Rust Pre-session

RUST PROGRAMMING LANGUAGE

TOPIC

1. While-Loop
2. For-Loop
3. Break and Continue
4. Nested Loop
5. Function

While-Loop

While-Loop

```
fn main() {
    let condition = true;

    while condition {
        println!("Hello World");
    }
}
```

The ***While-Loop*** is another control flow construct (similar to if-else); that is it changes the flow of execution. It's used to execute statements in repetition while the condition evaluates to `true`

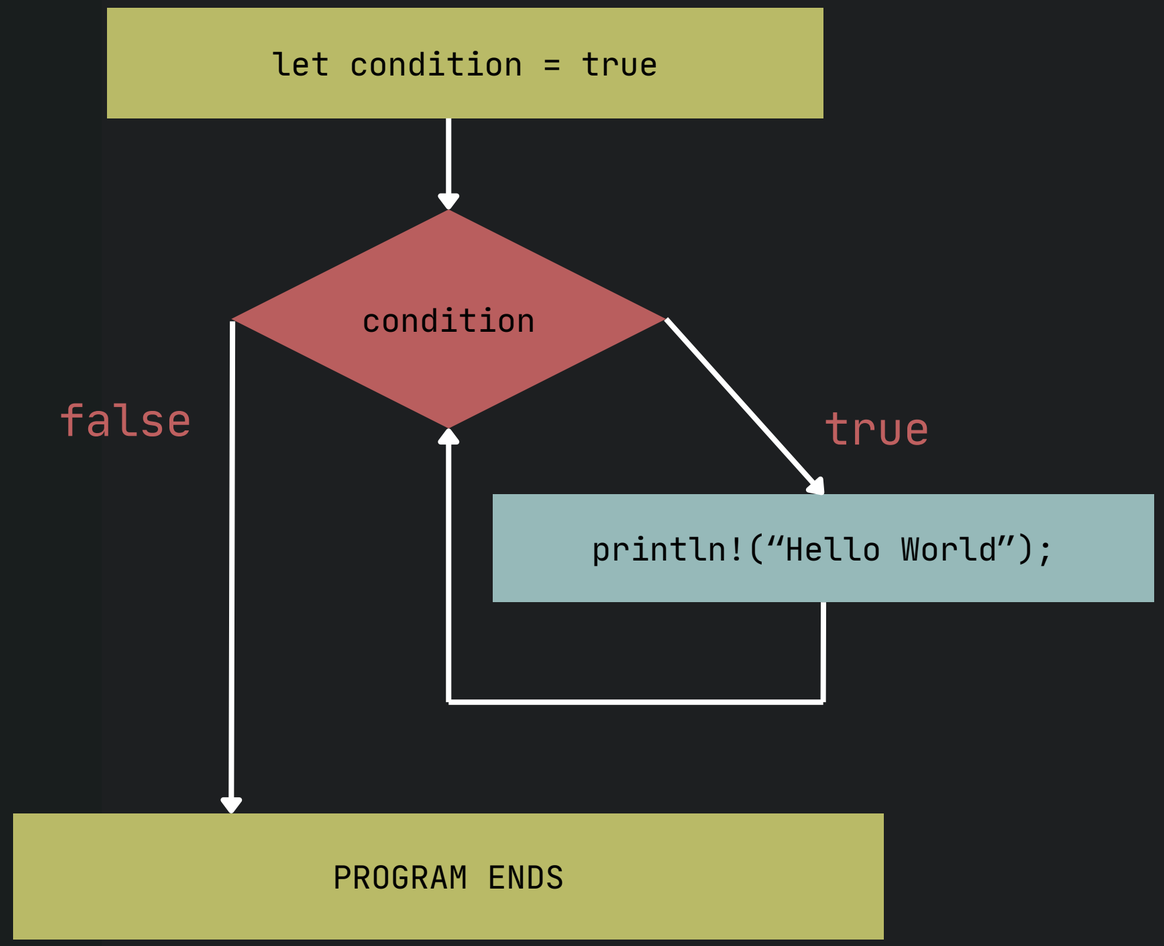
An **expression** that yields a **boolean** is expected here. Similar to what we've worked with if-else earlier.

Describing While-Loop with Flow Chart

```
fn main() {
    let condition = true;

    while condition {
        println!("Hello World");
    }
}
```

Can you guess what this program does by looking at this flowchart?



Describing While-Loop with Flow Chart

```
fn main() {
    let condition = true;

    while condition {
        println!("Hello World");
    }
}
```

You might’ve guessed it. This program prints the word ***Hello World*** infinitely 🔥

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Example Use Case Of While Loop

```
fn main() {
    println!("Number 1");
    println!("Number 2");
    println!("Number 3");
    println!("Number 4");
    println!("Number 5");
    println!("Number 6");
    println!("Number 7");
    println!("Number 8");
    println!("Number 9");
    println!("Number 10");
}
```

Imagine that you must write a program that prints out the word ***“Number 1”*** to ***“Number 10”*** .

We might’ve written this program if we hadn’t learned about ***While-Loop***.

We can do better! We’ll rewrite this program using ***While-Loop.***

Example Use Case Of While Loop

```
fn main() {
    let number = 1;

    while true {
        println!("Number {number}");
    }
}
```

We'll start with this simple code. If you run this program now, it will print out the word ***"Number 1"*** infinitely.

```
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
Number 1
```

It doesn't quite what we want yet but we'll get to it!

Right now, we're only able to print out the word ***"Number 1"*** multiple times. However, the number should be increasing every time it's printed.

Example Use Case Of While Loop

```
fn main() {
    let mut number = 1;

    while true {
        println!("Number {number}");
        number = number + 1;
    }
}
```

So, there should be an update to the variable `***number***` so that it increases every time it's printed.

The variable `***number***` is loaded, here and then add it by 1

Then, the expression `***number + 1***` got assigned to the variable `***number***` itself.

Consequently, the variable `***number***` is increased by one.

```
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
Number 9
Number 10
Number 11
Number 12
Number 13
Number 14
```

Example Use Case Of While Loop

```
fn main() {
    let mut number = 1;

    while true {
        println!("Number {number}");
        number = number + 1;
    }
}
```

Now we're getting closer!
The number increased!

However, it is way too many numbers printed out right now.

```
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
Number 9
Number 10
Number 11
Number 12
Number 13
Number 14
```

Example Use Case Of While Loop

```
fn main() {
    let mut number = 1;

    while true {
        println!("Number {number}");
        number = number + 1;
    }
}
```

It's because the condition was `***true***` all along; so, the while-loop will never stop -- it only stops when the condition is `***false***`

Therefore, we should **limit** the while-loop to only start or continue when the variable `***number***` is less than ***11***; since we don't want to see any other number greater than 10 printed like what we're seeing right now.

```
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
Number 9
Number 10
Number 11
Number 12
Number 13
Number 14
```

Example Use Case Of While Loop

```rust
fn main() {
    let mut number = 1;

    while number < 11 {
        println!("Number {number}");
        number = number + 1;
    }
}
```

*"Therefore, we should **limit** the while-loop to only start or continue when the variable `**number**` is less than **11**; since we don't want to see any other number greater than 10 printed like what we're seeing right now."*

```
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
Number 9
Number 10
```

Extra: Compound Assignment

```rust
fn main() {
    let mut number = 1;

    while number < 11 {
        println!("Number {number}");
        number = number + 1;
    }
}
```

```rust
fn main() {
    let mut number = 1;

    while number < 11 {
        println!("Number {number}");
        number += 1;
    }
}
```

On the right-hand side, it's called ***Compound Assignment***.
It's a shortened way to write the `***number = number + 1***` by writing it as `***number += 1***`. This works for multiple operators as well such as multiplication `*`, you can write it as `***=***`.

Exercise

WHILE-LOOP

Rust programming language does not have an exponent operator built-int. So you can’t write `**2^4**` in Rust.

Therefore, we’ll write a program that will do an exponent for us by using While-Loop.

```
fn main() {
    let base: f64 = 2.0;
    let exponent: u32 = 3;

    /*expected output: 2^3 = 8*/
}
```

Hint

$$b^n = 1 \times \underbrace{b \times b \times \cdots \times b \times b}_{n \text{ times}}.$$

Exponentiation here is multiplying the ***base repeatedly for n times*** and multiplying it to number one.

We multiply it by one since any number powers with zero equals one. (assuming zero powers zero equals one)

More Hint

```
fn main() {
    let base: f64 = 2.0;
    let exponent: u32 = 3;

    // the result of the exponentiation is stored here
    let mut result = 1.0;

    // used for keeping track of how many time
    // the number is multiplied
    let mut i = 0;

    while i < exponent {
        /*something*/
    }

    println!("{base}^{exponent} = {result}");
}
```

```rust
fn main() {
    let base: f64 = 2.0;
    let exponent: u32 = 3;

    // the result of the exponentiation is stored here
    let mut result = 1.0;

    // used for keeping track of how many time
    // the number is multiplied
    let mut i = 0;

    while i < exponent {
        result *= base; // shortened from result = result * base
        i += 1;         // shortened from i = i + 1
    }

    println!("{base}^{exponent} = {result}");
}
```

For-Loop

For-Loop

```
fn main() {
    for number in 1..11 {
        println!("Number {number}");
    }
}
```

For-loop is another way to express repetition in the code -- similar to the ***While-loop***.

However, this construct provides a more concise way to express.

For-Loop

```
fn main() {
    for number in 1..11 {
        println!("Number {number}");
    }
}
```

Unlike the ***While-loop*** which requires an **expression** that yields a boolean value here, the ***For-loop*** requires an **expression** that is an ***Iterator***

The ***For-loop*** will ***iterate*** each element in that given ***Iterator*** expression.

Explanation: Iterator

```
1..11
```

You can think of an ***Iterator*** as a bag of items.

Then the ***For-loop*** will take each item in the bag and give it to you.

In Rust, there are many kinds of ***Iterator.*** In this topic, you will encounter the ***range*** iterator.

Explanation: Iterator

```
1..11
```

So, this **range** iterator is a bag of integers.

Specifically, this range contains the number
from 1 to 11 (exclusive).

```
1, 2, 3, 4, 5, 6, 7 8, 9, 10
```

Bonus: Range Inclusicve

```
1..11
```

This is called `***Range***`

```
1, 2, 3, 4, 5, 6, 7 8, 9, 10
```

```
1..=11
```

This is called `***RangeInclusive***`

Unlike `***Range***`, this includes the ending number.

```
1, 2, 3, 4, 5, 6, 7 8, 9, 10, 11
```

For-Loop

```
fn main() {
1
    for number in 1..11 {
    2
        println!("Number {number}");
    }
}
```

The variable `*number*` is tied to the scope ***2*** (not ***1***) and assigned with the number of 1, 2, 3, ..., 10 every time the loop goes.

For-Loop vs While-Loop

```rust
fn main() {
    for number in 1..11 {
        println!("Number {number}");
    }
}
```

vs

```rust
fn main() {
    let mut number = 1;

    while number < 11 {
        println!("Number {number}");
        number += 1;
    }
}
```

These two codes behave the same way.

However, the ***For-loop*** version is more concise.

Normally, you should use ***For-loop*** most of the time since it’s more concise and can prevent some mistakes relating to while’s conditions, updating variables, etc.

Exercise

FOR-LOOP

Try to rewrite the previous exercise (exponentiation)
by using ***For-loop*** instead of ***While-loop***

Break and Continue

Break

```
fn main() {
    for number in 1..11 {
        if number == 6 {
            println!("HIT THE BREAK!");
            break;
        }

        println!("Number {number}");
    }
}
```

The `***break***` is used to immediately ***stop*** the loop (this includes the While and For loops).

If there weren't a ***break*** here, you would see the number being printed from 1 to 10.

```
Number 1
Number 2
Number 3
Number 4
Number 5
HIT THE BREAK!
```

Continue

```rust
fn main() {
    for number in 1..11 {
        if number % 2 == 0 {
            println!("skipping even number");
            continue;
        }

        println!("Number {number}");
    }
}
```

The `***continue***` is used to skip the current iteration of the loop.

When the `***continue***` is executed, the rest of the code in the loop in the current iteration is skipped and the new iteration is started again

```
Number 1
skipping even number
Number 3
skipping even number
number 5
skipping even number
number 7
skipping even number
number 9
skipping even number
```

Nested Loop

Nested Loop

```
fn main() {
    let width = 4;
    let height = 3;

    for y in 0..height {
        for x in 0..width {
            print!("*");
        }
        println!("");
    }
}
```

When putting the loop inside another loop, it's called ***Nested Loop.***

This program prints out a pattern of square asterisks.

```
****
****
****
```

Break Down

```
fn main() {
1
    let width = 4;
    let height = 3;

    for y in 0..height {
    2
        for x in 0..width {
        3
            print!("*");
        }
        println!("");
    }
}
```

This might look complicated at first but let's break it down.

Let's focus on loop with the scope number ***3***.

The loop itself is simple. It'll print out asterisks in the number of the variable `width`

If the `***width***` were 5, it would print `*****`, or if it were 7, it would print `*******`.

Break Down

```
fn main() {
1
    let width = 4;
    let height = 3;

    for y in 0..height {
    2
        for x in 0..width {
        3
            print!("*");
        }
        println!("");
    }
}
```

Now let's focus on the loop with scope ***2***.

The loop itself again is very simple. It only contains the inner loop (***3***) and a ***`println!("")`*** statement, which doesn't print anything but enters a new line.

Since the inner loop (***3***)'s functionality is to print out a line of asterisks. Therefore, the outer loop (***2***) will print multiple lines of asterisks, resulting in an asterisk square.

Break Down

```
fn main() {
1    let width = 4;
     let height = 3;

     for y in 0..height {
     2    print_line_of_asterisks();

          println!("");
     }
}
```

Or you could imagine that the inner loop (***3***) was something that can print a line of asterisks.

Now this whole nested loop thing can be a little bit more understandable :)

Exercise

NESTED-LOOP

Try to write a program that prints out the following pattern using nested loop.

```
n: 5
1
12
123
1234
12345
```

```
n: 4
1
12
123
1234
```

```
n: 3
1
12
123
```

```rust
fn main() {
    let n = 4 /*can be changed*/;
    println!("n: {n}");
}
```

Exercise

NESTED-LOOP

Try to write a program that prints out the following pattern using nested loop.

```
n: 5
1
12
123
1234
12345
```

```
n: 4
1
12
123
1234
```

```
n: 3
1
12
123
```

```rust
fn main() {
    let n = 4 /*can be changed*/;
    println!("n: {n}");
}
```

Hint

```
fn main() {
    let n = 4;
    println!("n: {n}");

    for i in 1..=n {
        // i need to put inner loop here

        println!("");
    }
}
```

This is `***RangeInclusive***`; therefore, this iterator will give out numbers: ***1, 2, 3, 4***

We should put some code that prints out the number in pattern i.e. `***123***`

Solution

```
fn main() {
    let n = 4;
    println!("n: {n}");

    for i in 1..=n {
        for j in 1..=i {
            print!("{j}");
        }

        println!("");
    }
}
```

Function

$$f(x) = x^2$$

$$g(x, y) = x^2 + y^2$$

$$f(3) = 3^2 = 9$$

$$g(2, 3) = 2^2 + 3^2 = 13$$

$$f(x) = x^2$$

$$g(x, y) = x^2 + y^2$$

The ***function*** takes some ***inputs*** and then ***returns*** (gives) an output

Function

```
fn f(x: f64) -> f64 {
    return x * x;
}

fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x_squared + y_squared;
}
```

$$f(x) = x^2$$

$$g(x, y) = x^2 + y^2$$

Parameter

```
fn f(x: f64) -> f64 {
    return x * x;
}

fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x_squared + y_squared;
}
```

These are called `***Parameter***`**.** The parameters are the input of the functions. The function can declared with any arbitrary number of parameters -- even zero.

Then inside the function, you can use the parameters like what you normally would with variables.

Like every ***Variable Declaration,*** the parameter is declared with the name and **must annotate** the **type**.

Return

```
fn f(x: f64) -> f64 {
    return x * x;
}

fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x_squared + y_squared;
}
```

These are called `***Return Type Annotations***`.

It specifies the ***type*** of the value the function can ***return***. However, this is optional; if not specified, it means the function will ***return*** nothing.

This is called ***return,*** this is where the value of the function is returned (given out). When the `***return***` is executed, the function is stopped immediately.

Return

```
fn say_hello() {
    println!("hello!");
    return;
    println!("oh no...");
}
```

This function is declared with no parameter and without return type annotation (`***-> i32***` syntax).

You can use the plain `***return***` here (without the expression followed after) if the function doesn't have a return type annotation.

The function is stopped here; therefore, you won't see the text *"oh no..."* printed.

```
hello!
```

Return

The function is declared with return type annotation`i32`.

```
fn greater_than(a: i32, b: i32) -> i32 {
    return a > b;
}
```

However, this ***returns*** a ***`bool`*** value.

Therefore, this is an error!

**The code will be rejected by the Rust compiler*

Function Call

```rust
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```

This is called a `function call` ***expression***. This function call has `*3.0*` and `*4.0*` as ***arguments***.

When calling the function, you must provide a correct number of arguments and the type must match with the declared type in the parameter.

```
the result is 25
```

Function Call

```rust
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```

When the function is called, the execution flow will move to the called function

```
the result is 25
```

Function Call

```
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```

When the ***`return`*** hits, the execution flow goes back to the caller (the place that called the function).

The value `25.0` is returned with the function call.

```
the result is 25
```

Don’t Repeat Yourself (DRY)

Imagine you need to write a program that needs to print out the result of the following formula.

$$x^w + y^z$$

```rust
fn main() {
    let x: f64 = 3.0;
    let w: u32 = 2;

    let y: f64 = 4.0;
    let z: u32 = 2;

    let result = /*some code*/;

    /*expected output is 25*/
    println!("the result is {result}");
}
```

```rust
fn main() {
    let x: f64 = 3.0;
    let w: u32 = 2;

    let y: f64 = 4.0;
    let z: u32 = 2;

    let mut x_raised_w = 1.0;
    for i in 0..w {
        x_raised_w *= x;
    }

    let mut y_raised_z = 1.0;
    for i in 0..z {
        y_raised_z *= y;
    }

    let result = x_raised_w + y_raised_z;

    /*expected output is 25*/
    println!("the result is {result}");
}
```

This is the possible implementation...

```rust
fn main() {
    let x: f64 = 3.0;
    let w: u32 = 2;

    let y: f64 = 4.0;
    let z: u32 = 2;

    let mut x_raised_w = 1.0;
    for i in 0..w {
        x_raised_w *= x;
    }

    let mut y_raised_z = 1.0;
    for i in 0..z {
        y_raised_z *= y;
    }

    let result = x_raised_w + y_raised_z;

    /*expected output is 25*/
    println!("the result is {result}");
}
```

However, these two blocks of code are very similar; the logic is identical but has only different variable names.

This is called ***code duplication***, which is *generally a bad practice*.

```rust
fn exponent(base: f64, exponent: u32) -> f64 {
    let mut result = 1.0;

    for i in 0..exponent {
        result *= base;
    }

    return result;
}

fn main() {
    let x: f64 = 3.0;
    let w: u32 = 2;

    let y: f64 = 4.0;
    let z: u32 = 2;

    let result = exponent(x, w) + exponent(y, z);

    println!("the result is {result}");
}
```

The exponentiation logic is rewritten into a function.

So that we can utilize the exponentiation logic easily

Why Code Duplication is Bad

```
fn main() {
    let x: f64 = 3.0;
    let w: u32 = 2;

    let y: f64 = 4.0;
    let z: u32 = 2;

    let mut x_raised_w = 1.0;
    for i in 0..w {
        x_raised_w *= x;
    }

    let mut y_raised_z = 1.0;
    for i in 0..z {
        y_raised_z *= y;
    }

    let result = x_raised_w + y_raised_z;

    /*expected output is 25*/
    println!("the result is {result}");
}
```

A good developer should avoid code duplication most of the time.

Writing code involves maintaining them as well.

Imagine the code written contains some bugs and then is duplicated all over the place. When it comes to the time you need to fix them, you'll need to find those mistakes multiple times.

Call Stack

```
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```

When running a program, the computer allocates memory for us. This memory is where all our variables are stored; it's called the ***Stack***.

When you call a function, a new ***Stack Frame*** is added.

Call Stack Visualization

```rust
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```

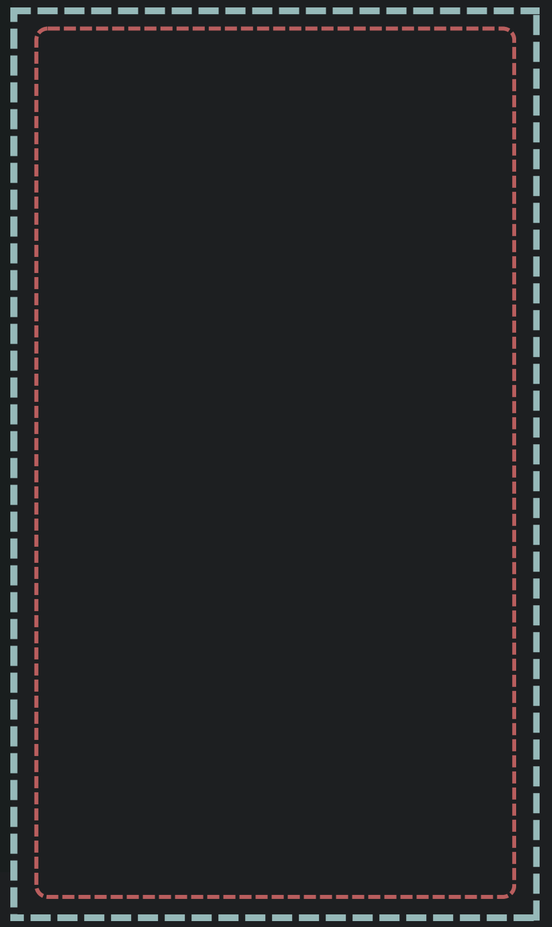
You might've noticed that the `*fn main*` we usually write is a ***function***. The `*fn main*` is the first function that the program will run. Think of it as the starting point.

Call Stack Visualization

```
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```



This is our **Stack** memory (typically a part of RAM).

The memory is laid out in a contiguous row.

You can think of a village that has thousands of houses built in a row.

Call Stack Visualization

```rust
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```

When the `***fn main***` is called, a new *stack frame* is added at the bottom.

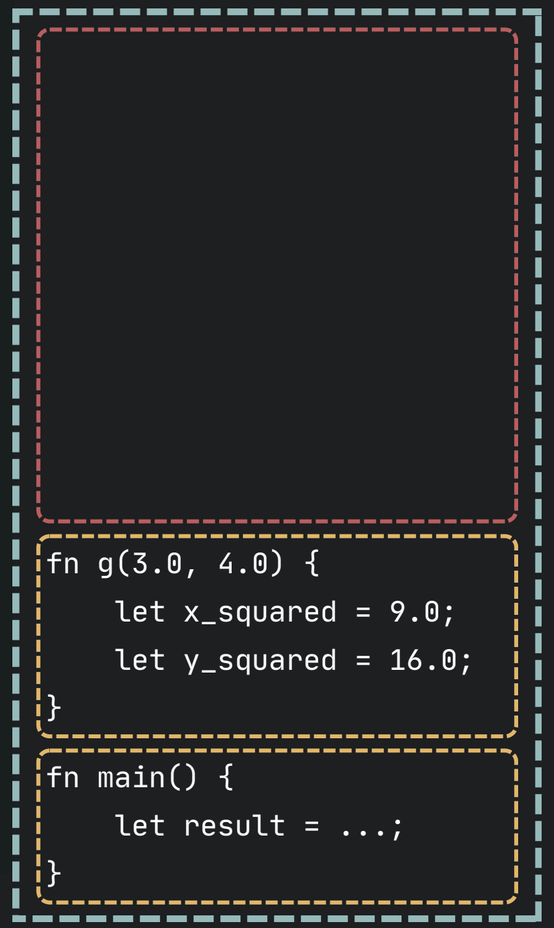
This stack frame includes all the variables in that particular function.

Call Stack Visualization

```rust
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```



When the code executes the function call `***g(3.0, 4.0)***` there, again, a new stack frame is created for that function call.

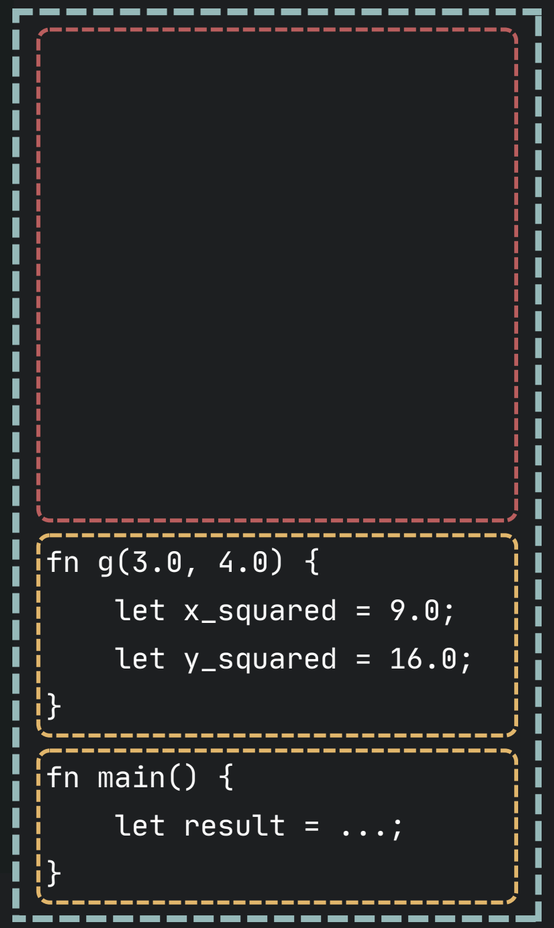
A new stack frame will be **Pushed** (added) on top of the previous `***fn main***` stack frame.

Call Stack Visualization

```rust
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```



After the `***return***` is executed or the function ends, the stack frame will be ***Popped*** (removed).

Call Stack Visualization

```rust
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```

Now the, `g(3.0, 4.0)` stack frame has been popped.

Now there's only ***`fn main`*** stack frame.

Call Stack Visualization

```rust
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```

After the `*println!(“..”)*`, which is the last statement, the `*fn main*` stack frame will be popped as well, finishing the program.

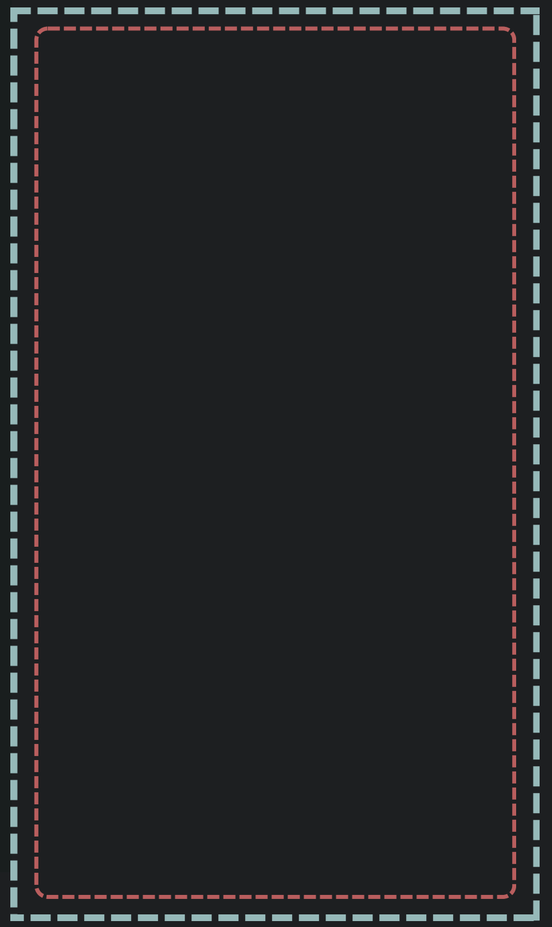
Call Stack Visualization

```rust
fn g(x: f64, y: f64) -> f64 {
    let x_squared = x * x;
    let y_squared = y * y;

    return x + y;
}

fn main() {
    let result = g(3.0, 4.0);
    println!("the result is {result}");
}
```

STACK



Done ...

This stack frames push/pop mechanism happens automatically

```
the result is 25
```

Recursion

```
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

Can you guess what does this program do?

Recursion

```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

Let’s look at the stack visualization.

The program first starts with the `***fn main***`.

Recursion

```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

When the `***say(1)***` is called,
a new stack frame is added.

Recursion

n: 1

```
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

STACK

The `***println!***` statement is executed so we’ll see the text *“n: 1”*.

Recursion

n: 1

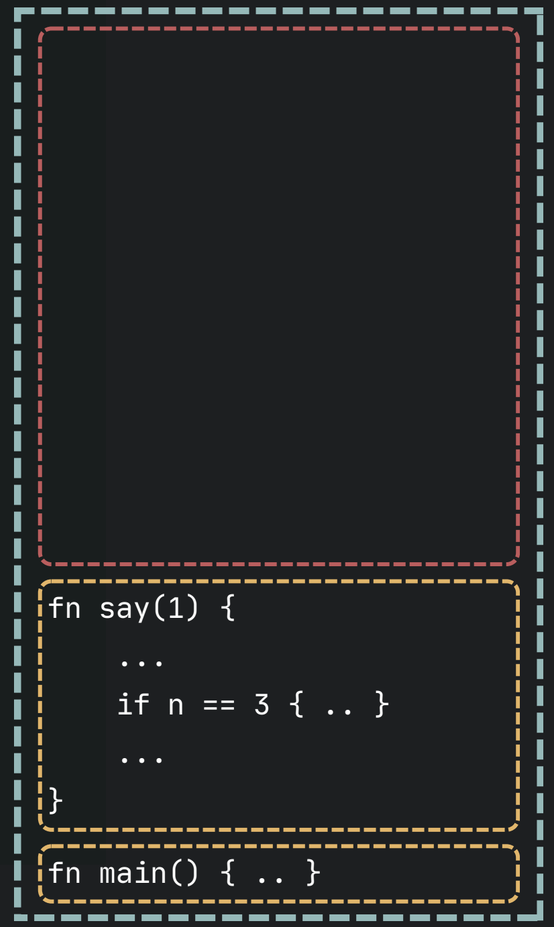
```
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

STACK



Since `*n*` doesn't equal to 3, this `*if*` will be skipped.

Recursion

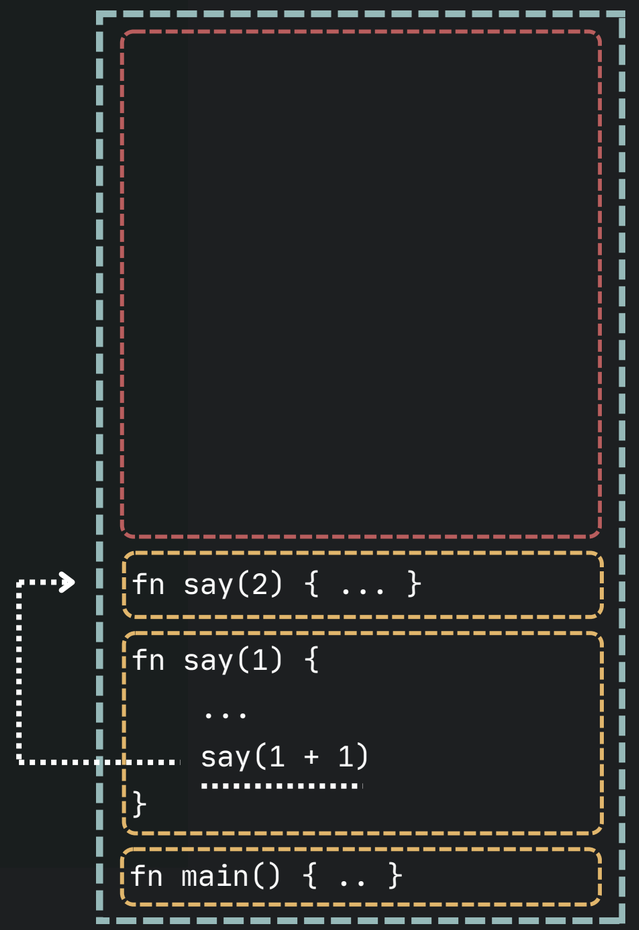
n: 1

```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```



The function `***say(2)***` is called, so there will be a new stack frame.

Recursion

```
n: 1
n: 2
```

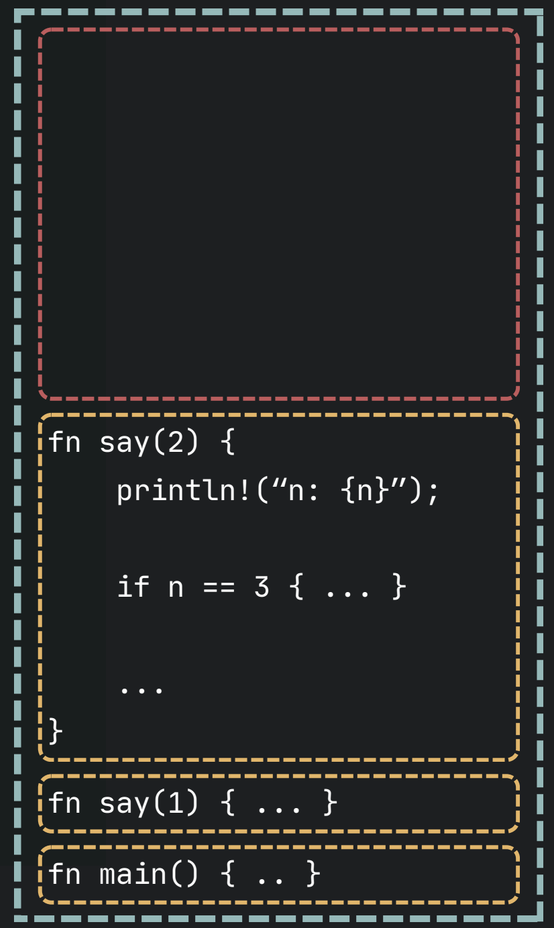
```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

STACK



Now, this function `***say(2)***` runs similarly to the `***say(1)***`. The `***println!***` is executed and the `***if n == 3***` is skipped.

Recursion

```
n: 1
n: 2
```

```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

STACK

Again, the function `***say(3)***` is called, so there will be a new stack frame.

Recursion

```
n: 1
n: 2
n: 3
```

```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

STACK

Again, the ***`println!`*** is executed, so we’ll see the text *“n: 3”*.

Recursion

```
n: 1
n: 2
n: 3
```

```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

STACK

This time, the ***`if n == 3`*** is satisfied so the ***`return`*** is executed, which means the function ends now, so there won't be another ***`say(3 + 1)`*** function call.

Recursion

```
n: 1
n: 2
n: 3
```

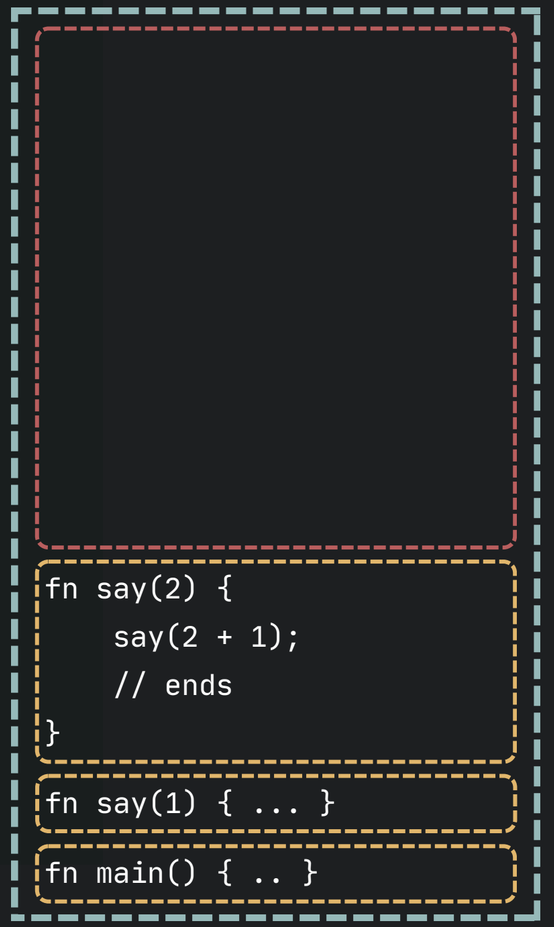
```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

STACK



Therefore, the `***say(3)***` stack frame is popped.

The `***say(2)***` will also soon be popped.

Recursion

```
n: 1
n: 2
n: 3
```

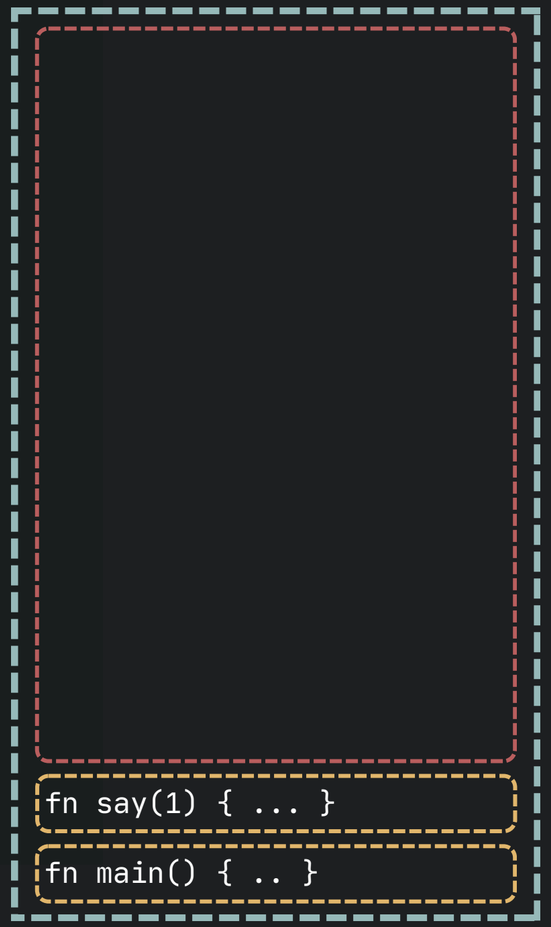
```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

STACK



And so will `***say(1)***`

Recursion

```
n: 1
n: 2
n: 3
```

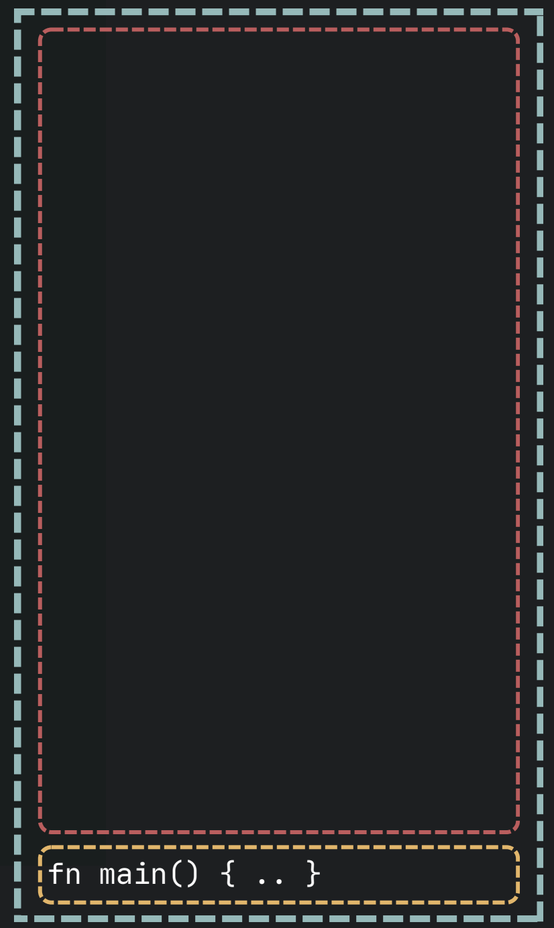
```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

STACK



Finally, the `***main()***` will be popped, finishing the program.

Recursion

```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

The function `***say***` is called inside the function `***say***`.

This is called ***Recursion***, it's when the function calls itself.

Recursion

```rust
fn say(n: i32) {
    println!("n: {n}");

    if n == 3 {
        return;
    }

    say(n + 1);
}

fn main() {
    say(1);
}
```

A recursive function typically contains two parts: *the base case/termination* and *the recursive call*

This is the *termination*. It specifies that when the `***n***` is 3, the function will not call itself again.

This is the recursive call, where the function calls itself.

Stackoverflow

```rust
fn say(n: i32) {
    println!("n: {n}");
    say(n + 1);
}

fn main() {
    say(1);
}
```

This is a recursive function without the base-case/termination.

There’s only the recurisve call

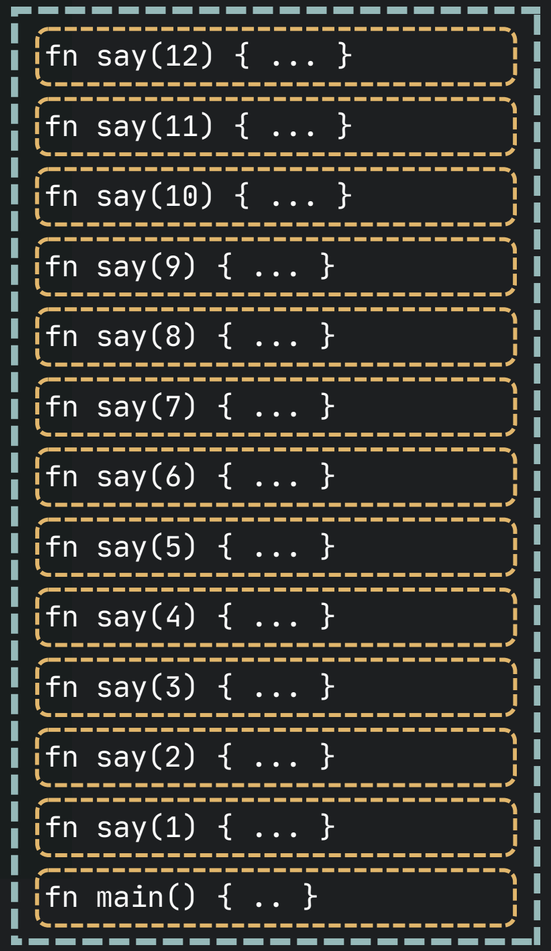
You might think it will print out the number infinitely similar to the `***while true***`.

However, it’s most likely that the program won’t run very long since it will cause ***Stackoverflow.***

Stackoverflow

```rust
fn say(n: i32) {
    println!("n: {n}");
    say(n + 1);
}

fn main() {
    say(1);
}
```



The stack will be rapidly filled with many stack frames generated indefinitely.

And since the stack memory is ***limited***, the program wouldn't be able to accommodate a new stack frame anymore.

Therefore, the program will crash and this is called ***Stackoverflow***.

Example use Case of Recursion

$$4! = 4 \times 3 \times 2 \times 1$$

This is called ***Factorial,*** where you multiply all the ***positive integers (except zero)*** that are less than or equal to the operand.

Surprisingly, you can use recursion to describe ***Factorial***.

Example use Case of Recursion

The Factorial can be written as conditional function like this.

$$f(0) = 1 \qquad (1)$$

This is the base case, saying that 0! = 1.

$$f(n) = n \times \underline{f(n-1)} \quad n > 0 \qquad (2)$$

If the `n>0` this case will be used.

This is the recursive call

Example use Case of Recursion

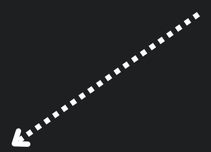
For instance, let’s compute 4!

$$f(0) = 1$$

$$f(n) = n \times f(n-1) \quad n > 0$$

$$(1) \qquad f(4) = 4 \times f(3)$$

$$(2)$$



Example use Case of Recursion

$$f(0) = 1 \qquad (1)$$

$$f(n) = n \times f(n-1) \quad n > 0 \qquad (2)$$

$$f(4) = 4 \times f(3)$$

$$f(3) = 3 \times f(2)$$

Computing `**f(3)**`

Example use Case of Recursion

$$f(0) = 1 \qquad (1)$$

$$f(n) = n \times f(n-1) \quad n > 0 \qquad (2)$$

$$f(4) = 4 \times f(3)$$

$$f(3) = 3 \times f(2)$$

$$f(2) = 2 \times f(1)$$

Computing `**f(2)**`

Example use Case of Recursion

$$f(0) = 1 \quad (1)$$

$$f(n) = n \times f(n-1) \quad n > 0 \quad (2)$$

$$f(4) = 4 \times f(3)$$

$$f(3) = 3 \times f(2)$$

$$f(2) = 2 \times f(1)$$

$$f(1) = 1 \times f(0)$$

Computing `**f(1)**`

Example use Case of Recursion

$$f(0) = 1 \quad (1)$$

$$f(n) = n \times f(n-1) \quad n > 0 \quad (2)$$

$$f(4) = 4 \times f(3)$$

$$f(3) = 3 \times f(2)$$

$$f(2) = 2 \times f(1)$$

$$f(1) = 1 \times f(0)$$

$$f(0) = 1$$

We've reached the base case

Example use Case of Recursion

$$f(0) = 1 \qquad (1)$$

$$f(n) = n \times f(n-1) \quad n > 0 \qquad (2)$$

$$f(4) = 4 \times f(3)$$

$$f(3) = 3 \times f(2)$$

$$f(2) = 2 \times f(1)$$

$$f(1) = 1 \times 1$$

$$f(0) = 1$$

Substituting `**f(0)**`

Example use Case of Recursion

$$f(0) = 1 \qquad (1)$$

$$f(n) = n \times f(n-1) \quad n > 0 \qquad (2)$$

$$f(4) = 4 \times f(3)$$

$$f(3) = 3 \times f(2)$$

$$f(2) = 2 \times \underline{1 \times 1}$$

$$f(1) = 1 \times 1$$

Substituting `***f(1)***`

Example use Case of Recursion

$$f(0) = 1 \qquad (1)$$

$$f(n) = n \times f(n-1) \quad n > 0 \qquad (2)$$

$$f(4) = 4 \times f(3)$$

$$f(3) = 3 \times \underline{2 \times 1 \times 1}$$

$$f(2) = 2 \times 1 \times 1$$

Substituting `**f(2)**`

Example use Case of Recursion

$$f(0) = 1 \qquad (1)$$

$$f(n) = n \times f(n-1) \quad n > 0 \qquad (2)$$

$$f(4) = 4 \times \underline{3 \times 2 \times 1 \times 1}$$

$$f(3) = 3 \times 2 \times 1 \times 1$$

Substituting `**f(3)**`

Example use Case of Recursion

$$f(0) = 1 \tag{1}$$

$$f(n) = n \times f(n-1) \quad n > 0 \tag{2}$$

$$f(4) = 4 \times 3 \times 2 \times 1 \times 1$$

Done computing factorial using recursion.