes

<https://youtu.be/gWcAMxhNOcg>

Project: System Monitor	
CH	
URCES	
CEPTS	
1. Introduction	
2. htop	
3. Starter Code	
4. Project Structure	
5. Build Tools	
6. System Class	
7. System Data	
8. LinuxParser Namespace	
9. String Parsing	
10. Processor Class	
11. Processor Data	
12. Process Class	
13. Process Data	
14. Goal	
15. Project Workspace	
16. Project: System Monitor	

1) Introduction

Updates!

Udacity has updated and improved the System Monitor Project!

These updates help organize and clarify the project.

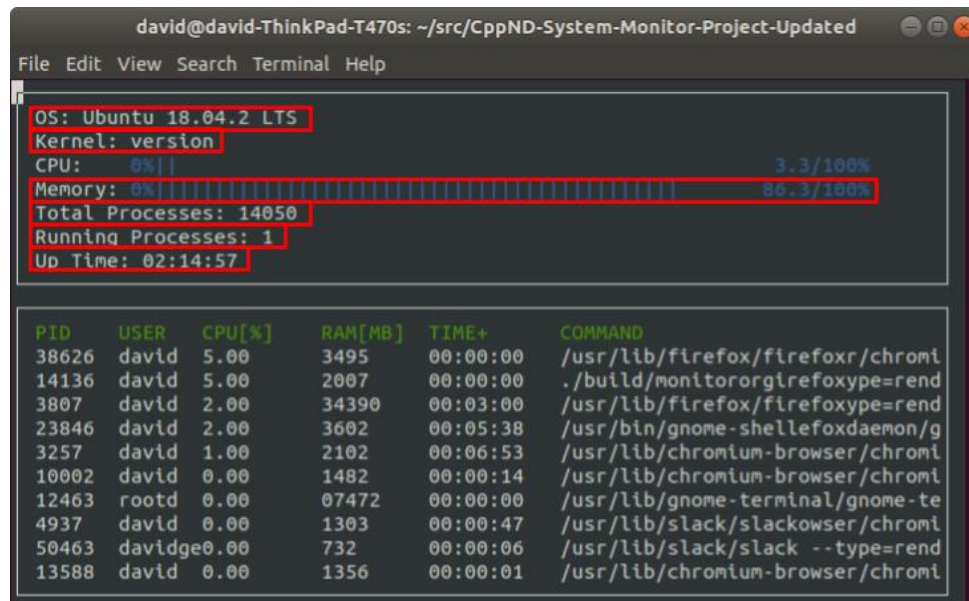
The new project version is available right now in this lesson.

Students who have already completed the System Monitor Project are all set! There is no requirement to revisit this lesson and complete the new version of the project.

<https://youtu.be/EbgJYBZ4QDA>

- 2) Htop
<https://youtu.be/Cz4rDC-WecA>
- 3) Starter Code
<https://youtu.be/eguBVmzhTS4>
- 4) Project Structure
<https://youtu.be/dOnUD8UUhMg>
https://youtu.be/10HWAXzY_90
- 5) Build Tools
<https://youtu.be/PSPI33rKQas>
- 6) System Class
<https://youtu.be/M6tpsAZWnjI>
- 7) System Data

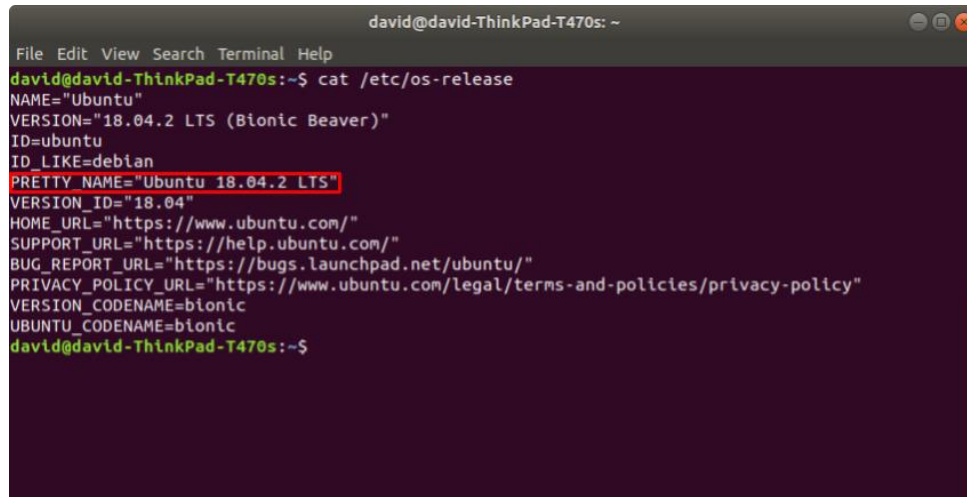
System Data



Linux stores a lot of system data in files within the `/proc` directory. Most of the data that this project requires exists in those files.

Operating System

Information about the operating system exists outside of the `/proc` directory, in the `/etc/os-release` file.

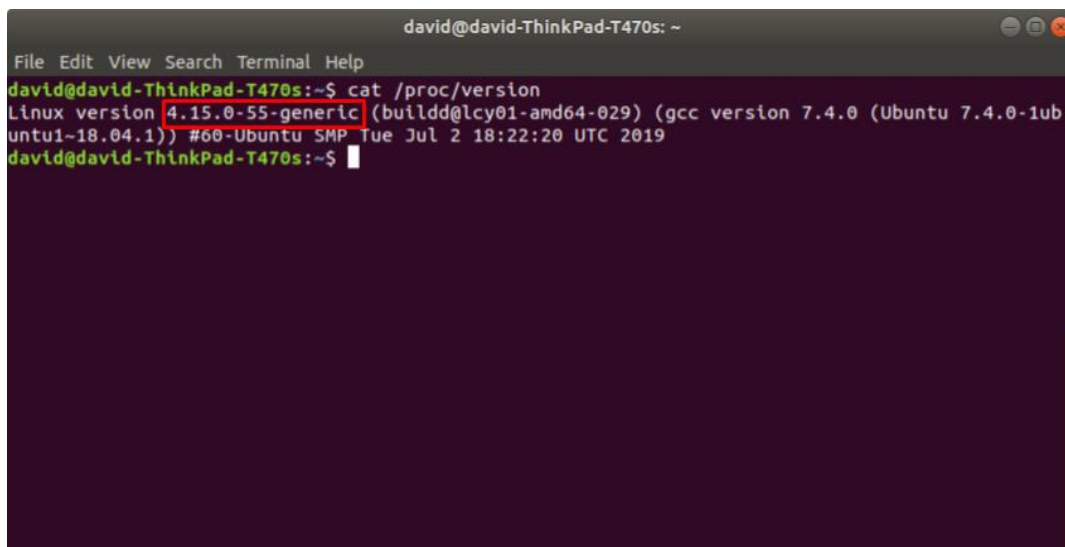
A terminal window titled 'david@david-ThinkPad-T470s: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The command 'cat /etc/os-release' has been executed, displaying the following output: NAME="Ubuntu", VERSION="18.04.2 LTS (Bionic Beaver)", ID=ubuntu, ID_LIKE=debian, PRETTY_NAME="Ubuntu 18.04.2 LTS", VERSION_ID="18.04", HOME_URL="https://www.ubuntu.com/", SUPPORT_URL="https://help.ubuntu.com/", BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/", PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy", VERSION_CODENAME=bionic, UBUNTU_CODENAME=bionic. The 'PRETTY_NAME' line is highlighted with a red box.

```
david@david-ThinkPad-T470s: ~  
File Edit View Search Terminal Help  
david@david-ThinkPad-T470s:~$ cat /etc/os-release  
NAME="Ubuntu"  
VERSION="18.04.2 LTS (Bionic Beaver)"  
ID=ubuntu  
ID_LIKE=debian  
PRETTY_NAME="Ubuntu 18.04.2 LTS"  
VERSION_ID="18.04"  
HOME_URL="https://www.ubuntu.com/"  
SUPPORT_URL="https://help.ubuntu.com/"  
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"  
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"  
VERSION_CODENAME=bionic  
UBUNTU_CODENAME=bionic  
david@david-ThinkPad-T470s:~$
```

There are several strings from which to choose here, but the most obvious is the value specified by "PRETTY_NAME".

Kernel

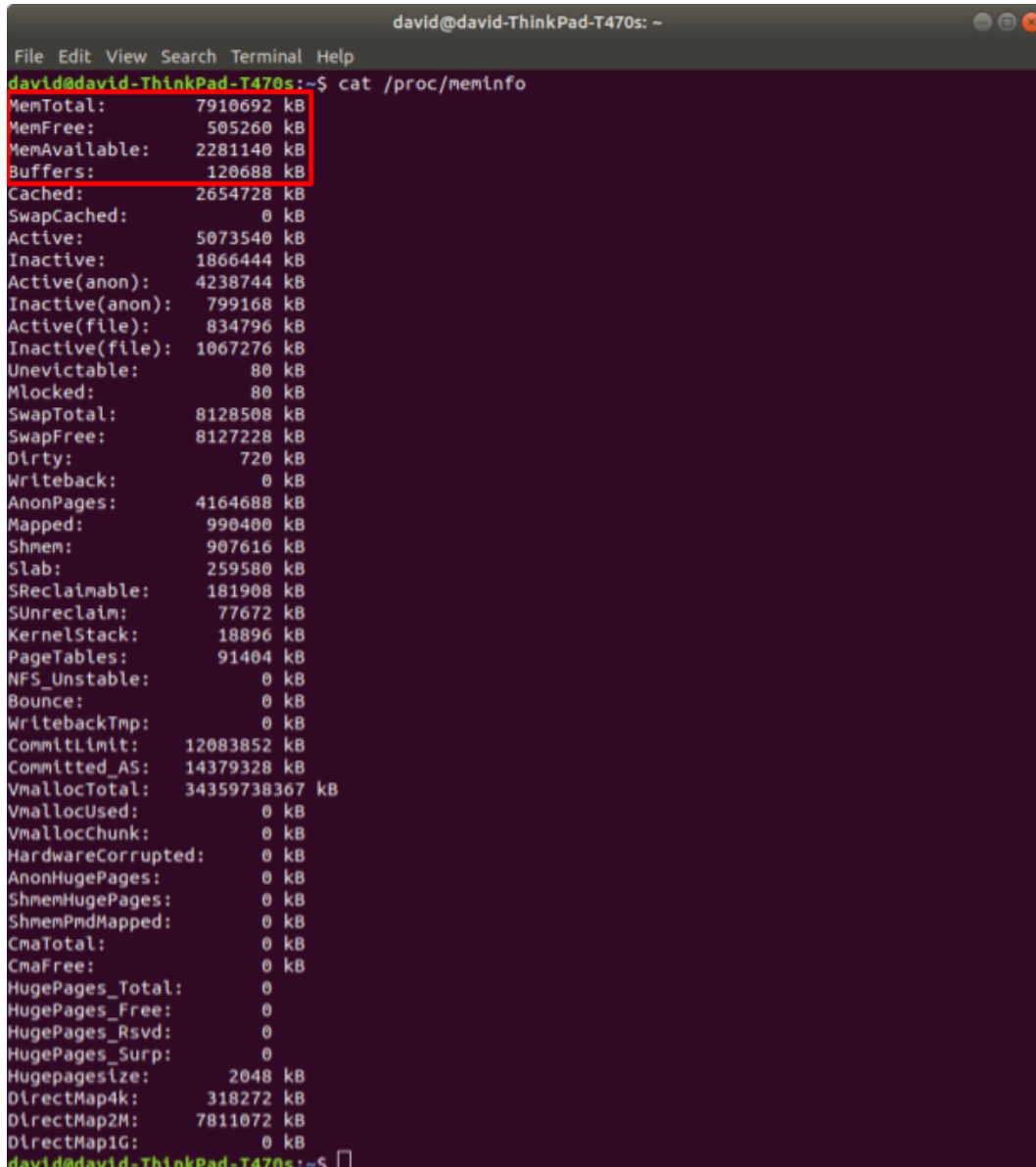
Information about the kernel exists `/proc/version` file.

A terminal window titled 'david@david-ThinkPad-T470s: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The command 'cat /proc/version' has been executed, displaying the following output: Linux version 4.15.0-55-generic (buildd@lcy01-amd64-029) (gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1-18.04.1)) #60-Ubuntu SMP Tue Jul 2 18:22:20 UTC 2019. The version string '4.15.0-55-generic' is highlighted with a red box.

```
david@david-ThinkPad-T470s: ~  
File Edit View Search Terminal Help  
david@david-ThinkPad-T470s:~$ cat /proc/version  
Linux version 4.15.0-55-generic (buildd@lcy01-amd64-029) (gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1-18.04.1)) #60-Ubuntu SMP Tue Jul 2 18:22:20 UTC 2019  
david@david-ThinkPad-T470s:~$
```

Memory Utilization

Information about memory utilization exists in the `/proc/meminfo` file.



```
david@david-ThinkPad-T470s: ~  
File Edit View Search Terminal Help  
david@david-ThinkPad-T470s:~$ cat /proc/meminfo  
MemTotal:       7910692 kB  
MemFree:        505260 kB  
MemAvailable:   2281140 kB  
Buffers:        120688 kB  
Cached:         2654728 kB  
SwapCached:      0 kB  
Active:         5073540 kB  
Inactive:       1866444 kB  
Active(anon):   4238744 kB  
Inactive(anon): 799168 kB  
Active(file):   834796 kB  
Inactive(file): 1067276 kB  
Unevictable:    80 kB  
Mlocked:        80 kB  
SwapTotal:      8128508 kB  
SwapFree:       8127228 kB  
Dirty:          720 kB  
Writeback:      0 kB  
AnonPages:      4164688 kB  
Mapped:         990400 kB  
Shmem:          907616 kB  
Slab:           259580 kB  
SReclaimable:   181908 kB  
SUnreclaim:     77672 kB  
KernelStack:   18896 kB  
PageTables:     91404 kB  
NFS_Unstable:   0 kB  
Bounce:         0 kB  
WritebackTmp:   0 kB  
CommitLimit:   12083852 kB  
Committed_AS:  14379328 kB  
VmallocTotal:  34359738367 kB  
VmallocUsed:    0 kB  
VmallocChunk:   0 kB  
HardwareCorrupted: 0 kB  
AnonHugePages:  0 kB  
ShmemHugePages: 0 kB  
ShmemPmdMapped: 0 kB  
CmaTotal:       0 kB  
CmaFree:        0 kB  
HugePages_Total: 0  
HugePages_Free: 0  
HugePages_Rsvd: 0  
HugePages_Surp: 0  
Hugepagesize:   2048 kB  
DirectMap4k:    318272 kB  
DirectMap2M:    7811072 kB  
DirectMap1G:    0 kB  
david@david-ThinkPad-T470s:~$
```

There are a [variety](#) of [ways](#) to use this data to calculate memory utilization. [Hisham H. Muhammad](#), the author of [htop](#), wrote a [Stack Overflow answer](#) about how htop calculates memory utilization from the data in `/proc/meminfo`. Use the formula that makes the most sense to you!

Total Processes

Information about the total number of processes on the system exists in the `/proc/meminfo` file.

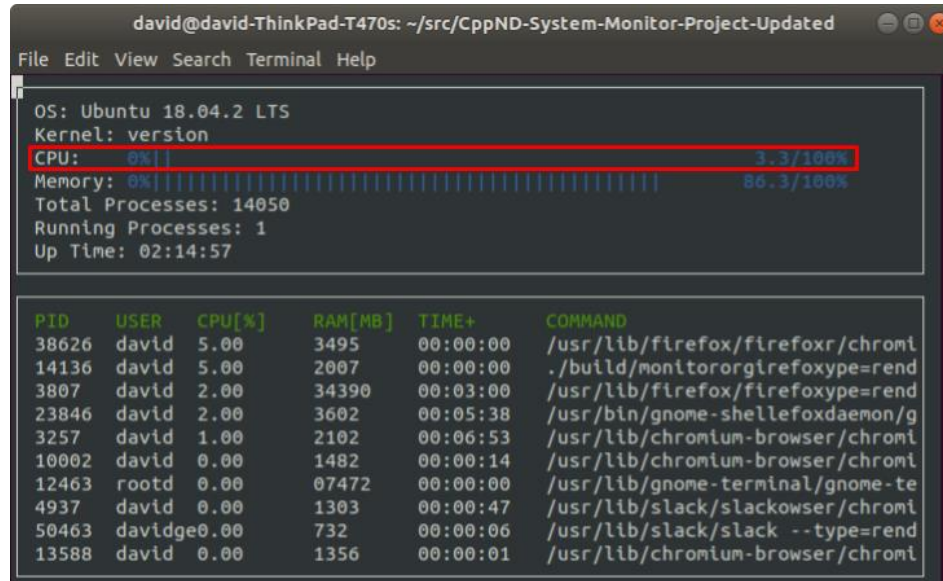
[illegible]

Running Processes

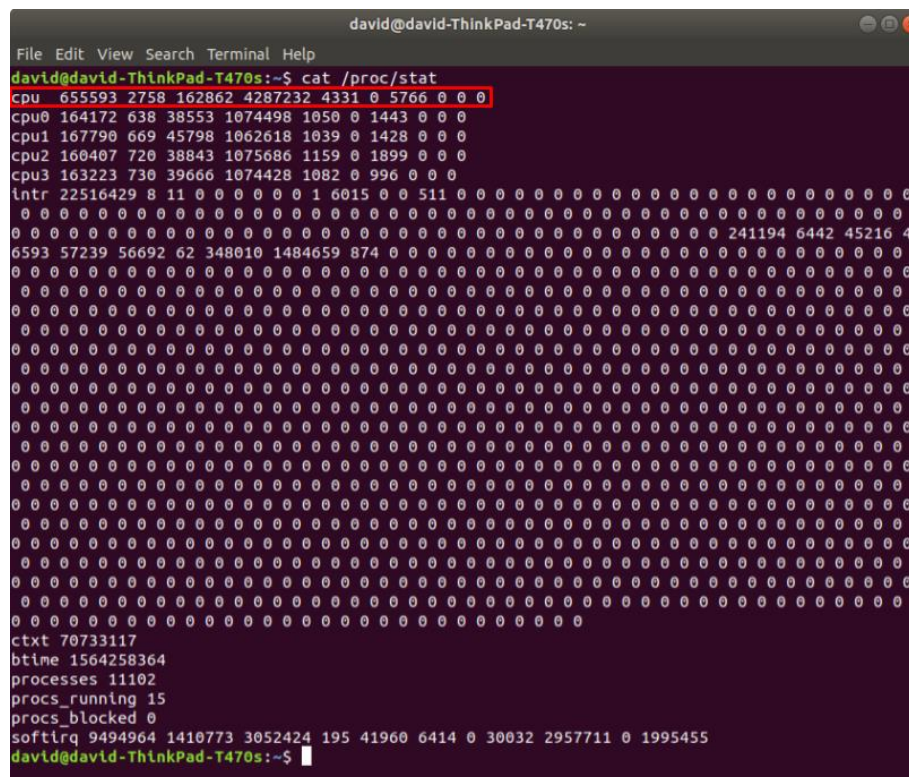
Information about the number of processes on the system that are currently running exists in the `/proc/meminfo` file.

11) Processor Data

Processor Data



Linux stores processor utilization data within the `/proc/stat` file.



For example, `/proc/stat` contains aggregate processor information (on the "cpu" line) and individual processor information (on the "cpu0", "cpu1", etc. lines). Indeed, [htop](#) displays utilization information for each individual processor.



Data

The very first "cpu" line aggregates the numbers in all of the other "cpuN" lines. These numbers identify the amount of time the CPU has spent performing different kinds of work. Time units are in USER_HZ (typically hundredths of a second). The meanings of the columns are as follows, from left to right:

- *user*: normal processes executing in user mode
- *nice*: niced processes executing in user mode
- *system*: processes executing in kernel mode
- *idle*: twiddling thumbs

- *iowait*: In a word, *iowait* stands for waiting for I/O to complete. But there are several problems:
 1. *Cpu* will not wait for I/O to complete, *iowait* is the time that a task is waiting for I/O to complete. When *cpu* goes into idle state for outstanding task io, another task will be scheduled on this CPU.
 2. In a multi-core CPU, the task waiting for I/O to complete is not running on any CPU, so the *iowait* of each CPU is difficult to calculate.
 3. The value of *iowait* field in `/proc/stat` will decrease in certain conditions. So, the *iowait* is not reliable by reading from `/proc/stat`.
- *irq*: servicing interrupts
- *softirq*: servicing softirqs
- *steal*: involuntary wait
- *guest*: running a normal guest
- *guest_nice*: running a niced guest

Even once you know what each of these numbers represents, it's still a challenge to determine exactly how to use these figures to calculate processor utilization. [This guide](#) and [this StackOverflow post](#) are helpful.

Measurement Interval

Once you've parsed `/proc/stat` and calculated the processor utilization, you've got what you need for this project. Congratulations!

However, when you run your system monitor, you might notice that the process utilization seems very stable. Too stable.

That's because the processor data in `/proc/stat` is measured since boot. If the system has been up for a long time, a temporary interval of even extreme system utilization is unlikely to change the long-term average statistics very much. This means that the processor could be red-lining *right now* but the system monitor might still show a relatively underutilized processor, if the processor has spent most of the time since boot in an idle state.

You might want to update the system monitor to report the current utilization of the processor, rather than the long-term average utilization since boot. You would need to measure the difference in system utilization between two points in time relatively close to the present. A formula like:

$\Delta \text{ active time units} / \Delta \text{ total time units}$

Consider this a bonus challenge that is not required to pass the project.

NEXT

<https://youtu.be/sEkf6TqLKBk>

13) Process Data

Process Data

```
david@david-ThinkPad-T470s: ~/src/CppND-System-Monitor-Project-Updated
File Edit View Search Terminal Help

OS: Ubuntu 18.04.2 LTS
Kernel: version
CPU: 0%| 3.3/100%
Memory: 0%| 86.3/100%
Total Processes: 14050
Running Processes: 1
Up Time: 02:14:57

PID USER CPU[%] RAM[MB] TIME+ COMMAND
38626 david 5.00 3495 00:00:00 /usr/lib/firefox/firefoxr/chromi
14136 david 5.00 2007 00:00:00 ./build/monitororgirefoxtype=rend
3807 david 2.00 34390 00:03:00 /usr/lib/firefox/firefoxtype=rend
23846 david 2.00 3602 00:05:38 /usr/bin/gnome-shellefoxdaemon/g
3257 david 1.00 2102 00:06:53 /usr/lib/chromium-browser/chromi
10002 david 0.00 1482 00:00:14 /usr/lib/chromium-browser/chromi
12463 rootd 0.00 07472 00:00:00 /usr/lib/gnome-terminal/gnome-te
4937 david 0.00 1303 00:00:47 /usr/lib/slack/slackowser/chromi
50463 davidge0.00 732 00:00:06 /usr/lib/slack/slack --type=rend
13588 david 0.00 1356 00:00:01 /usr/lib/chromium-browser/chromi
```

Linux stores data about individual processes in files within subdirectories of the `/proc` directory. Each subdirectory is named for that particular process's [identifier](#) number. The data that this project requires exists in those files.

PID

The process identifier (PID) is accessible from the `/proc` directory. Typically, all of the subdirectories of `/proc` that have integral names correspond to processes. Each integral name corresponds to a process ID.

```
david@david-ThinkPad-T470s: ~
File Edit View Search Terminal Help

david@david-ThinkPad-T470s:~$ ls /proc/
1 1260 13440 2278 2621 2831 3577 408 831 99 loadavg
10 1263 13459 2353 2625 2857 358 409 832 9941 locks
100 1265 13467 2354 2630 287 3581 41 841 9950 mdstat
1010 1278 13477 2367 2639 2876 3584 414 843 acpi meminfo
10107 1279 13486 2371 2640 2879 3590 419 851 asound misc
1015 1281 13552 2373 2642 2900 360 42 8805 buddyinfo modules
10165 1283 13553 24 2644 2906 363 43 8809 bus mounts
10185 1286 13585 240 2649 2995 364 44 8838 cgroups mtrr
1032 1291 13748 2405 2650 30 365 443 8859 cmdline net
10400 1293 13757 2408 2660 3001 366 445 896 consoles pagetypeinfo
105 1294 13762 245 2664 3048 369 446 9 cpuinfo partitions
1055 12979 13851 246 2668 3059 370 447 901 crypto sched_debug
1057 13 13855 2495 2672 31 371 45 903 devices schedstat
1077 1301 13896 2498 2675 312 372 46 905 diskstats scsi
1078 1305 14 25 2678 314 373 48 906 dma self
1090 1307 1412 2503 2682 315 374 49 907 driver slabinfo
1092 1308 15 2505 2683 317 38 50 909 execdomains softirqs
1094 1311 1524 2519 2687 32 380 51 911 fb stat
11 1312 1530 2524 2688 3211 384 52 913 filesystems swaps
1105 1313 1534 2542 2694 3249 387 55 9619 fs sys
11273 13166 16 2552 27 3258 389 56 967 interrupts sysrq-trigger
1135 1318 18 2564 2734 3264 39 57 9678 lsm sysvipc
114 132 19 2568 2736 3290 3926 6 9682 ioports thread-self
11966 1322 190 2570 2746 33 395 6249 9713 irq timer_list
12 1323 199 2573 2758 3341 398 7 9726 kallsyms tty
1221 13246 2 2583 2773 335 4 707 9759 kcore uptime
1229 13247 20 2587 2788 3390 40 709 9774 keys version
1234 13265 200 2595 2790 34 400 7370 9775 key-users version_signature
1236 13276 201 26 28 3475 401 8 9824 kmsg vmallocinfo
1240 1329 21 2606 2800 3478 402 825 9838 kpagecgroup vmstat
1241 13307 2187 2613 2805 3493 4035 828 9864 kpagecount zoneinfo
1257 1339 22 2617 2819 357 4036 829 9898 kpageflags
```

Parsing directory names with C++ is tricky, so we have provided in the project starter code a pre-implemented function to capture the PIDs.

User

Each process has an associated [user identifier \(UID\)](#), corresponding to the process owner. This means that determining the process owner requires two steps:

1. Find the UID associated with the process
2. Find the user corresponding to that UID

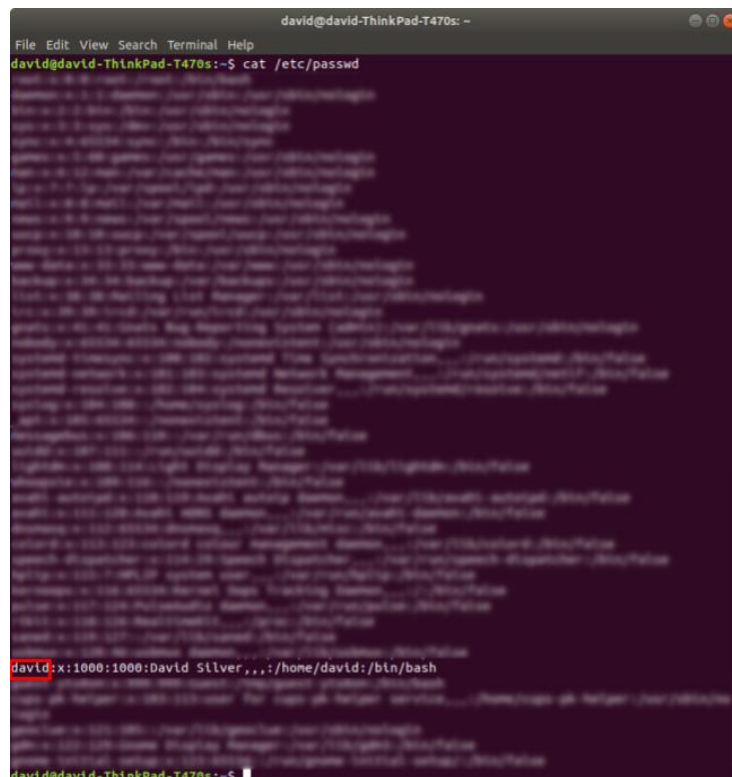
The UID for a process is stored in `/proc/[PID]/status`.

[illegible]

The [man page for proc](#) contains a `"/proc/[pid]/status"` section that describes this file. For the purposes of this project, you simply need to capture the first integer on the `"Uid:"` line.

Username

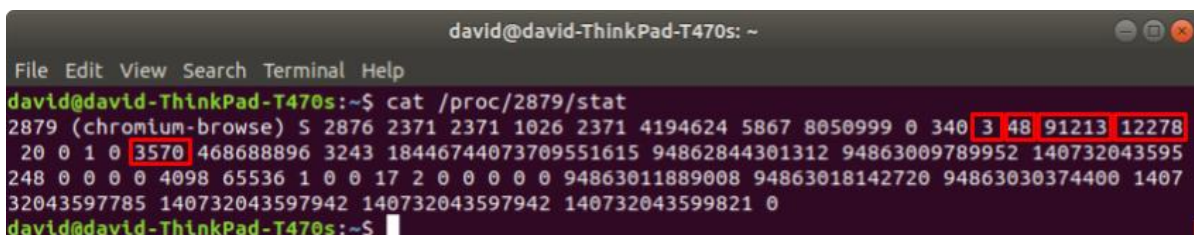
`/etc/passwd` contains the information necessary to match the UID to a username.



```
david@david-ThinkPad-T470s: ~  
File Edit View Search Terminal Help  
david@david-ThinkPad-T470s:~$ cat /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
bin:x:1:1:bin:/bin:/usr/sbin/nologin  
daemon:x:2:2:daemon:/usr/sbin:/usr/sbin/nologin  
...  
david:x:1000:1000:David Silver,,,:/home/david:/bin/bash  
...  
david@david-ThinkPad-T470s:~$
```

Processor Utilization

Linux stores the CPU utilization of a process in the `/proc/[PID]/stat` file.



```
david@david-ThinkPad-T470s: ~  
File Edit View Search Terminal Help  
david@david-ThinkPad-T470s:~$ cat /proc/2879/stat  
2879 (chromium-browser) S 2876 2371 2371 1026 2371 4194624 5867 8050999 0 340 3 48 91213 12278  
20 0 1 0 3570 468688896 3243 18446744073709551615 94862844301312 94863009789952 140732043595  
248 0 0 0 0 4098 65536 1 0 0 17 2 0 0 0 0 94863011889008 94863018142720 94863030374400 1407  
32043597785 140732043597942 140732043597942 140732043599821 0  
david@david-ThinkPad-T470s:~$
```

Much like the calculation of aggregate processor utilization, half the battle is extracting the relevant data from the file, and the other half of the battle is figuring out how to use those numbers to calculate processor utilization.

The `/proc/[pid]/stat` section of the `proc` [man page](#) describes the meaning of the values in this file. [This StackOverflow answer](#) explains how to use this data to calculate the process's utilization.

As with the calculation of aggregate processor utilization, it is sufficient for this project to calculate the average utilization of each process since the process launched. If you would like to extend your project to calculate a more current measurement of process utilization, we encourage you to do that!

Memory Utilization

Linux stores memory utilization for the process in `/proc/[pid]/status`.


```
david@david-ThinkPad-T470s: ~
File Edit View Search Terminal Help
david@david-ThinkPad-T470s:~$ cat /proc/2879/stat
2879 (chromium-browser) S 2876 2371 1026 2371 4194624 5867 8050999 0 340 3 48 91213 12278
20 0 1 0 3570 468688896 3243 18446744073709551615 94862844301312 94863009789952 140732043595
248 0 0 0 0 4098 65536 1 0 0 17 2 0 0 0 0 94863011889008 94863018142720 94863030374400 1407
32043597785 140732043597942 140732043597942 140732043599821 0
david@david-ThinkPad-T470s:~$
```

The `/proc/[pid]/stat` section of the [proc man page](#) describes each of the values in this file.

(22) `starttime %llu`

The time the process started after system boot. In kernels before Linux 2.6, this value was expressed in jiffies. Since Linux 2.6, the value is expressed in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).

Note that the "starttime" value in this file is measured in "clock ticks". In order to convert from "clock ticks" to seconds, you must:

- `#include <unistd.h>`
- divide the "clock ticks" value by `sysconf(_SC_CLK_TCK)`

Once you have converted the time value to seconds, you can use the `Format::Time()` function from the project starter code to display the seconds in a "HH:MM:SS" format.

Command

Linux stores the command used to launch the function in the `/proc/[pid]/cmdline` file.

```
david@david-ThinkPad-T470s: ~
File Edit View Search Terminal Help
david@david-ThinkPad-T470s:~$ cat /proc/2879/cmdline
/usr/lib/chromium-browser/chromium-browser --type=zygote --ppapi-flash-path=/usr/lib/adobe-fl
david@david-ThinkPad-T470s:~$
```

14) Goal

https://youtu.be/xw6_Mz3O54Y

15) Project workspace

Udacity Workspace

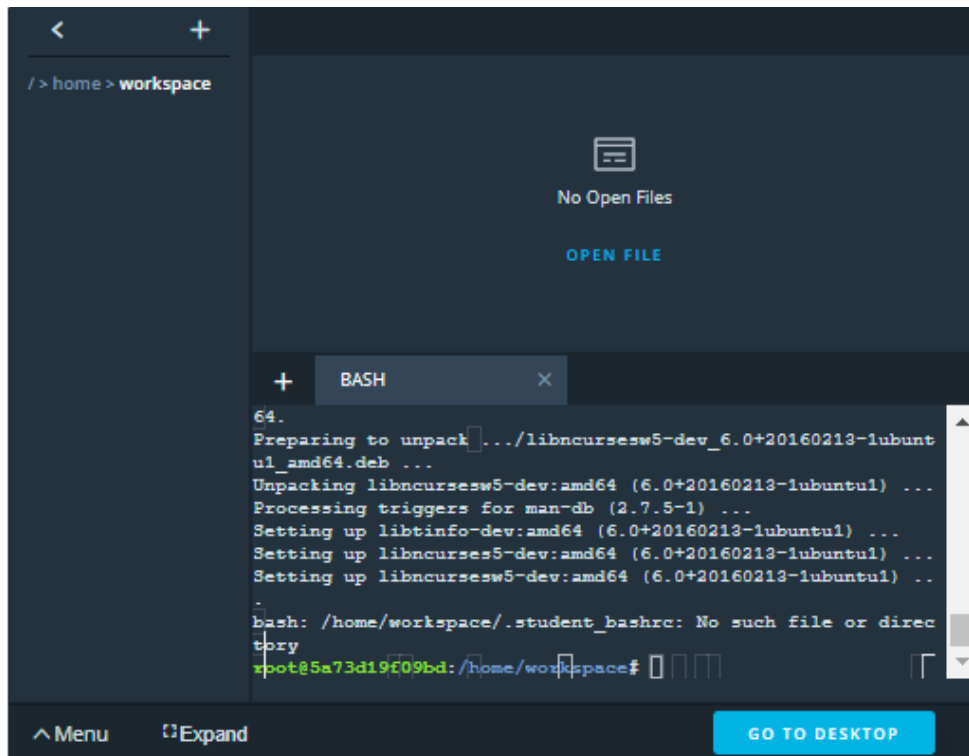
You are welcome to use this Udacity Workspace to complete the project. Or you can use your own Linux development environment, if you have one.

GitHub Repo

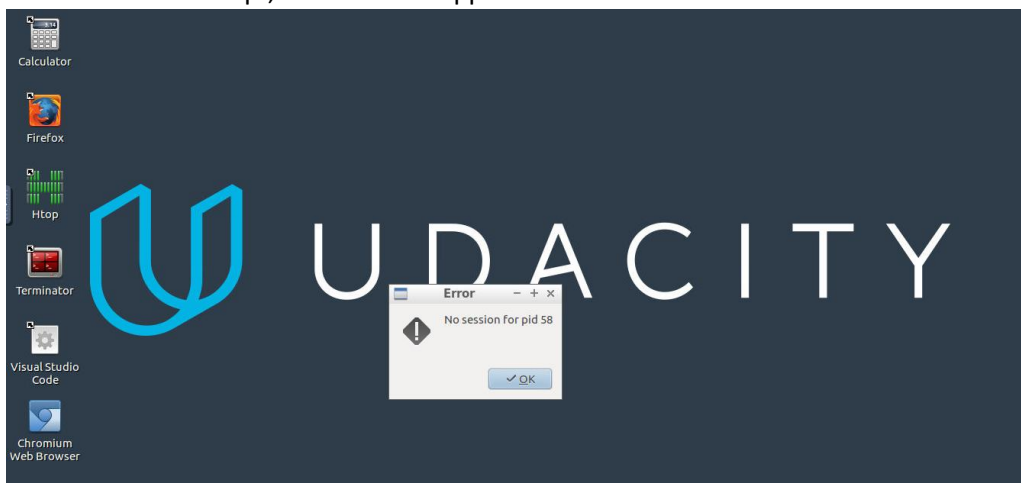
The starter code is available on GitHub: <https://github.com/udacity/CppND-System-Monitor>

You can clone the starter code repository, either into the Udacity Workspace or into your own development environment, by running:

```
git clone https://github.com/udacity/CppND-System-Monitor
```



When u click desktop , this is what happens



16) Project: system monitor

Project Submission

DUE DATE

Jun 23

STATUS

Not submitted

Due at: Tue, Jun 23 3:00 pm

CppND-System-Monitor

Starter code for System Monitor Project is provided on

GitHub: <https://github.com/udacity/CppND-System-Monitor-Project-Updated>

Follow along with the classroom lesson to complete the project!

Udacity Linux Workspace

Udacity provides a browser-based Linux [Workspace](#) for students.

You are welcome to develop this project on your local machine, and you are not required to use the Udacity Workspace. However, the Workspace provides a convenient and consistent Linux development environment we encourage you to try.

ncurses

[ncurses](#) is a library that facilitates text-based graphical output in the terminal. This project relies on ncurses for display output.

Within the Udacity Workspace, `.student_bashrc` automatically installs ncurses every time you launch the Workspace.

If you are not using the Workspace, install ncurses within your own Linux environment: `sudo apt install libncurses5-dev libncursesw5-dev`

Make

This project uses [Make](#). The Makefile has four targets:

- `build` compiles the source code and generates an executable
- `format` applies [ClangFormat](#) to style the source code
- `debug` compiles the source code and generates an executable, including debugging symbols
- `clean` deletes the `build/` directory, including all of the build artifacts

Rubric

Before you start the project, read the [project rubric](https://review.udacity.com/#!/rubrics/2518/view).
<https://review.udacity.com/#!/rubrics/2518/view>

Mentor

We suggest you schedule a check in call with your mentor before you start this project. Your mentor can help you develop a plan to successfully complete the project.

Instructions

1. Clone the project repository: `git clone https://github.com/udacity/CppND-System-Monitor-Project-Updated.git`
2. Build the project: `make build`
3. Run the resulting executable: `./build/monitor`
4. Follow along with the lesson.
5. Implement the `System`, `Process`, and `Processor` classes, as well as functions within the `LinuxParser` namespace.
6. Verify that your submission meets all of the criteria in the [project rubric](#).
7. Submit!