

1. Polymorphism and Inheritance

<https://youtu.be/91JxGNiQdSE>

1. Bjarne on Inheritance

<https://youtu.be/pxDZ7VuyaHI>

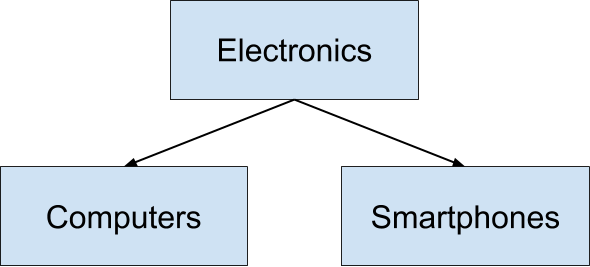
1. Inheritance

<https://youtu.be/qu4dDc-xARM>

**Inheritence**

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.

**[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/15cd39a7-3fda-495d-af24-c5ccd45826a8/concepts/23716416-dddf-491e-bc42-0a7ab887f957)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/15cd39a7-3fda-495d-af24-c5ccd45826a8/concepts/23716416-dddf-491e-bc42-0a7ab887f957)**

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 {

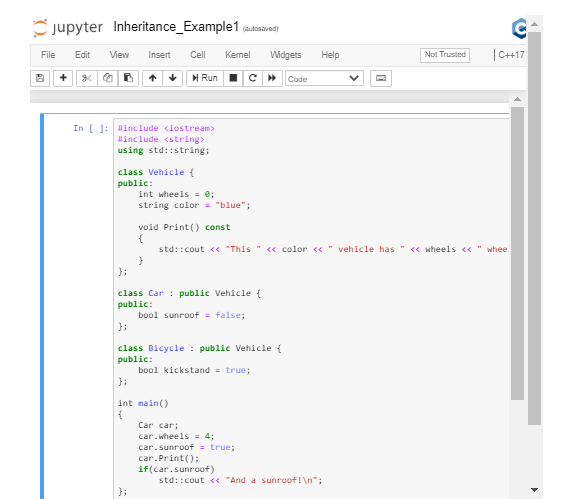
**public**:

**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.



1. 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:**](https://en.cppreference.com/w/cpp/language/derived_class#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:**](https://en.cppreference.com/w/cpp/language/derived_class#Protected_inheritance) the public and protected members of the base class listed after the specifier are protected members of the derived class

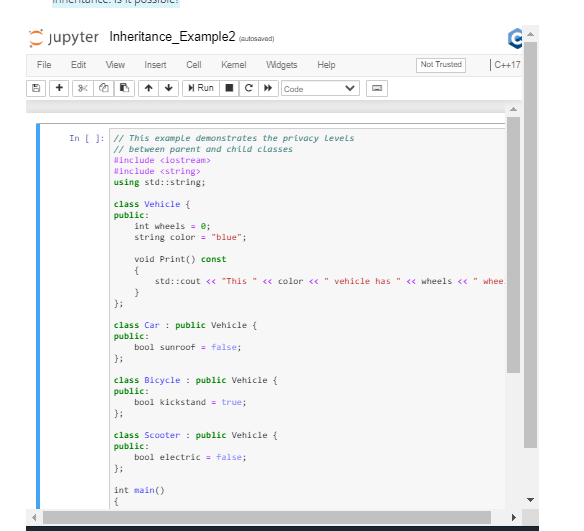
[**Private inheritance:**](https://en.cppreference.com/w/cpp/language/derived_class#Private_inheritance) the public and protected members of the base class listed after the specifier are private members of the derived class

Source: [**C++ reference**](https://en.cppreference.com/w/cpp/language/access)

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?



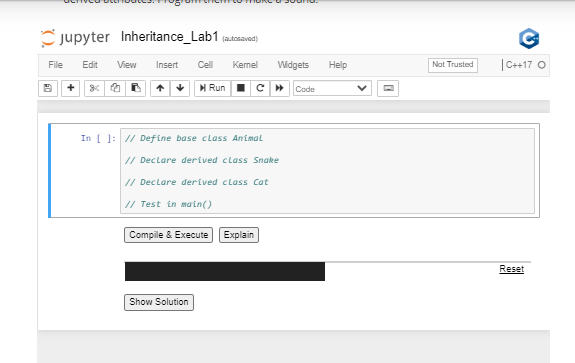
1. Exercise: Animal Class

## 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.



1. Composition

<https://youtu.be/iUkRGy6kK4A>

**Composition**

[**Composition**](https://en.wikipedia.org/wiki/Composition_over_inheritance) 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"**](https://en.wikipedia.org/wiki/Has-a) versus [**"is a"**](https://en.wikipedia.org/wiki/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**](https://www.google.com/search?q=when+to+use+composition+and+when+to+use+inheritance&oq=when+to+use+composition+and+when+to+use+inheritance) 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.

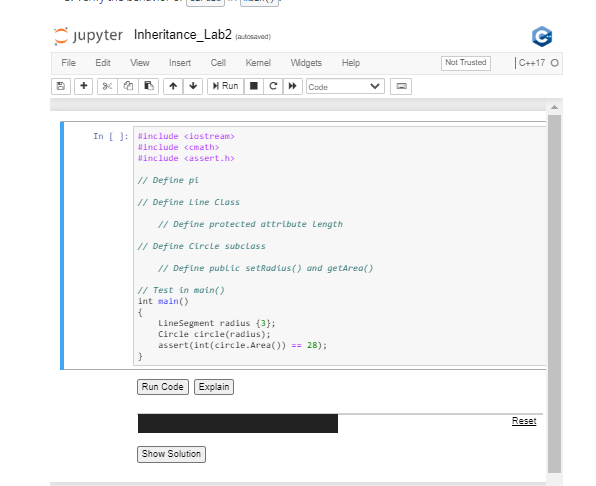
In this example, you'll practice working with composition in C++.

**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.

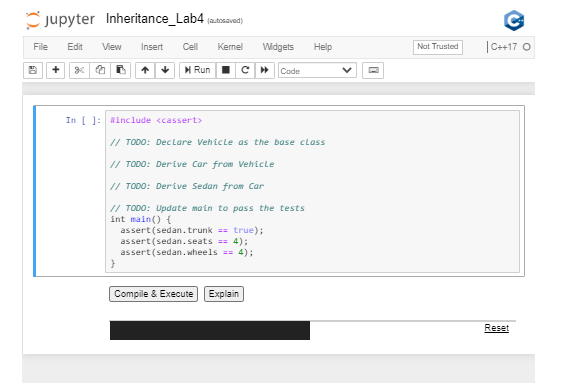
1. Create a class LineSegment.
2. Declare an attribute length in class LineSegment.
3. Define pi (3.14159) with a [**macro**](http://www.cplusplus.com/doc/tutorial/preprocessor/).
4. Create a class Circle, composed of a LineSegment that represent's the circle's radius. Use this radius to calculate the area of the circle (area of a circle = \pi r^2*πr*2).
5. Verify the behavior of Circle in main().



1. 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.



1. Exercise: Friends

<https://youtu.be/GxdPV4mz7wg>

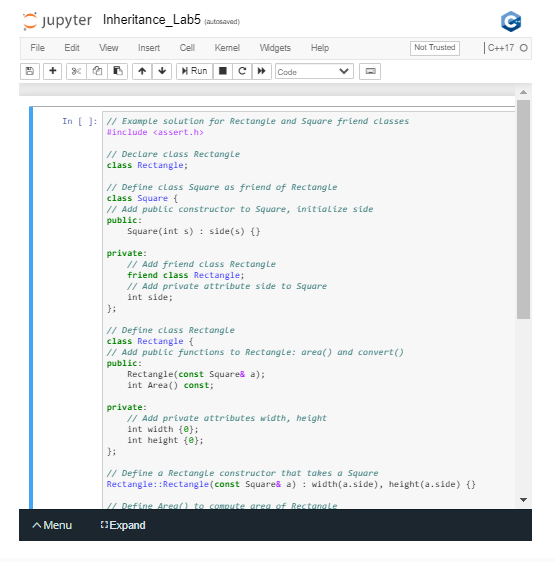
**Friends**

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.



1. Polymorphism: Overloading

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

**Polymorphism**

[**Polymorphism**](https://www.merriam-webster.com/dictionary/polymorphism) is means "assuming many forms".

In the context of object-oriented programming, [**polymorphism**](https://en.wikipedia.org/wiki/Polymorphism_(computer_science)) 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"**](https://en.wikipedia.org/wiki/Function_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 current year*

{

**time\_t** t = time(NULL);

tm\* timePtr = localtime(&t);

year\_ = timePtr->tm\_year;

}

**private**:

**int** day\_;

**int** month\_;

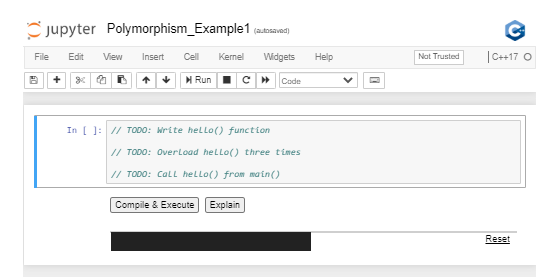
**int** year\_;

};

**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.



1. Polymorphism: Operator Overloading

<https://youtu.be/ejJ8uoPtFoo>

**Operator Overloading**

. In this exercise you'll see how to achieve polymorphism with [**operator overloading**](https://en.cppreference.com/w/cpp/language/operators). 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.



1. 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**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary) 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**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary) is a [**virtual function**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary) that the base class [**declares**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary) but does not [**define**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary).

A pure virtual function has the side effect of making its class [**abstract**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-glossary). 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;

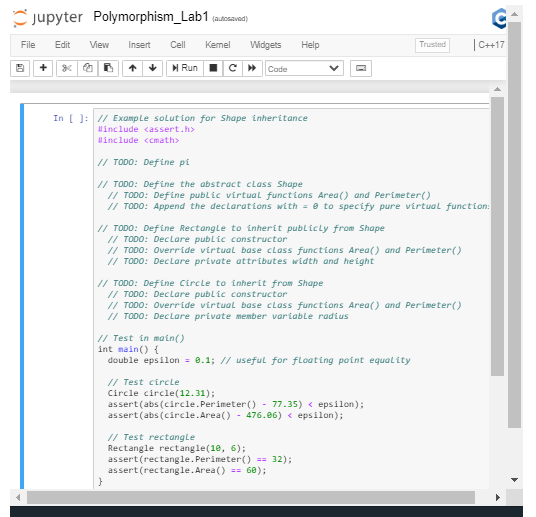
**virtual** **double** **Perimeter**() **const** = 0;

};

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()



1. Polymorphism: Overriding

<https://youtu.be/u15HcpiBeRc>

**Polymorphism: Overriding**

[**"Overriding"**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary) a function occurs when:

1. A base class declares a [**virtual function**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary%20function).
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"); }

};

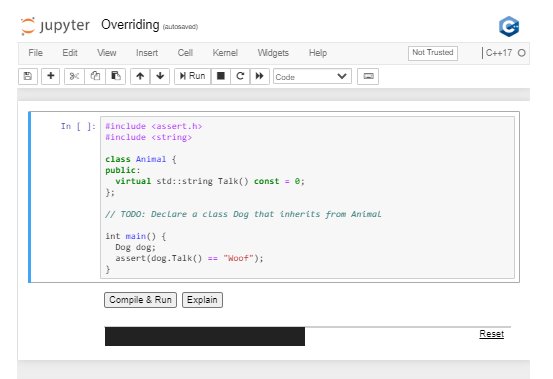
In this example, Animal exposes a virtual function: Talk(), but does not define it. Because Animal::Talk() is undefined, it is called a [***pure virtual function***](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary), as opposed to an ordinary (impure? 😉) [***virtual function***](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary).

Furthermore, because Animal contains a pure virtual function, the user cannot instantiate an object of type Animal. This makes Animal an [***abstract class***](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#glossary).

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.



### Function Hiding

Function hiding is [**closely related, but distinct from**](https://stackoverflow.com/questions/19736281/what-are-the-differences-between-overriding-virtual-functions-and-hiding-non-vir), 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**](https://isocpp.org/wiki/faq/strange-inheritance), particularly when the two functions have slightly different function signatures.

1. 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**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-override), 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**](http://www.cs.cmu.edu/~motionplanning/reading/PlanningforDynamicVeh-1.pdf), 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 \* cos(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)

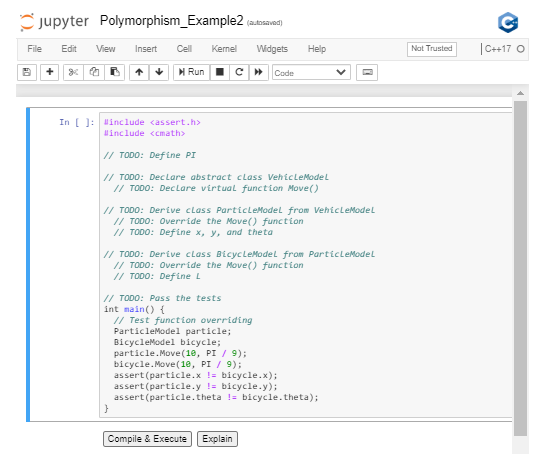
x += v \* cos(theta)

y += v \* cos(theta)

You are encouraged to [**read more**](http://www.cs.cmu.edu/~motionplanning/reading/PlanningforDynamicVeh-1.pdf) 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**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c128-virtual-functions-should-specify-exactly-one-of-virtual-override-or-final).
5. Pass the tests in main() by verifying that the two Move() functions override each other in different scenarios.



1. Multiple Inheritance

<https://youtu.be/jEoPLBdLLsw>

**Multiple Inheritance**

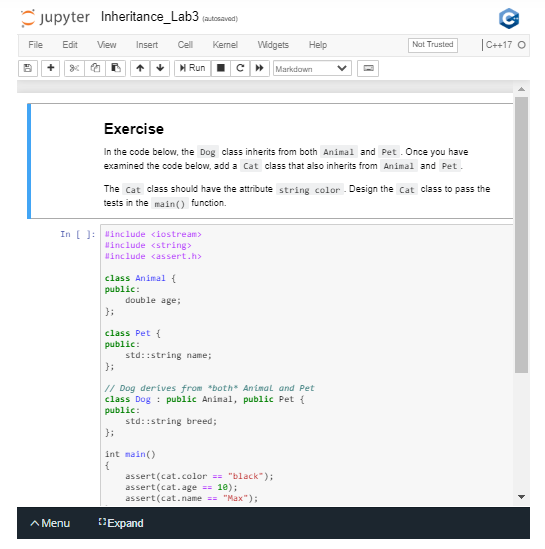
In this exercise, you'll get some practical experience with multiple inheritance. If you have class Animal and another class Pet, then you can construct a class Dog, which inherits from both of these base classes. In doing this, you are able to incorporate attributes of multiple base classes.

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

* [**"Use multiple inheritance to represent multiple distinct interfaces"**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c135-use-multiple-inheritance-to-represent-multiple-distinct-interfaces)
* [**"Use multiple inheritance to represent the union of implementation attributes"**](http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c136-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().



<https://youtu.be/p29phGPfKnQ>

1. Generic Programming

<https://youtu.be/k2Hai5sBemU>

1. Bjarne on Generic Programming

<https://youtu.be/m3a4ojP0dVQ>

1. 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) {

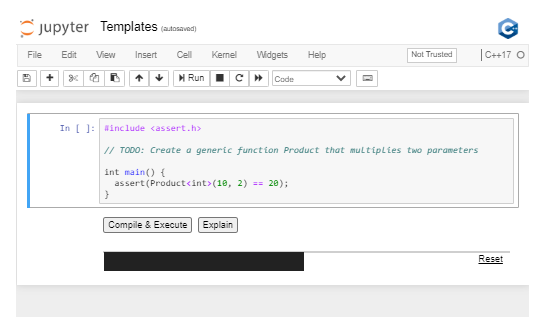
**return** a+b;

}

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 templatized class, use a templatized 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 vectors are indeed a generic class!



1. Bjarne on Templates

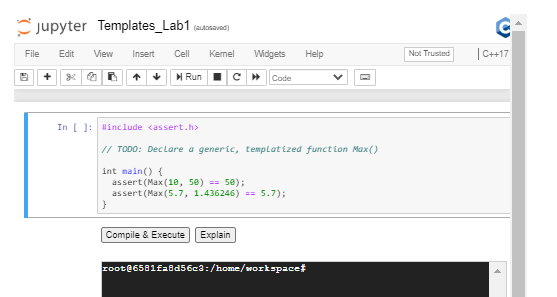
<https://youtu.be/tnOsS8JEO0U>

1. 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.



1. Deduction

Deduction

In this example, you will see the difference between total and partial [**deduction**](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-deduce).

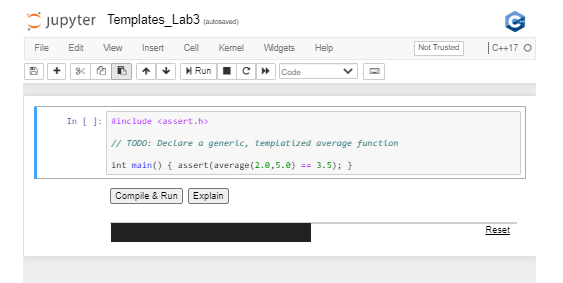
Deduction occurs when you instantiate an object without explicitly identifying the types. Instead, the compiler "deduces" the types. This can be helpful for writing code that is generic and can handle a variety of inputs.

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/JJLGNlQ1QLk>



1. 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.

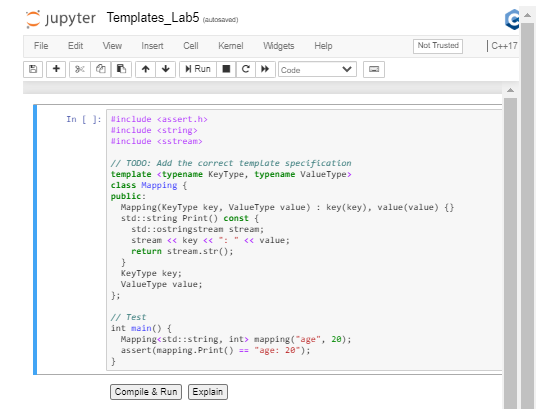
In this exercise you will create a class Mapping that maps a generic key to a generic value.

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.

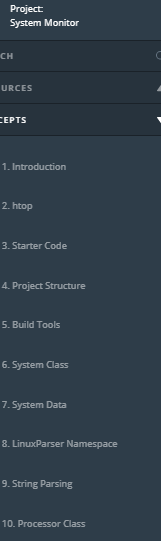
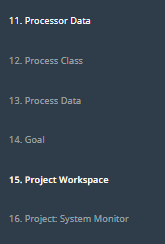


1. Summary

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

1. Bjarne on Best Practices with Classes

<https://youtu.be/gWcAMxhNOcg>

* 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>

* 1. Htop

<https://youtu.be/Cz4rDC-WecA>

* 1. Starter Code

<https://youtu.be/eguBVmzhTS4>

* 1. Project Structure

<https://youtu.be/dOnUD8UUhMg>

<https://youtu.be/10HWAXzY_90>

* 1. Build Tools

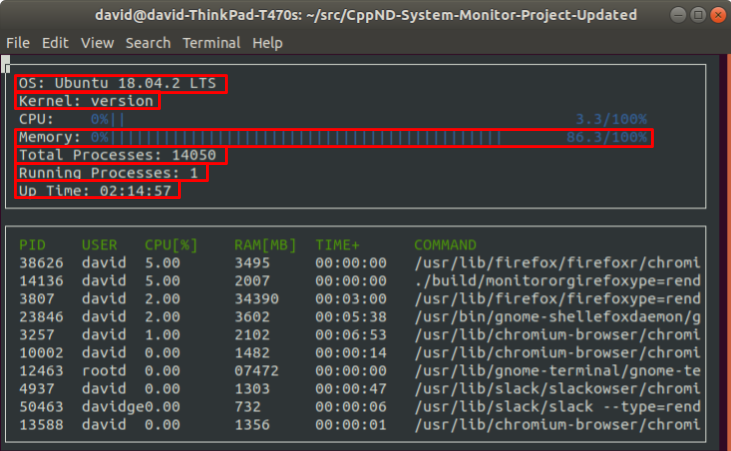
<https://youtu.be/PSPI33rKQas>

* 1. System Class

<https://youtu.be/M6tpsAZWnjI>

* 1. System Data

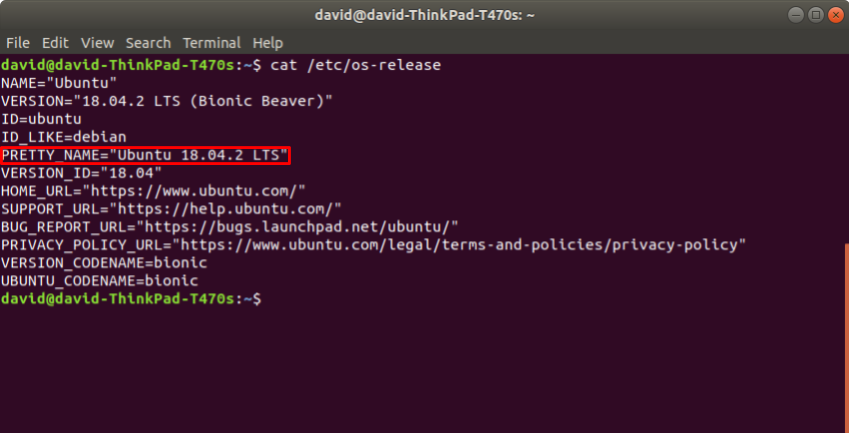
System Data

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)

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.

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)

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

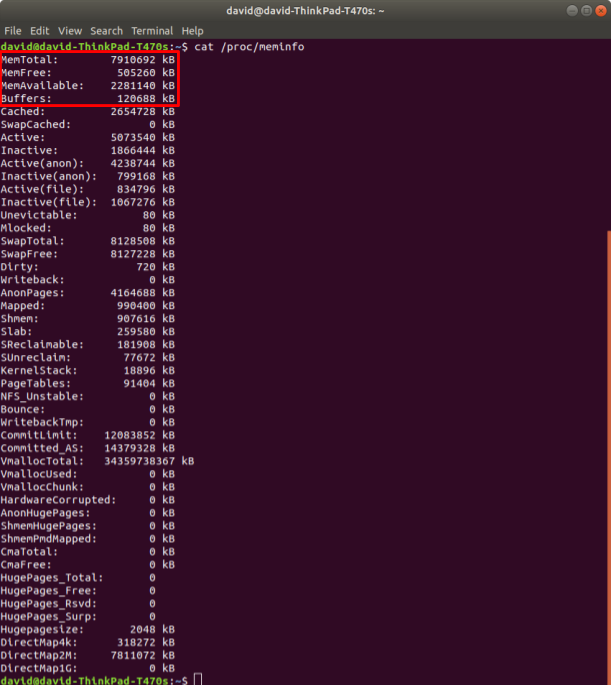
## Kernel

Information about the kernel exists /proc/version file.

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)

Memory Utilization

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

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)

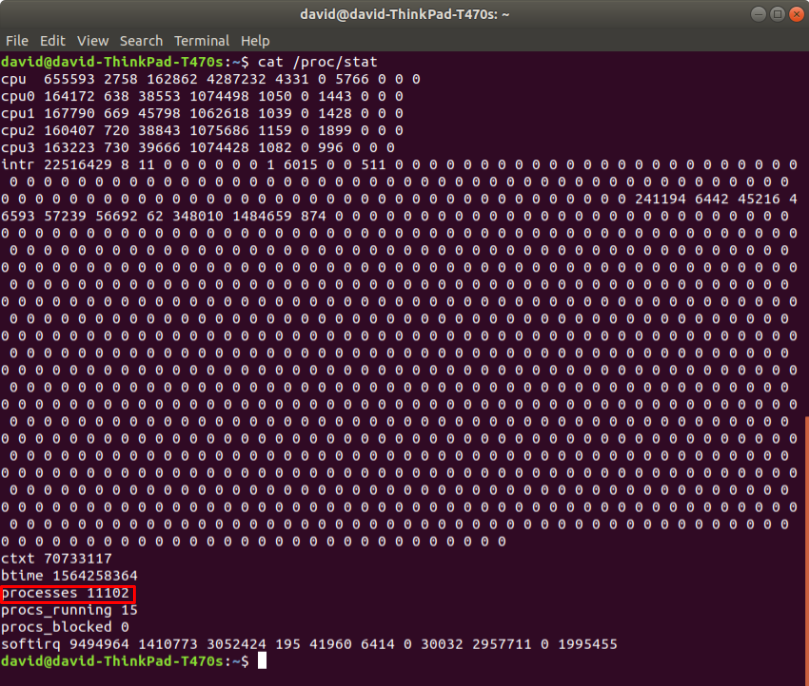
There are a [variety](https://www.thegeekdiary.com/understanding-proc-meminfo-file-analyzing-memory-utilization-in-linux/) of [ways](https://access.redhat.com/solutions/406773) to use this data to calculate memory utilization.

[Hisham H. Muhammad](http://hisham.hm/about), the author of [htop](http://hisham.hm/htop/index.php" \t "_blank), wrote a [Stack Overflow answer](https://stackoverflow.com/a/41251290) about how htop calculates memory utilization from the data in /proc/meminfo.

Use the formula that makes the most sense to you!

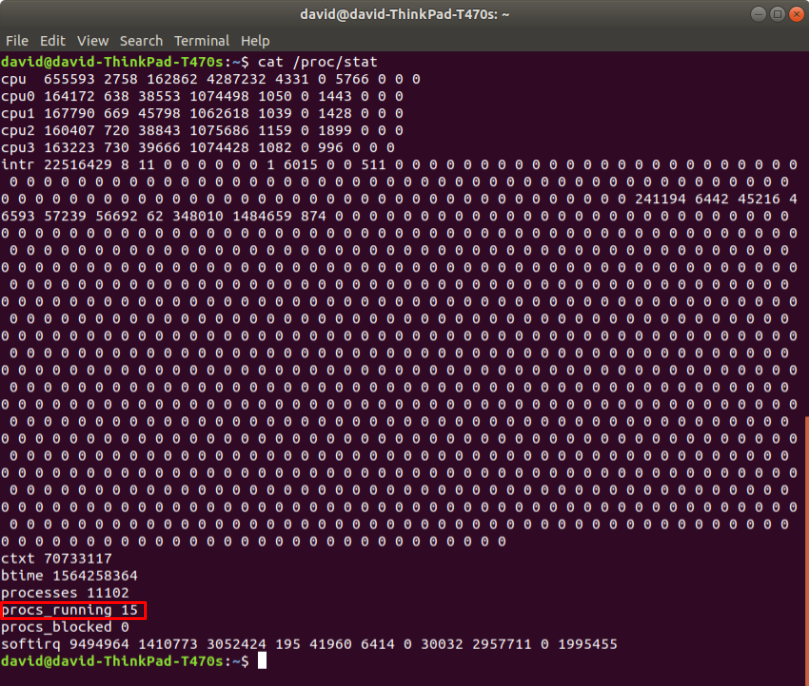
## Total Processes

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

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)

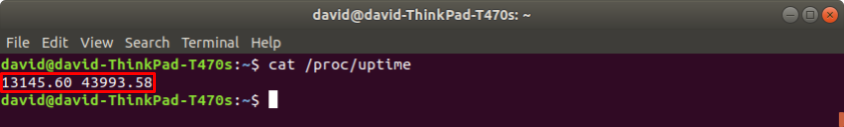
## Running Processes

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

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)

Up Time

Information about system up time exists in the /proc/uptime file.

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/8b7ff1b6-3cb8-4573-9743-173895cd8224)

*This file contains two numbers (values in seconds): the uptime of the system (including time spent in suspend) and the amount of time spent in the idle process.*

From the [man page for proc](http://man7.org/linux/man-pages/man5/proc.5.html)

* 1. LinuxParser Namespace

<https://youtu.be/f9Qt2AlPQeE>

**Relationship To System Class**

<https://youtu.be/XQrAStOlZCQ>

* 1. String Parsing

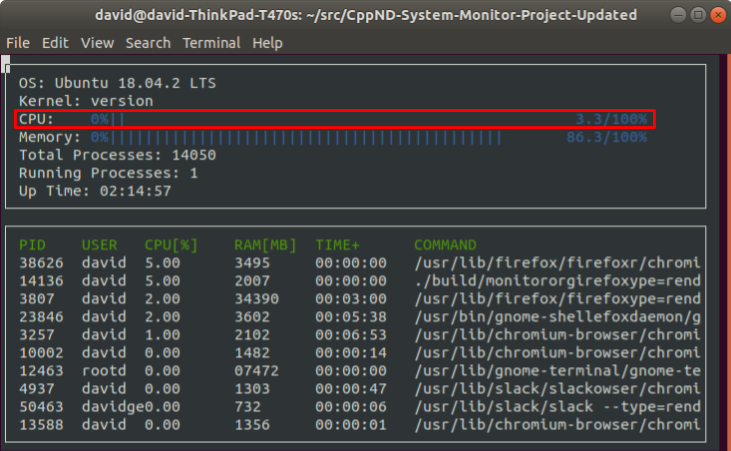
<https://youtu.be/vFhTd8HyiJw>

* 1. Processor Class

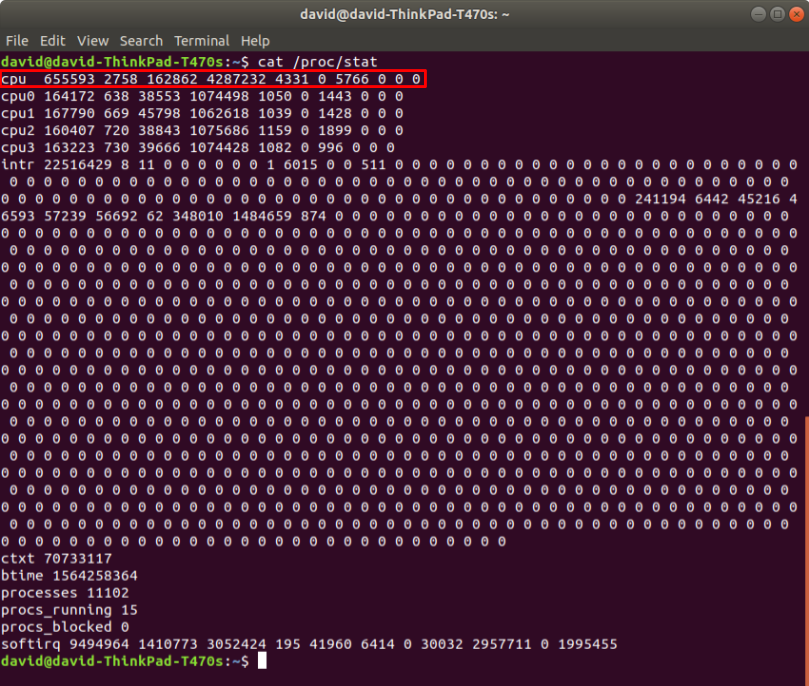
<https://youtu.be/eMkJE3y9bwo>

* 1. Processor Data

Processor Data

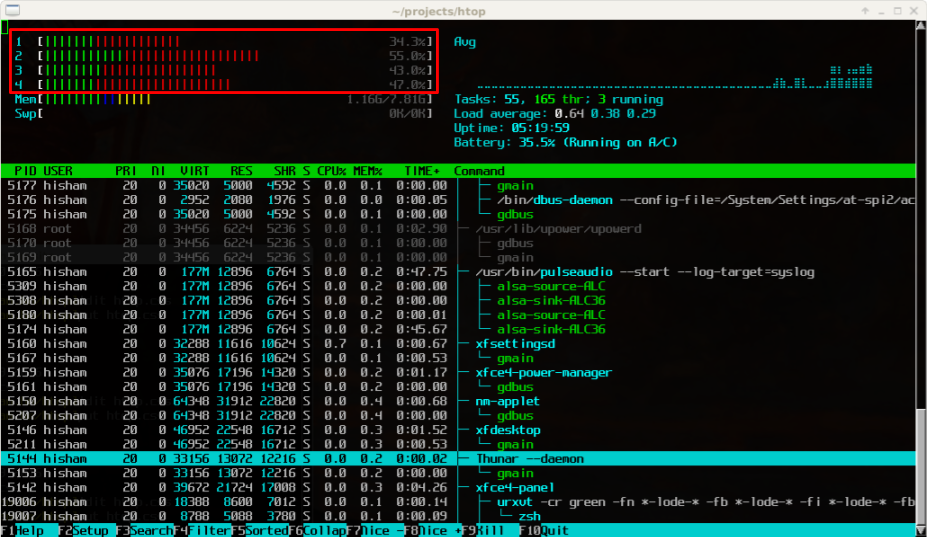
[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/beb4374f-bbf3-4706-a7dc-702199a8426a)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/beb4374f-bbf3-4706-a7dc-702199a8426a)

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

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/beb4374f-bbf3-4706-a7dc-702199a8426a)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/beb4374f-bbf3-4706-a7dc-702199a8426a)

This data is more complex than most of the other data necessary to complete this project.

For example, /proc/stat contains aggregate processor information (on the "cpu" line) and individual processor information (on the "cpu0", "cpu1", etc. lines). Indeed, [htop](https://hisham.hm/htop/" \t "_blank) displays utilization information for each individual processor.

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/beb4374f-bbf3-4706-a7dc-702199a8426a)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/beb4374f-bbf3-4706-a7dc-702199a8426a)

For this project, however, you only need to display aggregate CPU information, which you can find on the "cpu" line of /proc/stat.

If you would like to add individual processor information to your system monitor project, go for it!

Data

/proc/stat contains 10 integer values for each processor. The Linux source code [documents each of these numbers](https://github.com/torvalds/linux/blob/master/Documentation/filesystems/proc.txt):

*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](https://github.com/Leo-G/DevopsWiki/wiki/How-Linux-CPU-Usage-Time-and-Percentage-is-calculated) and [this StackOverflow post](https://stackoverflow.com/questions/23367857/accurate-calculation-of-cpu-usage-given-in-percentage-in-linux) 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:

Δ active time units / Δ total time units

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

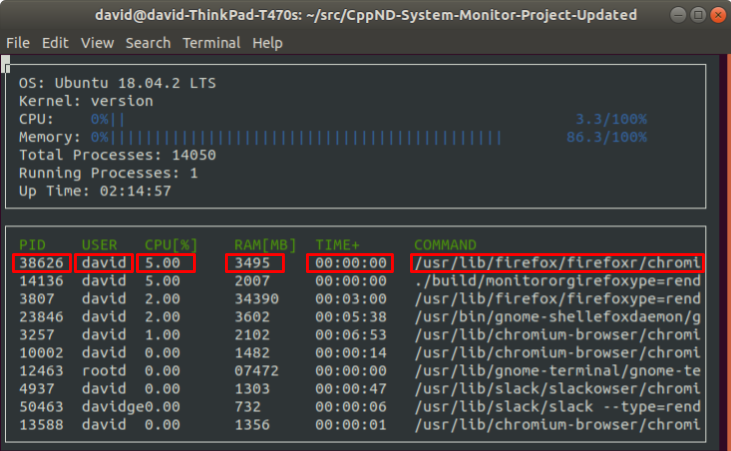
NEXT

* 1. Process Class

<https://youtu.be/sEkf6TqLKBk>

* 1. Process Data

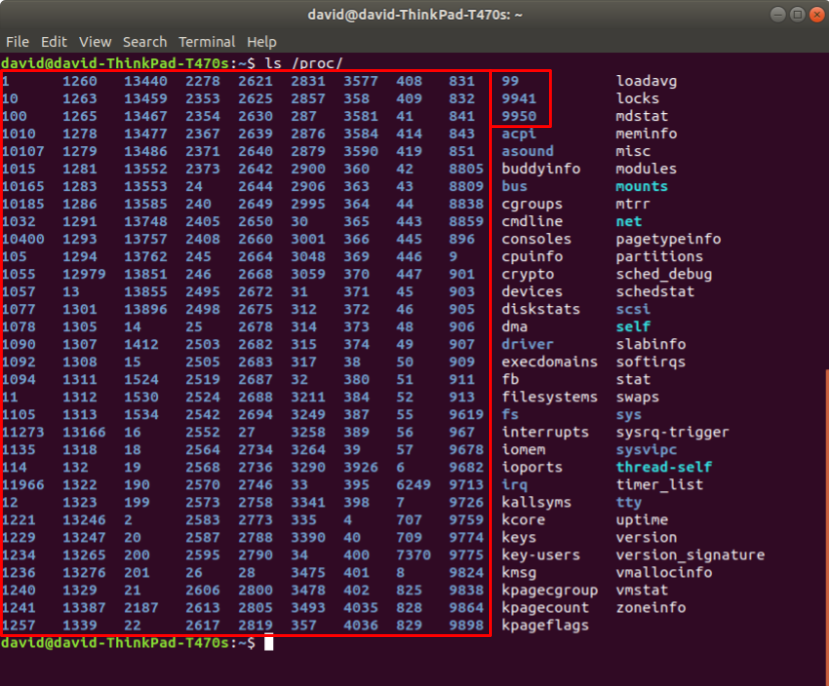
Process Data

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)

Linux stores data about individual processes in files within subdirectories of the /proc directory. Each subdirectory is named for that particular process's [identifier](https://en.wikipedia.org/wiki/Process_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.

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)

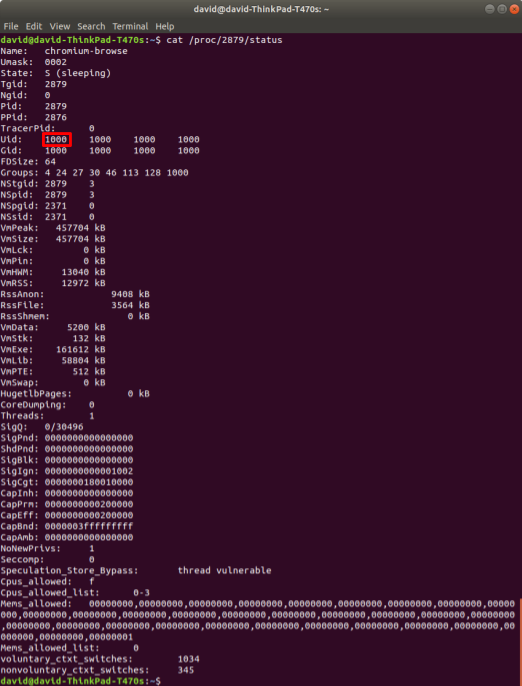
Parsing directory names with C++ is tricky, so we have provided in the project starter code a pre-implemented function to capture the PIDs.

## User

Each process has an associated [user identifier (UID)](https://en.wikipedia.org/wiki/User_identifier), 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.

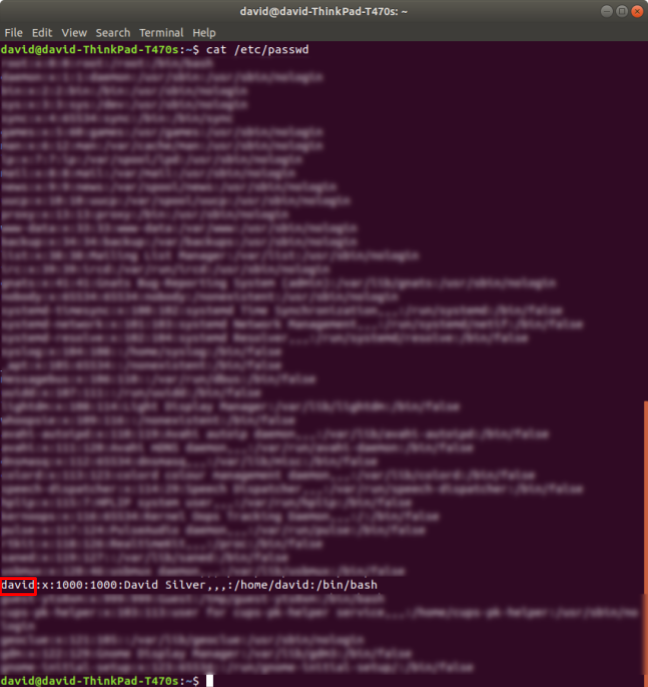
[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)

The [man page for proc](http://man7.org/linux/man-pages/man5/proc.5.html) 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.

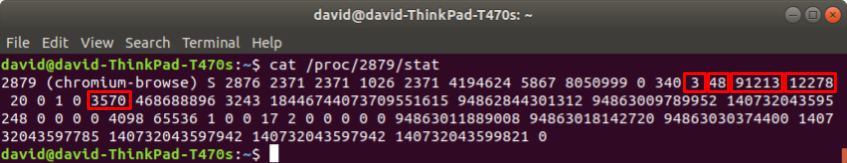
### Username

[/etc/passwd](http://man7.org/linux/man-pages/man5/passwd.5.html) contains the information necessary to match the UID to a username.

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)

Processor Utilization

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

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)

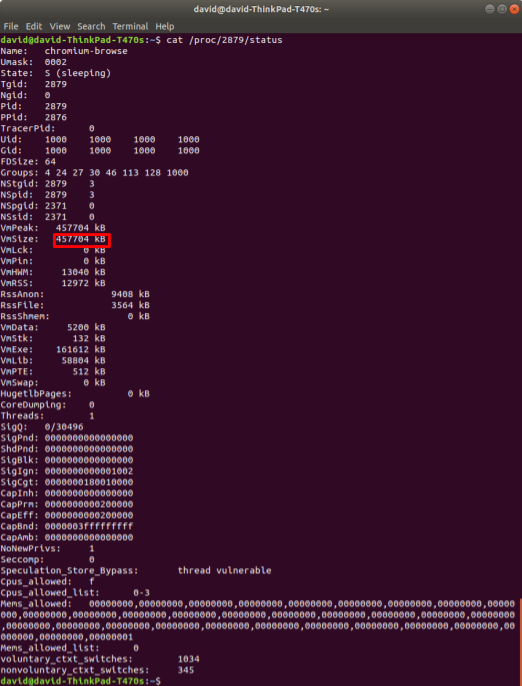
Much like the calculation of aggregate processor utilization, half the battle is extracting the relevant data from the file, and the other half of the battle is figuring out how to use those numbers to calculate processor utilization.

The "/proc/[pid]/stat" section of the [proc man page](http://man7.org/linux/man-pages/man5/proc.5.html) describes the meaning of the values in this file. [This StackOverflow answer](https://stackoverflow.com/a/16736599) 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

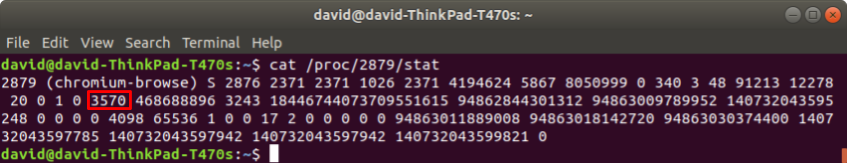
Linux stores memory utilization for the process in /proc/[pid]/status.

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)

In order to facilitate display, consider [converting the memory utilization into megabytes](https://www.google.com/search?q=convert+from+kb+to+mb&oq=convert+from+kb+to+mb).

## Up Time

Linux stores the process up time in /proc/[pid]/stat.

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)

The "/proc/[pid]/stat" section of the [proc man page](http://man7.org/linux/man-pages/man5/proc.5.html) 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)).*

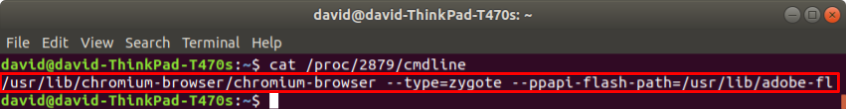
Note that the "starttime" value in this file is measured in "clock ticks". In order to convert from "clock ticks" to seconds, you must:

* #include [<unistd.h>](http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/unistd.h.html)
* 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.

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)](https://classroom.udacity.com/nanodegrees/nd213/parts/f9fffe8e-1984-4045-92b6-64854de4df2b/modules/70db33ed-8e7e-45e9-ab1d-3d0b257fc196/lessons/4636c0ce-e36e-479b-ba36-58b4b849a9e8/concepts/629dbd7b-18f2-4f43-9cb7-d74714b761af)

* 1. Goal

<https://youtu.be/xw6_Mz3O54Y>

* 1. Project workspace

Udacity Workspace

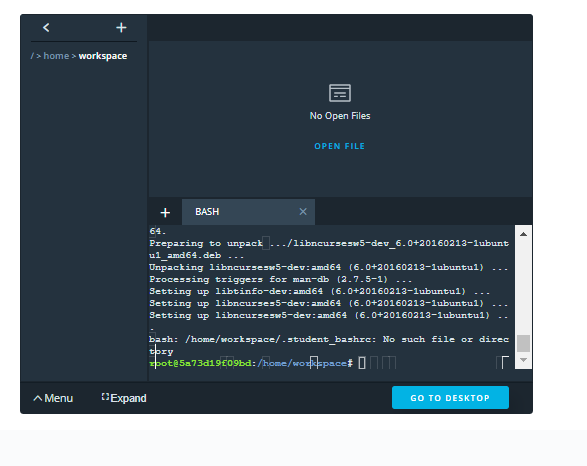
You are welcome to use this Udacity Workspace to complete the project. Or you can use your own Linux development environment, if you have one.

## GitHub Repo

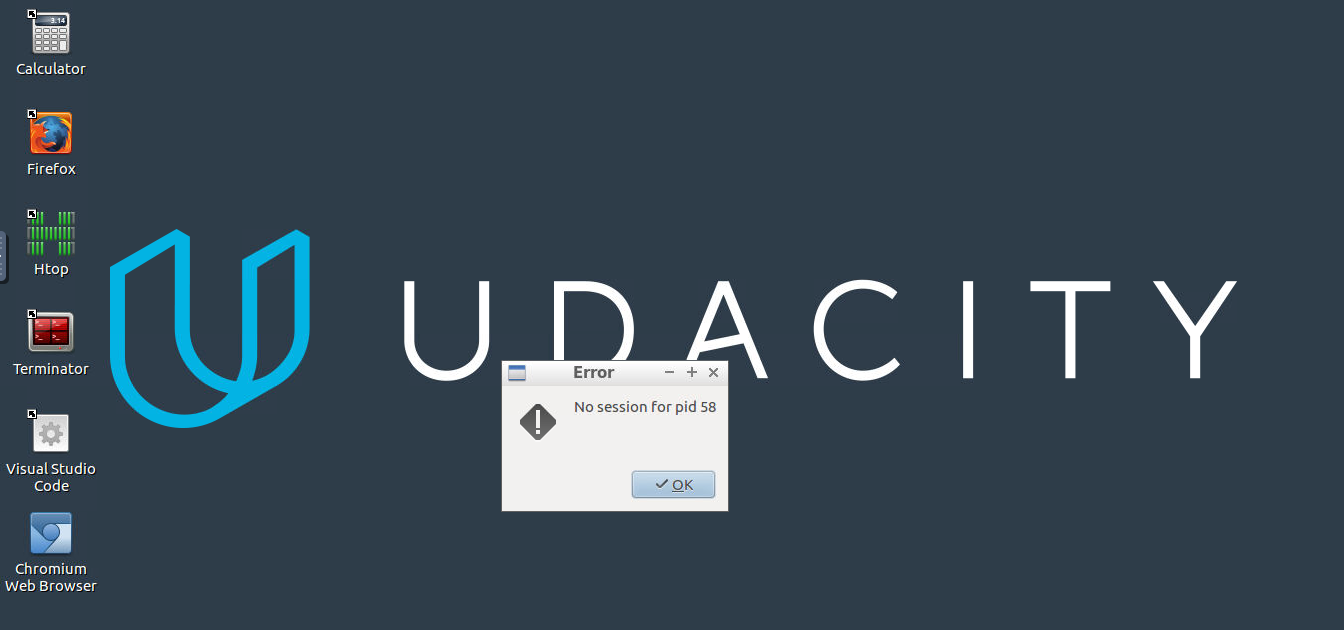
The starter code is available on GitHub: [**https://github.com/udacity/CppND-System-Monitor**](https://github.com/udacity/CppND-System-Monitor)

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



* 1. Project: system monitor

## Project Submission MENTOR

###### 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**](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**](https://engineering.udacity.com/creating-a-gpu-enhanced-virtual-desktop-for-udacity-497bdd91a505) 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**](https://www.gnu.org/software/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**](https://www.gnu.org/software/make/). The Makefile has four targets:

* build compiles the source code and generates an executable
* format applies **[ClangFormat](https://clang.llvm.org/docs/ClangFormat.html" \t "_blank)** 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**](https://review.udacity.com/#!/rubrics/2518/view).
7. Submit!