GOLANG
FOR
BEGINNE

by TAM SE

RUST FOR BEGINNERS

by TAM SEL

RUST AND GOLANG FOR BEGINNERS

LEARN TO CODE FAST

BY

TAM SEL

RUST
FOR
BEGINNERS

LEARN TO CODE FAST

BY

TAM SEL

Rust Programming Language

Our Rust programming language is illustrated for the beginners and professionals. Rust programming language is designed to provide better memory safety, but it is still under the maintenance process.

What is Rust?

- Rust is a system programming language developed by a Mozilla employee ***"Graydon Hoare" in 2006***. He described this language as a **"safe, concurrent and practical language"** that supports the functional and imperative paradigm.
- The syntax of rust is similar to the C++ language.
- Rust is **free and open source software**, i.e., anyone can use the software freely, and the source code is openly shared so that the people can also improve the design of the software.
- Rust is declared as one of the ***"most loved programming language"*** in the stack overflow developer survey in 2016, 2017 and 2018.
- There is no direct memory management like calloc or malloc. It means, the memory is managed internally by Rust.

Rust is for

Rust language is ideal for many people for many reasons.

Let's look:

- **Team of developers:** Rust proves to be quite useful for the "team of developers". Low- level programming code contains bugs which need extensive testing by the testers. However, in case of Rust, compiler refuses to compile the code if the program contains bugs. By working parallel to the compiler, the developer can focus on the program's logic rather than focusing on the bugs.
- **Students:** Using Rust, many people can learn how to develop the operating system. The Rust team is trying to make the system concepts more accessible to the ordinary people, especially for those who are new to the programming.
- **Companies:** Large or small companies use Rust to accomplish various tasks. These tasks include command line tools, web services, DevOps tooling, embedded devices, audio, and video analysis and transcoding, cryptocurrencies, bioinformatics, search engines, Internet of Things applications, machine learning, and even significant parts of the Firefox web browser.
- **Open source developers:** Rust is an open source language means that the source code is available to the people. Therefore, they can use the source code to improve the design of Rust.

Features of Rust

Rust is a system programming language. Rust provides the following features:

1. Zero cost abstraction
2. Error messages
3. Move semantics
4. Threads without data races
5. Pattern matching
6. Guaranteed memory safety
7. Efficient C bindings
8. Safe memory space allocation
9. Minimal time

1. Zero cost abstraction

In Rust, we can add abstractions without affecting the runtime performance of the code. It improves the code quality and readability of the code without any runtime performance cost.

2. Error messages

In C++ programming, there is an excellent improvement in error messages as compared to GCC. Rust goes one step further in case of clarity. Error messages are displayed with (formatting, colors) and also suggest misspellings in our program.

3. Type inference

Rust provides the feature of a Type inference which means that it determines the type of an expression automatically.

4. Move semantics

Rust provides this feature that allows a copy operation to be replaced by the move operation when a source object is a temporary object.

5. Threads without data races

A data race is a condition when two or more threads are accessing the same memory location. Rust provides the feature of threads without data races because of the ownership system. Ownership system transmits only the owners of different objects to different threads, and two threads can never own the same variable with write access.

6. Pattern matching

Rust provides the feature of pattern matching. In pattern matching, patterns in Rust are used in conjunction with the 'match' expressions to give more control over the program's control flow. Following are the combinations of some patterns:

- Literals
- Arrays, enums, structs, or tuples
- Variables
- Wildcards
- Placeholders

7. Guaranteed memory safety

Rust guaranteed the memory safety by using the concept of ownership. Ownership is a middle ground between the memory control of C and the garbage *collection of java*. In Rust programs, memory space is owned by the variables and temporarily borrowed by the other variables. This allows Rust to provide the memory safety at the compile time without relying on the garbage collector.

8. Efficient C bindings

Rust provides the feature of *'Efficient C bindings'* means that the Rust language can be able to interoperate with the C language as it talks to itself. Rust provides a 'foreign function interface' to communicate with C API's and leverage its **ownership** system to guarantee the memory safety at the same time.

9. Safe memory space allocation

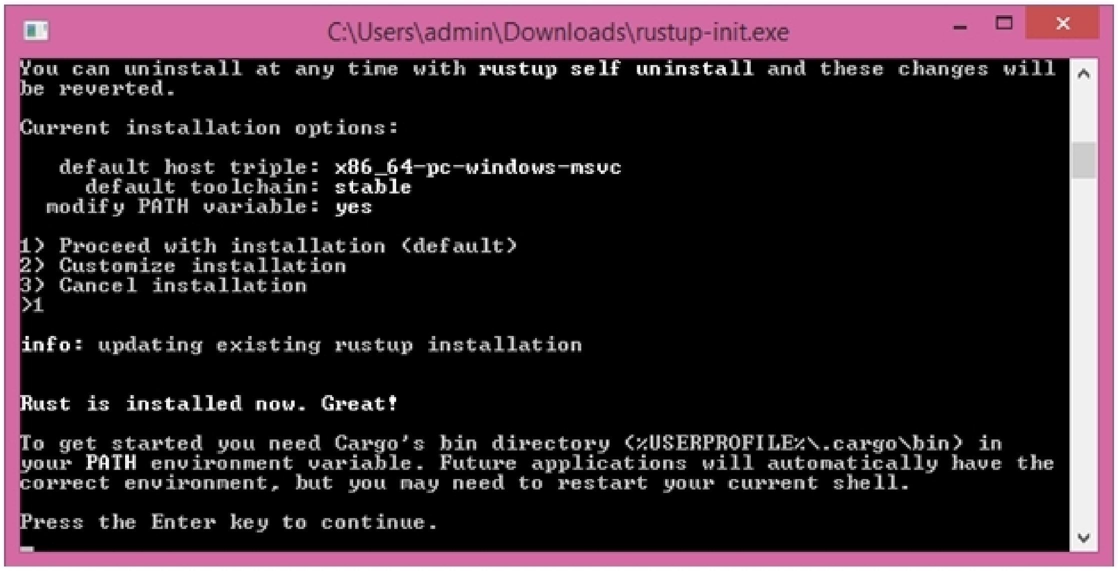
In Rust, memory management is manual, i.e., the programmer has explicit control over where and when memory is allocated and deallocated. In C language, we allocate the memory using malloc function and then initialize it but Rust refuses these two operations by a single **'~'** operator. This operator returns the smart pointer to int. A smart pointer is a special kind of value that controls when the object is freed. Smart pointers are *"smart"* because they not only track where the object is but also know how to clean it up.

Rust Installation

The first step is to install Rust. First of all, download Rust through **rustup** which is a command line tool for managing all the Rust versions and its associated tools.

Rust Installation in Windows

- On Windows, open the link https://www.rust-lang.org/install.html and follow the instructions for installing Rust. After following all the instructions, Rust will get installed and the screen appears:



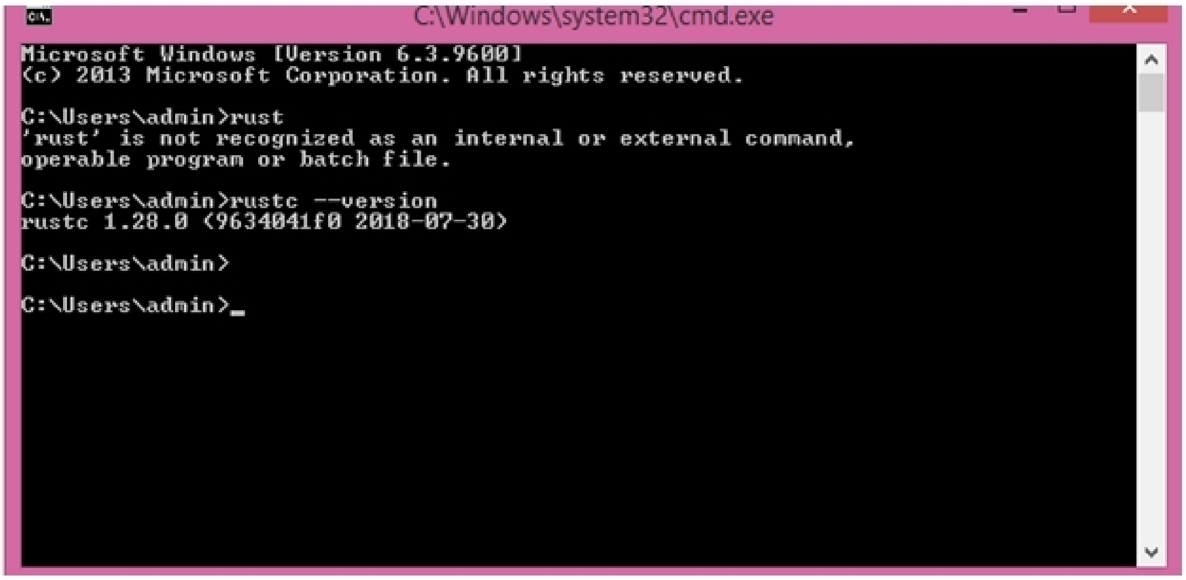
```
You can uninstall at any time with rustup self uninstall and these changes will
be reverted.

Current installation options:

   default host triple: x86_64-pc-windows-msvc
     default toolchain: stable
  modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
>1

info: updating existing rustup installation


Rust is installed now. Great!

To get started you need Cargo's bin directory (%USERPROFILE%\.cargo\bin) in
your PATH environment variable. Future applications will automatically have the
correct environment, but you may need to restart your current shell.

Press the Enter key to continue.
```

- After installation, PATH variable of Rust automatically adds in your system PATH.
- Open command prompt then runs the following command:

1. $ rustc --version

After running this command, you should see the version number, commit hash, and commit date.

If you do, it means Rust has been installed successfully. Congratulations!!!



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>rust
'rust' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\admin>rustc --version
rustc 1.28.0 (9634041f0 2018-07-30)

C:\Users\admin>

C:\Users\admin>_
```

Rust Installation in Linux or macOS

If you are using Linux or macOS, open a terminal then use the following command:

```
1. $ curl https://sh.rustup.rs -sSf | sh
```

- The above command downloads a script and starts the installation of rustup tool. This installs the latest version of Rust. If the installation is done successfully, then the following message will appear:

```
1. Rust is installed now. Great!
```

- This installation adds automatically Rust to your system path after your next login. If you want to run Rust just right away without restarting the terminal, then run the following command to your shell to add the path to your system PATH manually:

```
1. $ source $HOME/.cargo/env
```

After installation, you need a linker. When you try to run your Rust program, you will get the error that a linker could not execute. It means that the linker is not installed in your system. C compilers always come up with the correct compiler. Install a C compiler. Also, some of the Rust packages depends upon the C code and will need a C compiler.

Updating & Uninstalling

Update: After you have installed your Rust through **"rustup"**, updating to the latest version. Run the following command to update to the latest version:

```
1. $ rustup update
```

Uninstall: If you want to uninstall your Rust then runs the following command from the shell:

```
1. $ rustup self uninstall
```

First Rust program

Let's write the simple program in Rust language. Now, open the notepad file and write the following code:

```
fn main()

    println!("Hello, world!");
```

Output:

```
Hello, world!
```

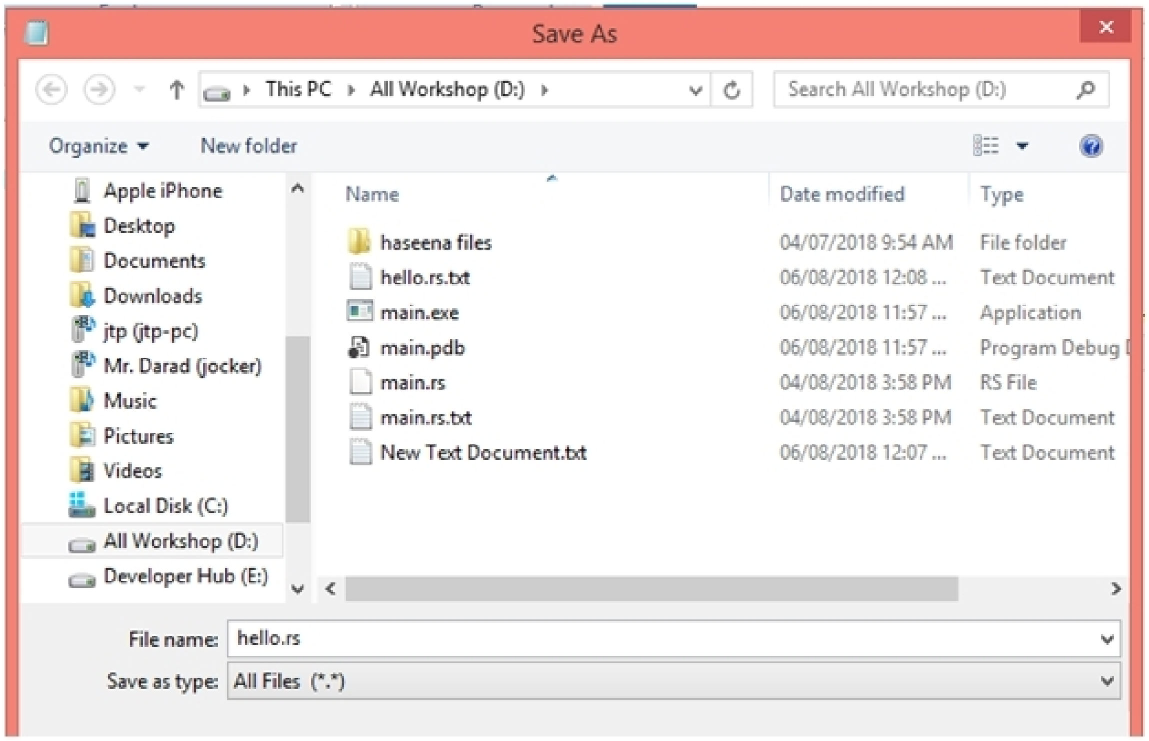
main(): The main() function is always the first code in every Rust executable code. The main() function is enclosed in curly braces{}. The main() function does not contain any parameter as well as it does not return any value.

println!: It is a Rust macro. If it calls the function, then it does not contain '!'.

"Hello World": It is a string passed as a parameter to the println!, and the string is printed to the console.

Procedure to create, compile and run the program

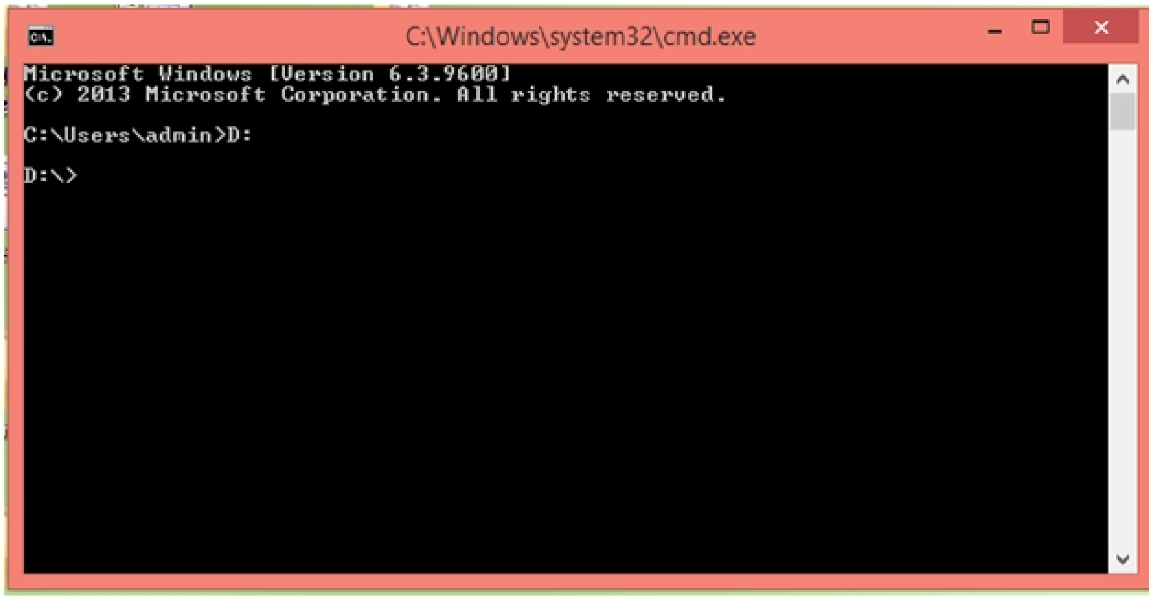
1. Open the notepad file and write the code in a notepad file.
2. Save the file with *.rs* extension.



Name	Date modified	Type
haseena files	04/07/2018 9:54 AM	File folder
hello.rs.txt	06/08/2018 12:08 ...	Text Document
main.exe	06/08/2018 11:57 ...	Application
main.pdb	06/08/2018 11:57 ...	Program Debug
main.rs	04/08/2018 3:58 PM	RS File
main.rs.txt	04/08/2018 3:58 PM	Text Document
New Text Document.txt	06/08/2018 12:07 ...	Text Document

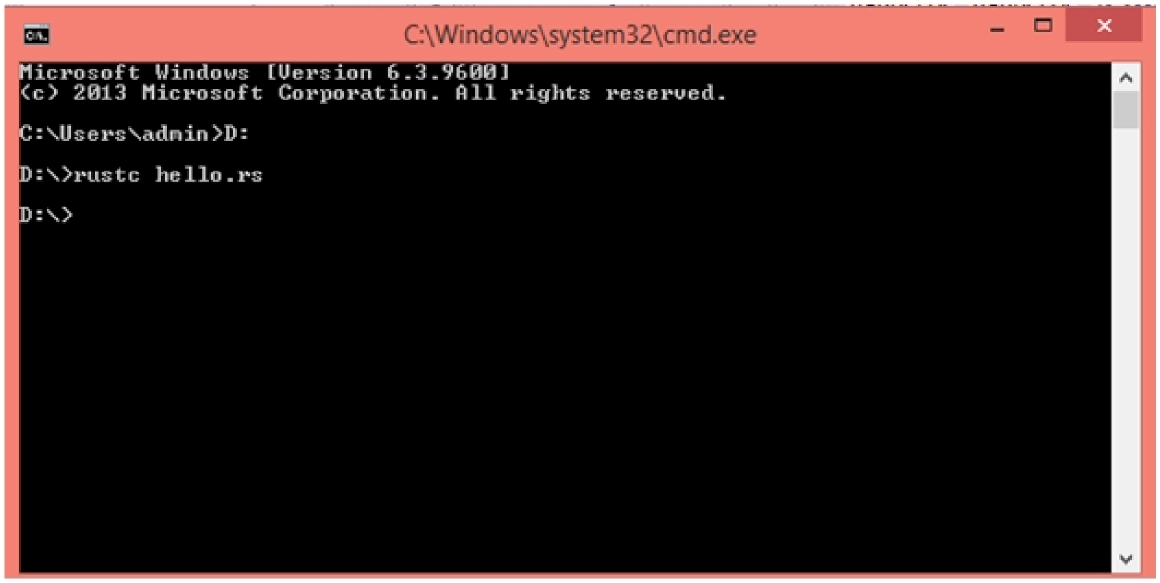
File name: hello.rs
Save as type: All Files (*.*)

3. Open the command prompt
4. Set the path of the directory. Suppose the project is located in D drive.



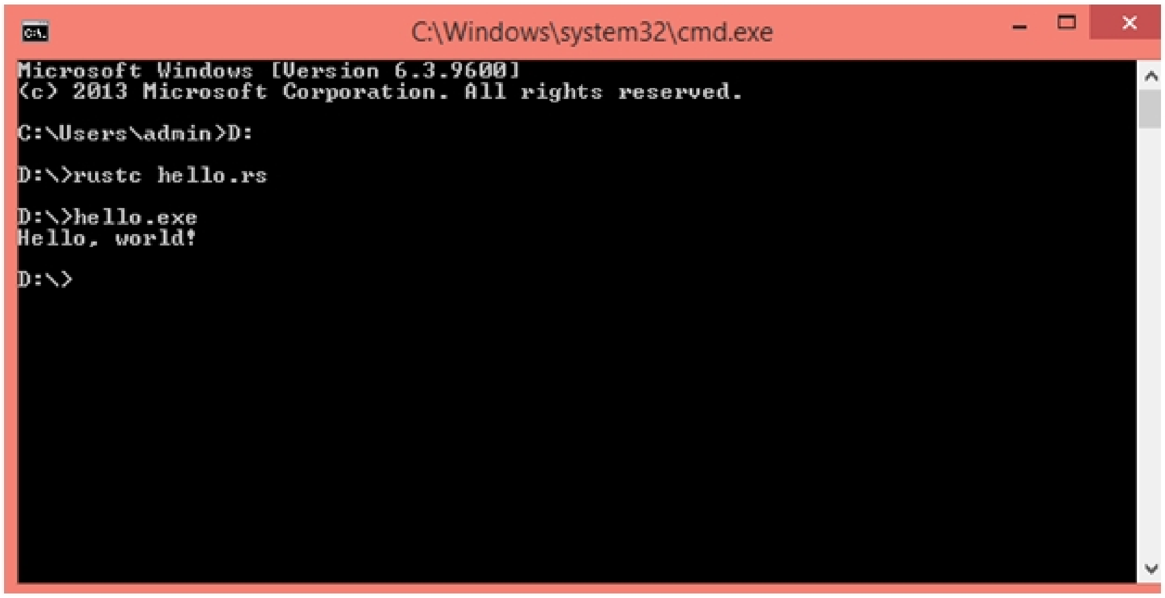
```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>D:

D:\>
```

5. Compile the above program using the rustc command.



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>D:

D:\>rustc hello.rs

D:\>
```

6. Finally, run the program by using the command **filename.exe**.



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>D:

D:\>rustc hello.rs

D:\>hello.exe
Hello, world!

D:\>
```

'If' statement

The 'if' statement determines whether the condition is true or not. If condition is true then the 'if' block is executed otherwise, control skips the 'if' block.

Different ways to represent 'if' block:

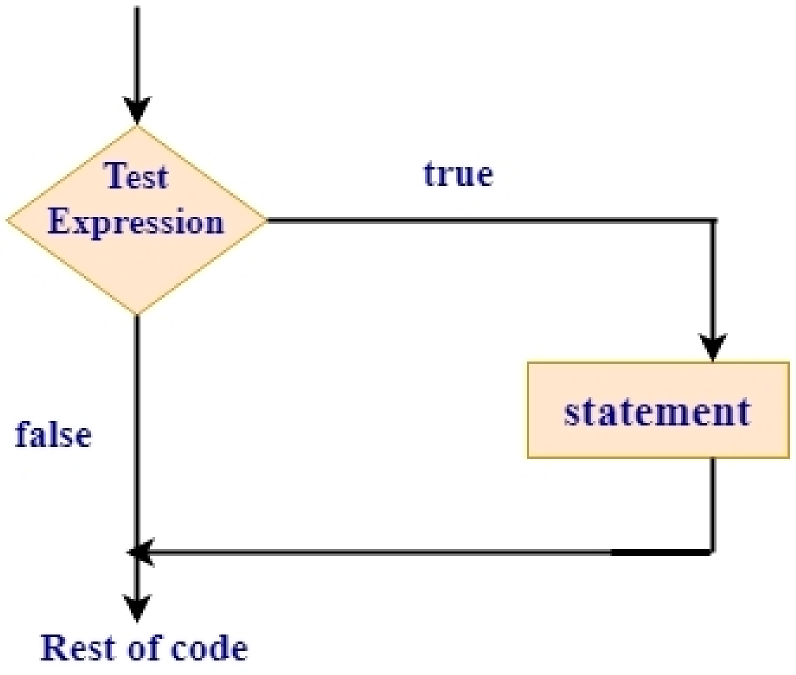
- if block
- if-else block
- if else-if ladder
- nested if

Syntax of 'if':

```
if condition
 {
          //block statements;
}
```

In the above syntax, if the condition is true then the block statements are executed otherwise it skips the block.

Flow Chart of "if statement"



For Example:

Let's see a simple example of 'if' statement.

```
fn main()

 let a=1;
 if a==1
  {
     println!("a is equal to 1");
  }
```

Output:

```
a is equal to 1
```

In this example, value of **a** is equal to 1. Therefore, condition given in **'if'** is true and the string passed as a parameter to the println! is displayed on the console.

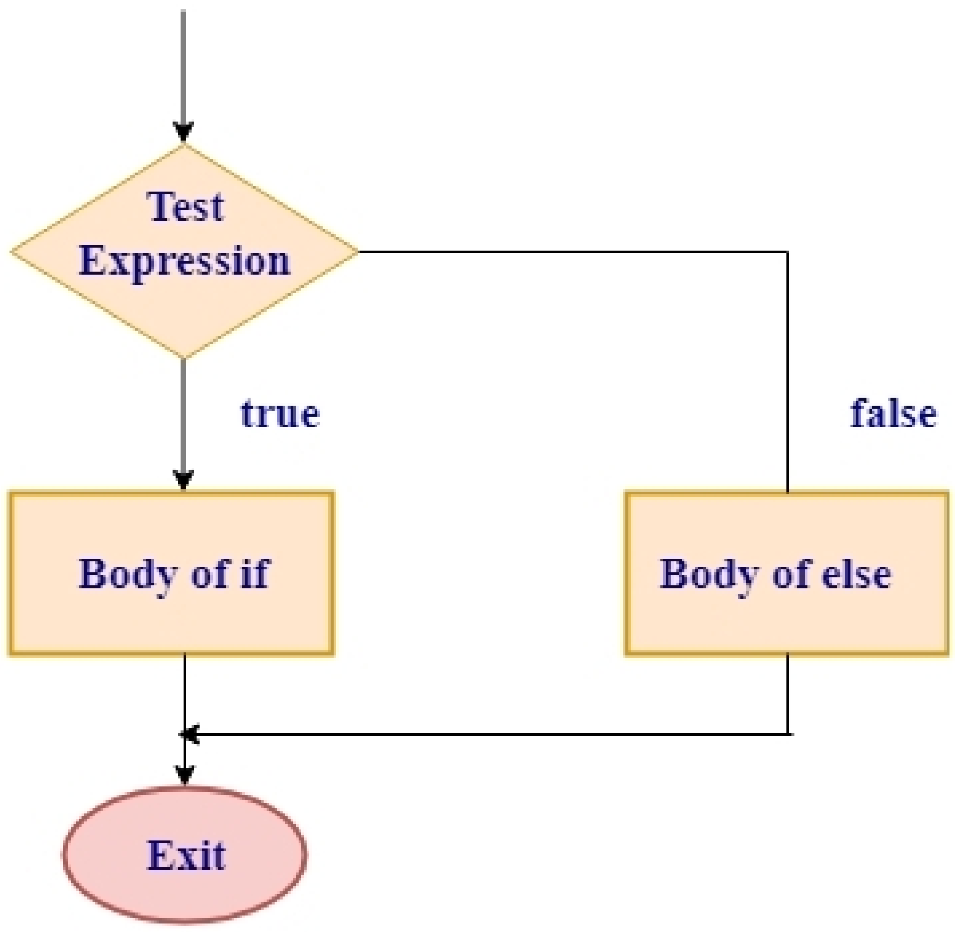
"if-else"

If the condition is true then 'if' block is executed and the statements inside the 'else' block are skipped. If the condition is false then 'else' block is executed and the statements inside the 'if' block are skipped.

Syntax of "if-else"

```
 if condition
{
  //block statements
}
else
{
   //block statements
 }
```

Flow Chart of "if-else"



Let's see a simple example of 'if-else' statement.

```
fn main()
{
 let a=3;
 let b=4;
 if a>b
 {
    println!("a is greater than b");
 }
 else
  {
    println!("a is smaller than b");
  }
}
```

Output:

```
a is smaller than b
```

In this example, value of a is equal to 3 and value of a is less than the value of b. Therefore, else block is executed and prints "a is smaller than b" on the screen.

else-if

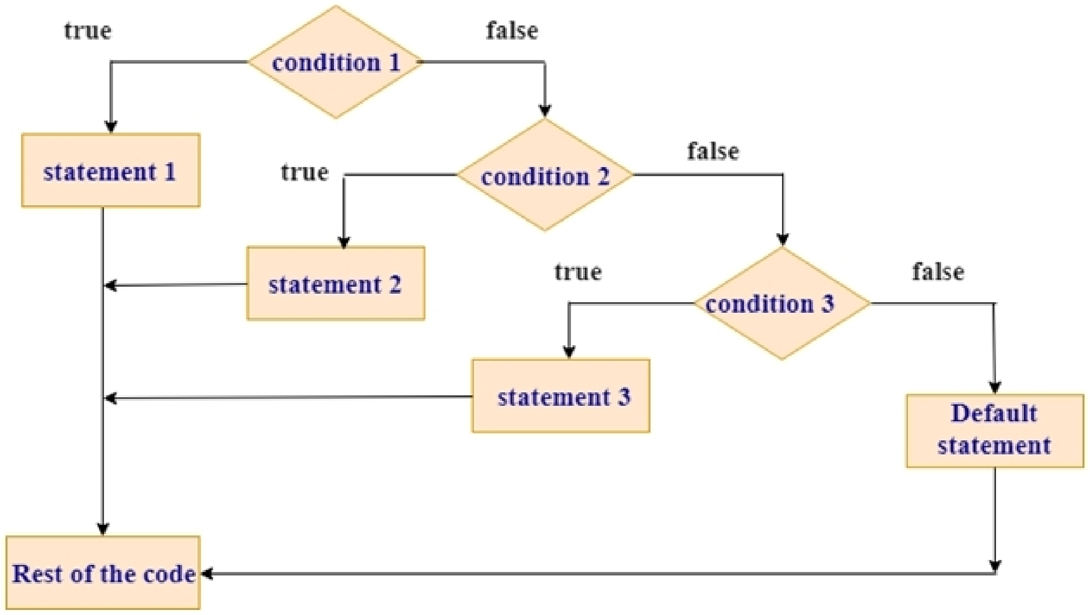
When you want to check the multiple conditions, then 'else-if' statement is used.

Syntax of else-if

```
 if condition 1
{
  //block statements
}
else if condition 2
{
  //block statements
}
.
.
else{
//block statements
}
```

In the above syntax, Rust executes the block for the first true condition and once it finds the first true condition then it will not execute the rest of the blocks.

Flow Chart of "else if"



Let's see a simple example of else-if statement

```
fn main()

let num= -5;
if num>0
{
  println!("number is greater than 0");
}
else if num<0
{
  println!("number is less than 0 ");
}
else
{
  println!("number is not equal to 0");
}
```

Output:

```
number is less than 0
```

In this example, value of num is equal to -5 and num is less than 0. Therefore, else if block is executed.

Nested if-else

When an if-else statement is present inside the body of another if or else block then it is known as nested if-else.

Syntax of Nested if-else

```
 if condition 1
{
  // block statements
  if condition 2
{
     //block statements
}
else
{
   //block statements
}
}
else
{
  //block statements
}
```

Let's see a simple example of nested if-else

```
fn main()

 let a=5;
 let b=6;
 if a!=b
  {
   if a>b
    {
     println!("a is greater than b");
    }
   else
    {
     println!("a is less than b");
    }
}

 else
 {
     println!("a is equal to b");
 }
```

Output:

```
a is less than b
```

In this example, value of a is not equal to b. So, control goes inside the 'if' block and the value of a is less than b. Therefore, 'else' block is executed which is present inside the 'if' block.

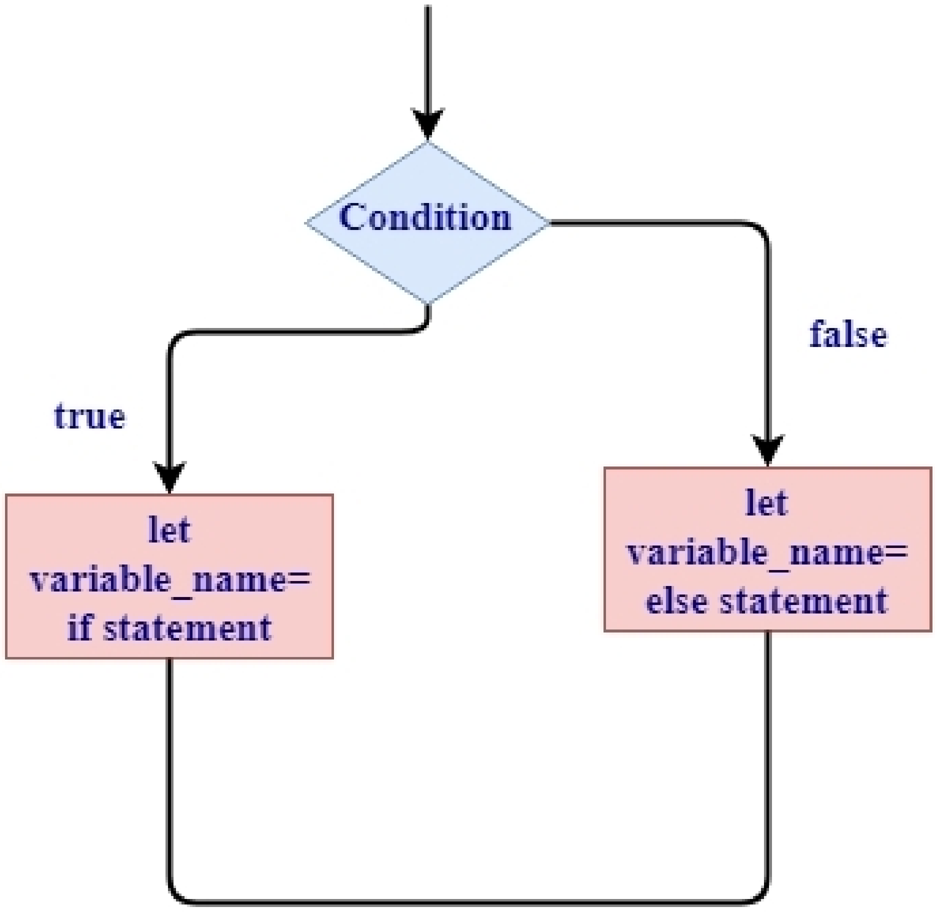
Using "if in a let" statement

An **'if'** expression is used on the right hand side of the let statement and the value of **'if'** expression is assigned to the **'let'**statement.

Syntax of 'if in a let'

```
1. Let variable_name= if condition{
2. //code blocks
3. }
4. else{
5. //code block
6. }
```

In the above syntax, if the condition is true then the value of 'if' expression is assigned to the variable and if the condition is false then the value of 'else' is assigned to the variable.


Condition
true
false
let variable_name= if statement
let variable_name= else statement

Example 1

Let's see a simple example.

```
fn main()

 let a=if true
       {
          1
       }
       else
       {
           2
       };

 println!("value of a is: {}", a);
```

Output:

```
value of a is: 1
```

In this example, condition is true. Therefore, 'a' variable bounds to the value of 'if' expression. Now, a consist 1 value.

Let's see a another simple example.

```
fn main()

 let b=if false
       {
          9
       }
       else
       {
           "Rustlang"
       };

 println!("value of a is: {}", a);
```

Output:

```
Some errors occurred:E0308
```

In this example, 'if' block evaluates to an integer value while 'else' block evaluates to a string value. Therefore, this program throws an error as both the blocks contains the value of different type.

Loops

If we want to execute the block of statements more than once, then loops concept comes under the role. A loop executes the code present inside the loop body till the end and starts again immediately from the starting.

Rust consists of three kinds of loops:

- loops
- for loop
- while loop

loop

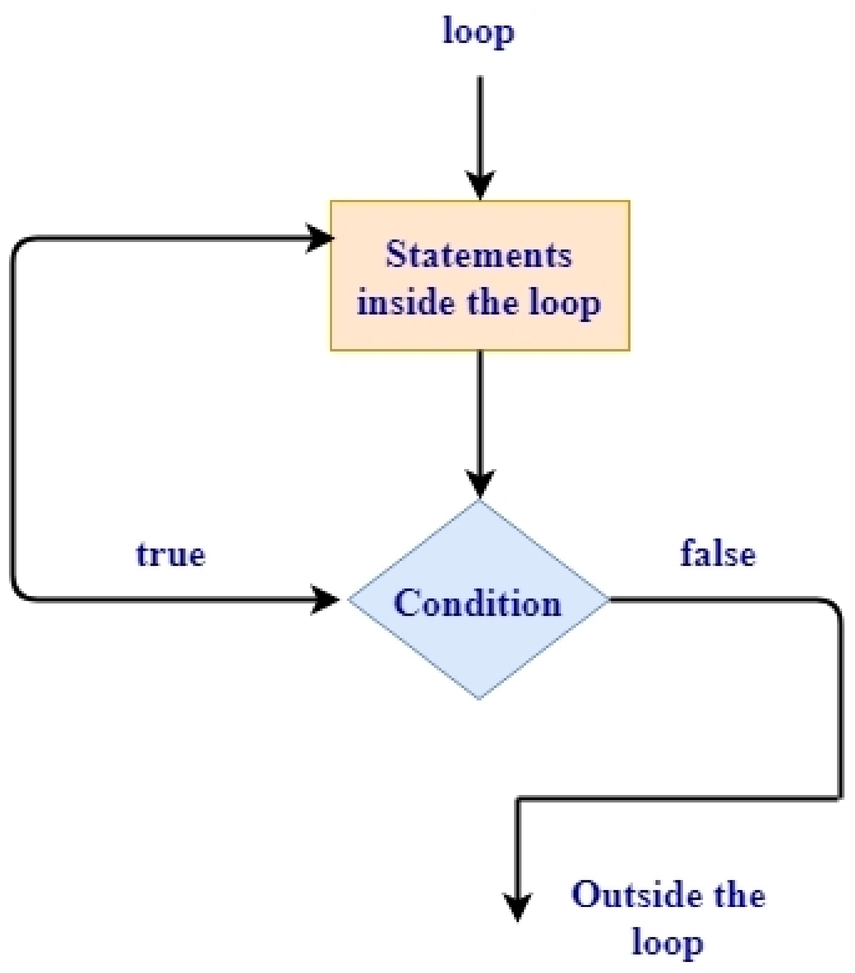
The loop is not a conditional loop. It is a keyword that tells the Rust to execute the block of code over again and again until and unless you explicitly stop the loop manually.

Syntax of loop

```
loop{
  //block statements
}
```

In the above syntax, block statements are executed infinite times.

Flow diagram of loop:



Let's see a simple example of infinite loop

```
fn main()

 loop
 {
    println!("Hello Rustlang");
}
```

Output:

```
Hello Rustlang
Hello Rustlang
Hello Rustlang
Hello Rustlang
.
.
.
infinite times
```

In this example, "Hello Rustlang" is printed over and over again until and unless we stop the loop manually. Generally, "ctrl+c" command is used to terminate from the loop.

Termination from loops

The 'Break' keyword is used to terminate from the loop. If 'break' keyword is not used then the loop will be executed infinite times.

Let's see a simple example

```
fn main()

 let mut i=1;
 loop
 {
      println!("Hello Rustlang");
      if i==7
      {
        break;
      }
 i+=1;
 }}
```

Output:

```
Hello Rustlang
Hello Rustlang
Hello Rustlang
Hello Rustlang
Hello Rustlang
Hello Rustlang
Hello Rustlang
```

In the above example, i is a counter variable, and it is a mutable variable which conveys that the counter variable can be changed for the future use.

While loop

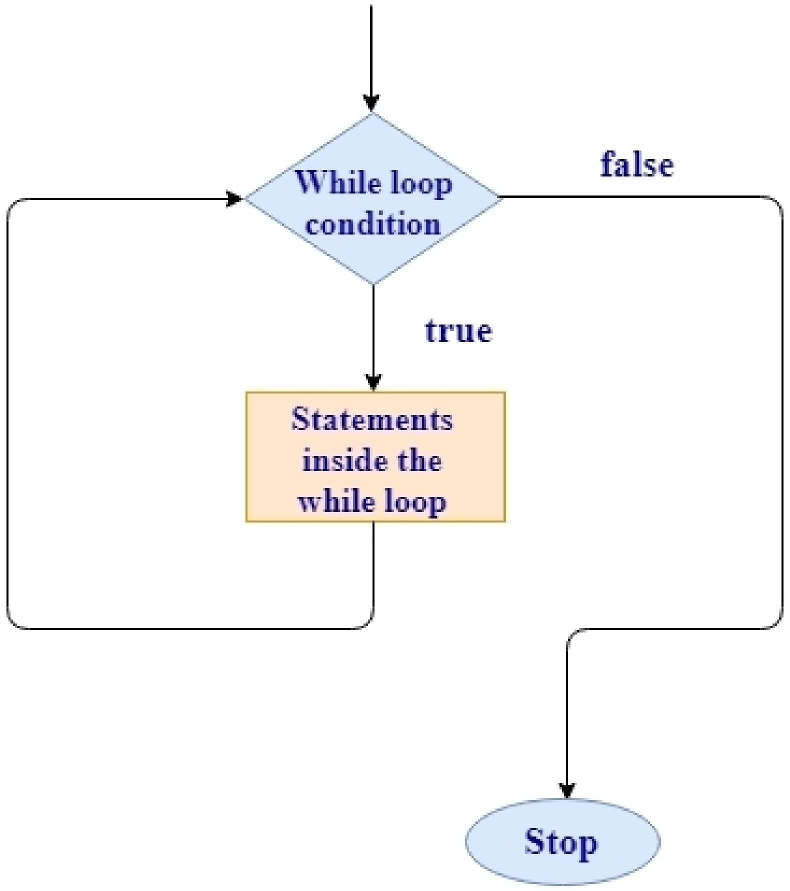
The 'while-loop' is a conditional loop. When a program needs to evaluate a condition then the conditional loop is used. When the condition is true then it executes the loop otherwise it terminates the loop.

Syntax of 'while loop'

```
while condition

//block statements;
```

- In the above syntax, while loop evaluates the condition. If the condition is true, block statements are executed otherwise it terminates the loop. Rust provides this inbuilt construct which can be used in combinations with 'loop', 'if', 'else' or 'break' statement.

Flow diagram of while loop



Let's see a simple example

```
fn main()
{
 let mut i=1;
 while i<=10
{
  print!("{}", i);
  print!(" ");
  i=i+1;
}
}
```

Output:

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

In the above example, 'i' is a mutable variable means that the value of 'i' can be modified. The while loop executes till the value of 'i' is less than 10 or equal to 10.

Let's see a simple example

```
fn main()
{
 let array=[10,20,30,40,50,60];
 let mut i=0;
 while i<6
 {
   print!("{}",array[i]);
   print!(" ");
   i=i+1;
 }
}
```

Output:

```
10 20 30 40 50 60
```

In the above example, the elements of an array has been iterated using while loop.

Disadvantages of while loop:

- While loop can cause the problem if the index length is incorrect.
- It is also slow as the compiler adds the runtime code to perform the conditional check on every iteration through this loop.

For loop

The **for loop** is a conditional loop, i.e., the loop runs for the particular number of times. The behavior of for loop in rust language is slightly different from the other languages. The for loop is executed till the condition is true.

Syntax of for loop

```
for var in expression
{
    //block statements
}
```

In the above syntax, an expression can be converted into an iterator which iterates through the elements of a data structure. In every iteration, value is fetched from an iterator. When there are no values left to be fetched, loop is over.

Let?s see a simple example.

```
 fn main()
{

  for i in 1..11
  {
    print!("{} ",i);
  }
}
```

Output:

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

In the above example, 1..11 is an expression and the iterator will iterate over these values. The upper bound is exclusive, so loop will print from 1 to 10 values.

Let?s see a simple example.

```
 fn main()
{
let mut result;
for i in 1..11
{
result=2*i;
println!("2*{}={}",i,result);
}
}
```

Output:

```
2*1=2
2*2=4
2*3=6
2*4=8
2*5=10
2*6=12
2*7=14
2*8=16
2*9=18
2*10=20
```

In the above example, for loop prints the table of 2.

Let?s see another simple example.

```
fn main()


 let fruits=["mango","apple","banana","litchi","watermelon"];
  for a in fruits.iter()
 {
   print!("{} ",a);
 }
```

Output:

```
mango apple banana litchi watermelon
```

In the above example, iter() method is used to access the each element of fruits variable. Once, it reaches the last element of an array, then the loop is over.

Difference between the while loop and for loop:

If the index length of an array is increased at runtime, then the while loop shows the bug but this would not be happened in the case of for loop. Therefore, we can say that for loop increases the safety of the code and removes the chances of the bugs.

Rust Ownership

Understanding Ownership

Ownership is the unique feature that Rust programming language offers and provides the guarantee of memory safety without using garbage collector or pointers.

What is Ownership?

When a block of code owns a resource, it is known as **ownership**. The block of code creates an object that contains the resource. When the control reaches the end of the block, the object is destroyed, and the resource gets released.

Important points of Ownership:

- The "owner" can change the owning value of a variable according to mutability.
- Ownership can be transferred to another variable.
- Ownership is just moved semantics in Rust.
- Ownership model also guarantees the safeness in parallel.

Rules of Ownership

- In Rust, every value has a variable associated with it and that is called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value associated with it is destroyed.

Example of Ownership

Multiple variables can interact with each other in Rust. Let's look at an example:

Assigning the value of x to the variable y:

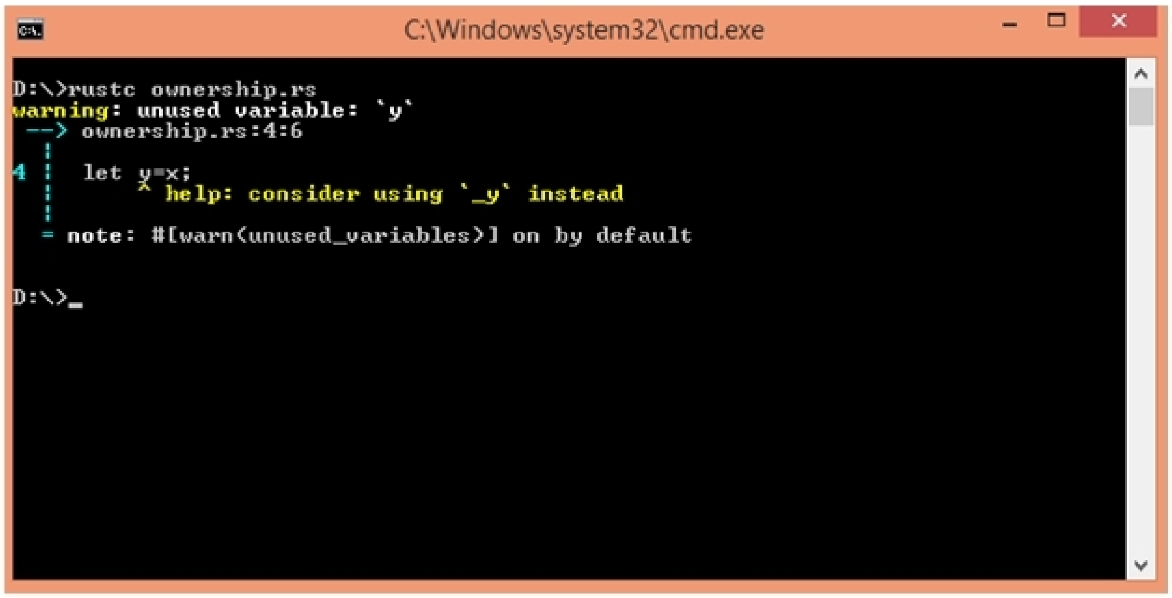
```
let x=10;
```

```
et y=x;
```

In the above example, x binds to the value 10. Then, the value of x is assigned to the variable y. In this case, the copy of x is not created rather than the value of x is moved to the variable y. Therefore, ownership of x is transferred to the variable y, and the variable x is destroyed. If we try to reuse the variable x, then Rust throws an error. Let's understand this through an example.

```
fn main()

let x=10;
let y=x;
println!("value of x :{}",x);
```

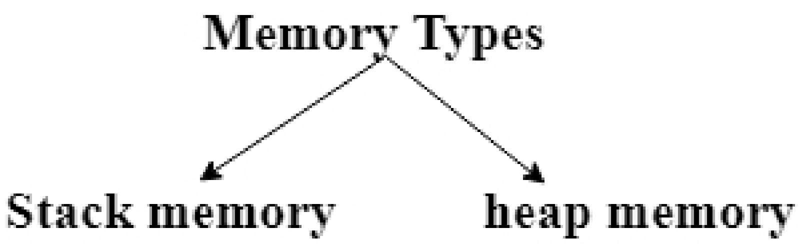
Following is the output of the above example:



```
D:\>rustc ownership.rs
warning: unused variable: `y`
 --> ownership.rs:4:6
  |
4 |   let y=x;
  |       ^ help: consider using `_y` instead
  |
  = note: #[warn(unused_variables)] on by default


D:\>_
```

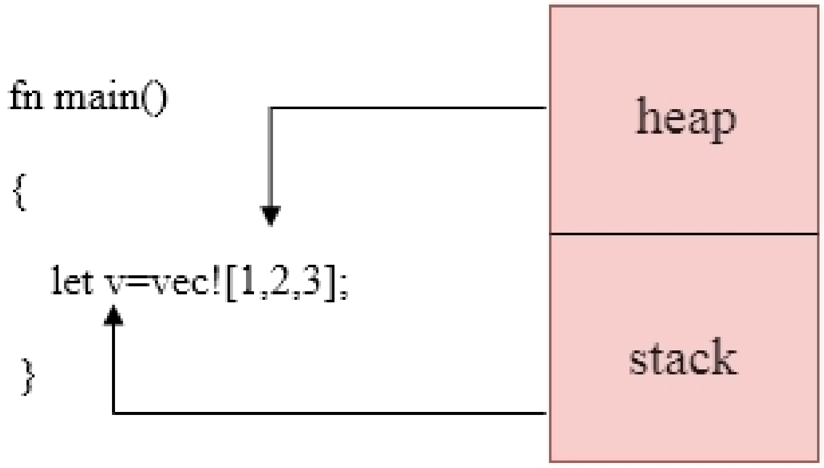
Memory and Allocation

In Rust, data can be stored either in stack or heap memory.



Stack memory: In stack memory, data is always placed in order and removed in the opposite order. It follows the principle "last in first out", i.e., the data which is inserted last is always removed first. Stack memory is an organized memory. It is faster than the heap memory because of the way it accesses the memory. If the size of the data is unknown at compile time, then heap memory is used for storing the content.

Heap memory: Heap memory is an organized memory. The operating system finds an empty space in the heap memory and returns a pointer. This process is known as "allocating on the heap".



This diagram shows that stack contains the pointer while heap contains the content.

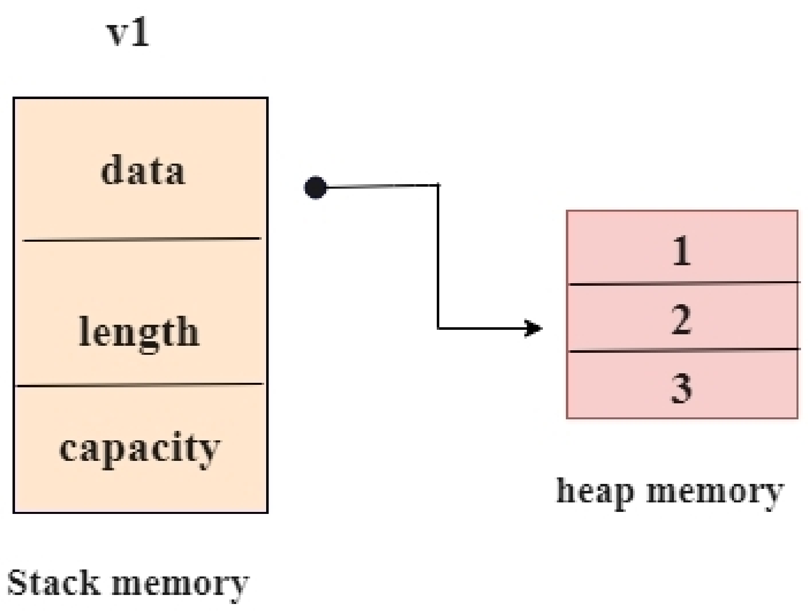
Let's see a simple example of memory allocation.

```
fn main()
{
 let v1=vec![1,2,3];
```

```
  let v2=v1;
}
```

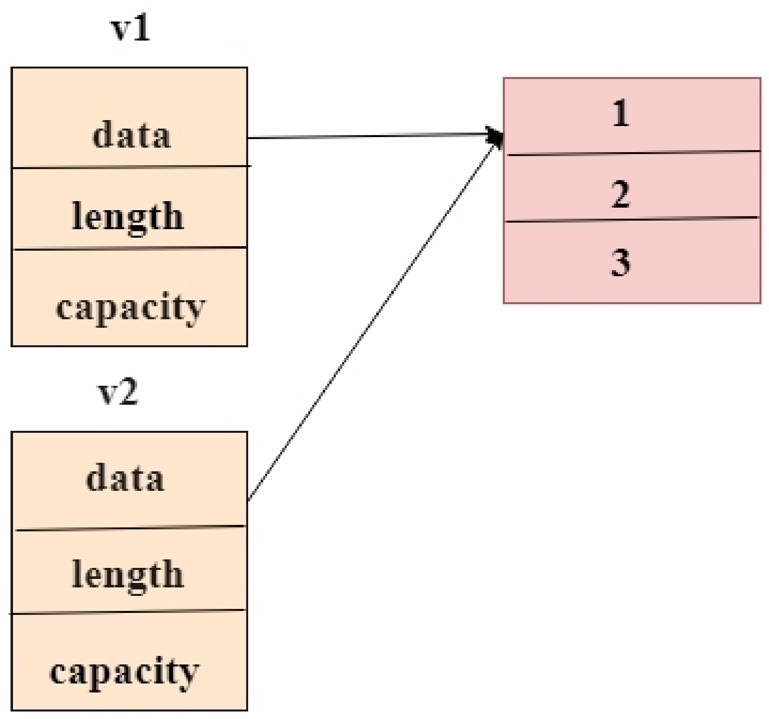
Step 1:

In the first statement of the program, vector v1 binds with the values 1,2 and 3. A vector is made up of three parts, i.e., a pointer to the memory pointing to the data stored in memory, length, and capacity of the vector. These parts are stored in the stack while data is stored in the heap memory as shown given below:



Step 2:

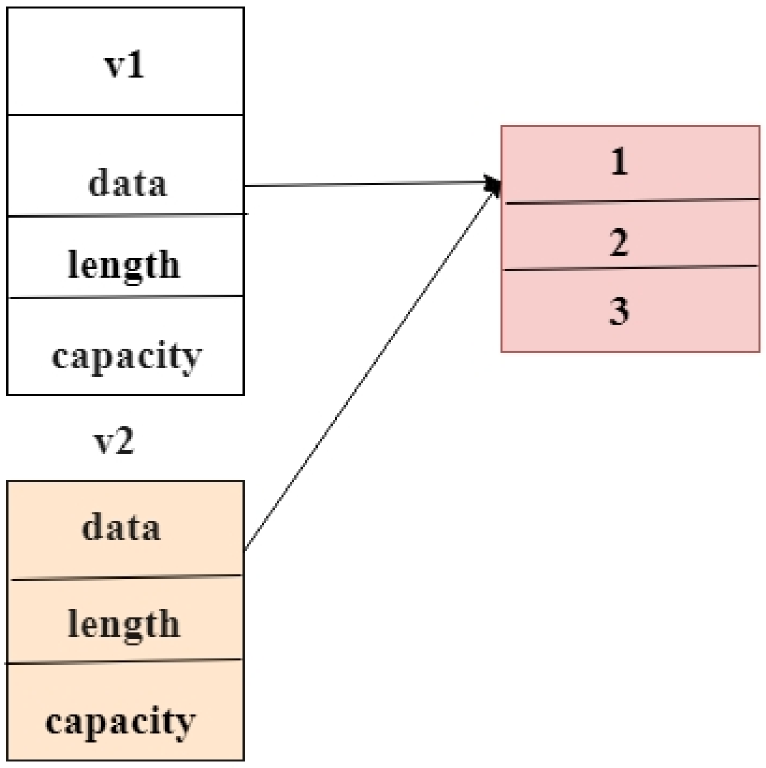
In the second statement of program, v1 vector is assigned to the vector v2. The pointer, length, and capacity are copied on the stack, but we do not copy the data in the heap memory. Let's look at the memory representation:



However, this type of representation can create a problem. When both v1 and v2 goes out of the scope, then both will try to free the memory. This causes the double free memory, and this leads to the memory corruption.

Step 3:

Rust avoids the step 2 condition to ensure memory safety. Instead of copying the allocated memory, Rust considers that v1 vector is no longer valid. Therefore, it does not need to free the memory of v1 as v1 goes out of the scope.


v1
data
length
capacity
v2
data
length
capacity
1
2
3

Use of Copy trait

Copy trait is a special annotation which is placed on the types like integers that are stored on the stack. If copy trait is used on the types, then the older variable can be further used even after the assignment operation.

Here are some of the types that are copy:

- All the integer types such as u32.
- The Boolean type, bool with the value true or false.
- All the floating types such as f64.
- The character type, char.

Ownership and functions

When a variable is passed to the function, then the ownership moves to the variable of a called function. The semantics of passing value is equal to the assigning a value to a variable.

Let's understand this through an example:

```
 fn main()
{
 let s=String::from("Rustlang");
 take_ownership(s);
 let ch='a';
 moves_copy(ch);
 println!("{}",ch);
}
fn take_ownership(str:String)
{
println!("{}",str);
}
fn moves_copy(c:char)
{
 println!("{}",c);
}
```

Output:

```
Rustlang
a
a
```

In the above example, string 's' binds with the value "Rustlang" and the ownership of 's' variable is passed to the variable 'str' through a take_ownership() function. The 'ch' variable binds with a value 'a,' and the ownership of 'ch' variable is passed to the variable 'c' through a moves_copy() function. The 'ch' variable can also be used afterwards as the type of this variable is a "copy" trait.

Returning value and scope

Returning values from the function also transfer the ownership. **Let's look at this:**

```
 fn main()
{
 let x= gives_ownership();
 println!("value of x is {}",x);
 }
fn gives_ownership()->u32
{
    let y=100;
    y
}
```

Output:

```
value of x is 100
```

In the above example, gives_ownership() function returns the value of y, i.e., 100 and the ownership of y variable is transferred to the x variable.

Rust References and Borrowing

Reference is an address which is passed to a function as an argument. **Borrowing** is just like when we borrow something, and we are done with it, we give it back. References and Borrowing are mutual to each other, i.e. when a reference is released then the borrowing also ends.

Why Borrowing?

Borrowing concept is used because of the following reasons:

- Borrowing allows to have multiple references to a single resource but still obeys to have a "single owner".
- References are just like pointers in C.
- A reference is an object. References are of two types, i.e., mutable references and immutable references. Mutable references are moved while immutable references are copied.

Let's understand this through an example.

```
 fn main()
{
 let str=String::from("Rustlang");
 let len=calculate_length(&str);
 println!("length of the string {}",len);
}
fn calculate_length(s:&String)->usize
{
  s.len()
}
```

Output:

```
length of the string 10
```

In the above example, calculate_length() function has a reference to string str as a parameter without taking its ownership.

```
let str=String::from("Rustlang");
et len=calculate_length(&str);
```

In the above scenario, &str is a reference to variable str, but it does not own it. Therefore, the value pointed by the reference will not be dropped even when the reference goes out of scope.

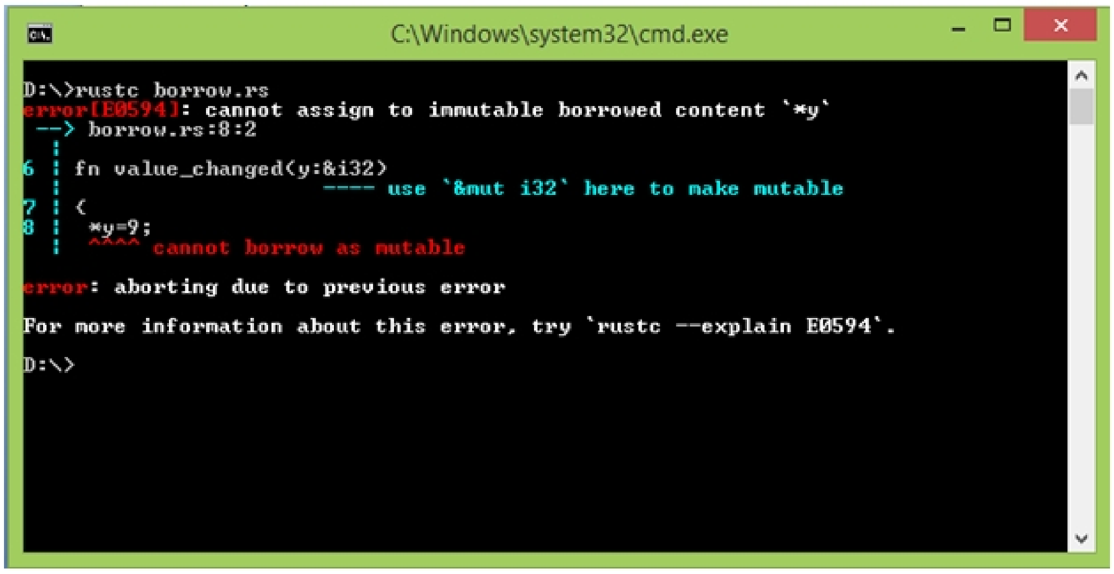
```
fn calculate_length(s:&String)->usize

 s.len()
```

In the above case, variable 's' is valid until the control does not go back to the main() function. When the variables are passed as a reference to the function instead of actual values, then we don't need to return the values to give back the ownership.

Let's try to modify the borrowed value.

```
 fn main()
{
 let x=1;
 value_changed(&x)
}
fn value_changed(y:&i32)
{
 *y=9;
}
```

Following is the output of the above program:



```
D:\>rustc borrow.rs
error[E0594]: cannot assign to immutable borrowed content `*y`
 --> borrow.rs:8:2
  |
6 | fn value_changed(y:&i32)
  |                    ---- use `&mut i32` here to make mutable
7 | {
8 |  *y=9;
  |  ^^^^ cannot borrow as mutable

error: aborting due to previous error

For more information about this error, try `rustc --explain E0594`.

D:\>
```

In the above example, it throws an error as &x is an immutable reference. Therefore, we cannot change the value of y.

Mutable Reference

We can fix the above error by using a mutable reference. Mutable references are those references which are changeable. Let's understand this through an example.

```
fn main()
{
let mut x=1;
value_changed(&mut x);
println!("After modifying, the value of x is {}",x);
}
fn value_changed(y:&mut i32)
{
*y=9;
}
```

Output:

```
After modifying, the value of x is 9
```

In the above example, we create a mutable reference, i.e., &mut x and the reference is pointed by the variable y which is of &i32 type. Now, we can change the value which is referenced by 'y' variable. We assign 9 value i.e*y=9. Therefore, the value x also becomes 9 as the same memory location referenced by both the variables.

Restrictions of Mutable references

- We can have only one mutable reference to a piece of data in a particular scope.

For example:

```
let mut str=String::from("Rustlang");
let a= &mut str;
let b= &mut str;
```

In the above scenario, the compiler throws an error as it consists of two mutable references which are not possible in Rust language.

- If the immutable reference exists in our program, then we cannot have a mutable reference in our program.

For example:

```
let mut str=String::from("Rustlang");
let a= &str;
let b=&str;
let c=&mut str;
```

In the above scenario, the compiler throws an error because we cannot have a mutable reference while we have an immutable reference.

Slice In Rust

Slice is a data type that does not have ownership. Slice references a contiguous memory allocation rather than the whole collection. It allows safe, efficient access to an array without copying. Slice is not created directly but from an existing variable. A slice consists of the length, and it can be mutable or not. Slices behave like arrays only.

String slice

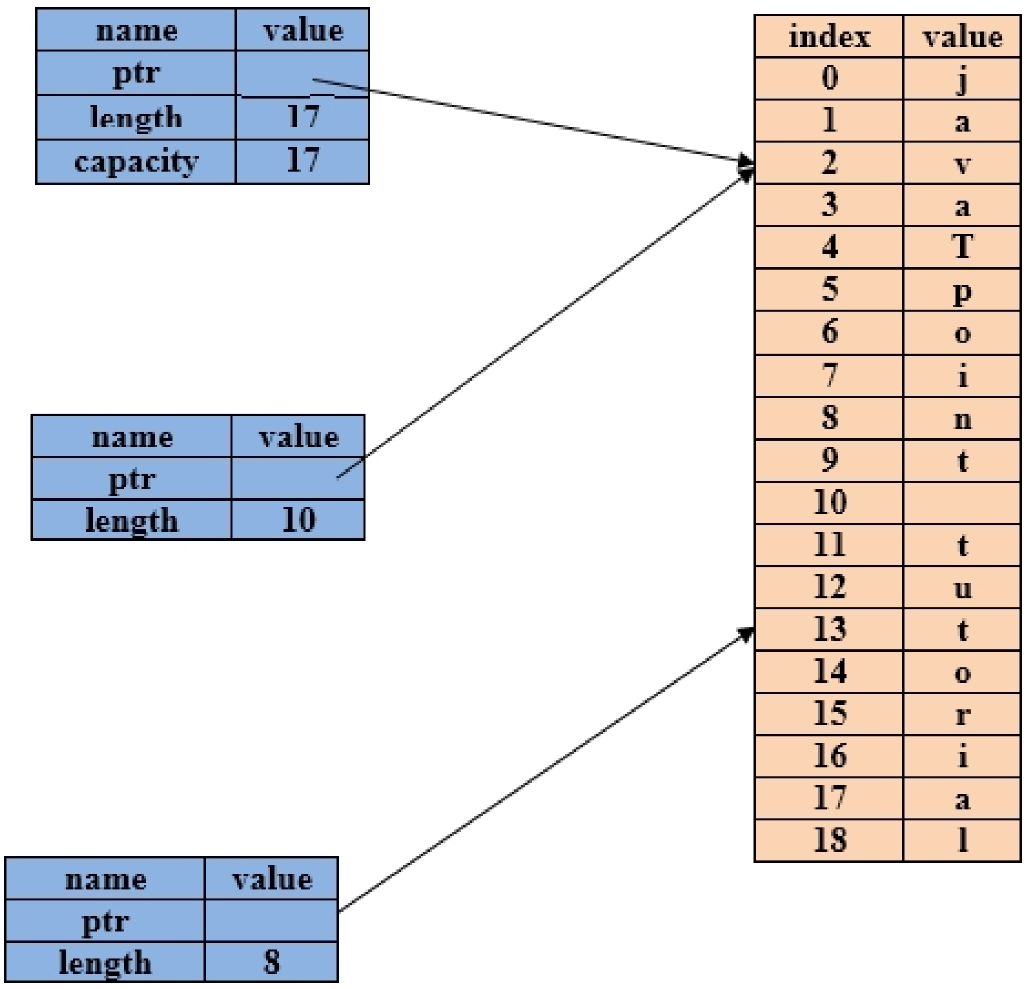
A string slice refers to a part of the string. Slice looks like:

```
 let str=String::from("Rustlang tutorial");
let Rustlang=&str[0..10];
let tutorial=&str[11,18];
```

Rather than taking an entire string, we want to take a part of the string. The [start..end] syntax is a range that begins at the start but does not include end. Therefore, we can create a slice by specifying the range within brackets such as [start..end] where 'start' specifies the starting position of an element and 'end' is one more than the last position in the slice. If we want to include the end of the string, then we have to use '..=' instead of '..'.

```
 let str= String::from("Rustlang tutorial");
let Rustlang = &str[0..=9];
let tutorial= &str[11..=18] ;
```

Diagrammatically representation:



name	value
ptr	
length	17
capacity	17

name	value
ptr	
length	10

name	value
ptr	
length	8

index	value
0	j
1	a
2	v
3	a
4	T
5	p
6	o
7	i
8	n
9	t
10	
11	t
12	u
13	t
14	o
15	r
16	i
17	a
18	l

- If you want to start the index from 0, then we can drop the starting index also. It looks like:

```
 let str= String::from("hello world");
let hello = &str[0..5];
let hello=&str[..5];
```

- If slice includes the last byte of the string, then we can drop the starting index. It looks like:

```
 let str= String::from("hello world") ;
let hello=&str[6..len];
let world = &str[6..];
```

Let's see a simple example of string slice:

```
fn main()

let str=String::from("Rustlang tutorial");
let Rustlang=&str[..=9];
println!("first word of the given string is {}",Rustlang);
```

Output:

```
first word of the given string is Rustlang
```

String slices are literals

String literals are stored in binary and string literals are considered as string slices only. Let's look:

```
let str = "Hello Rustlang" ;
```

The type of 'str' is '&str'. It is a slice pointing to a specific point of the binary. String literals are immutable, and '&str' is an immutable reference.

String slices as parameters

If we have a string slice, then we can pass it directly. Instead of passing the reference, we pass the string slice as a parameter to the function to make an API more general and useful without losing its functionality.

```
 fn main()
{
 let str= String:: from("Computer Science");
let first_word=  first_word(&str[..]); //first_word function finds the first word of the string.
let s="Computer Science" ; //string literal
let first_word=first_word(&s[..]); // first_word function finds the first word of the string.
let first_word=first_word(s) ; //string slice is same as string literal. Therefore, it can also be
                                            written in this way also.

}
```

Other slices

An array can also be treated as slices. They behave similarly as a string slice. The slice has the type [&i32]. They work similarly as a string slice by storing a reference as a first element and length as a second element.

Consider a array:

```
let arr = [100,200,300,400,500];  // array initialization
et a = &arr[1..=3]; // retrieving second,third and fourth element
```

Let's see a simple example.

```
fn main()

let arr = [100,200,300,400,500,600];
let mut i=0;
let a=&arr[1..=3];
let len=a.len();
println!("Elements of 'a' array:");
while i<len
```

```
{
println!("{}",a[i]);
i=i+1;
}
}
```

Output:

```
Elements of 'a' array:
200
300
400
```

What is a structure?

A structure is a user-defined data type that consists of variables of different data types. A structure is defined by using the struct keyword before the structure name. Structure members are enclosed within the curly brackets. Inside the curly brackets, structure members are defined with their name and type and structure members are also known as **fields**.

The Syntax of structure:

```
struct Student
{
member-variable1;
member-variable2;
.
.
}
```

In the above syntax, structure is defined by using the keyword struct. A structure contains the variables of dissimilar types.

How to declare the instance of a structure

```
let user = Student{
// key:value pairs;
}
```

In the above declaration, a user is an instance of Student structure. It is defined by using the structure name and then curly brackets. The curly brackets contain key:value pairs where keys are the name of the fields and value is the data which we want to store in the key field.

Let's create a structure of employee:

```
struct Employee{
employee_name : String,
employee_id: u64,
employee_profile: String,
active: bool,
}
```

An Instance of employee structure:

```
let employee = Employee{
employee_name : String::from("Akshay Gupta"),
employee_id: 12,
employee_profile : String::from("Computer Engineer"),
active : true,
};
```

How to access a specific member variable of Structure?

We can access the specific member variable of a structure by using dot notation. Suppose we want to access the employee_name variable of an Employee structure, then it looks like:

```
employee.employee_name;
 let mut employee = Employee{
employee_name : String::from("William"),
employee_id: 12,
employee_profile : String::from("Computer Engineer"),
active : true,
};
employee.employee_name = String :: from("John");
```

Creating an instance within the function body:

```
fn create_employee(name:String, profile:String)
{
Employee{
employee_name:name,
employee_id:12,
employee_profile:profile,
active:true,
}
}
```

In the above example, an instance of Employee structure is created implicitly within the function body. The create_employee() function returns the instance of Employee structure with the given name and profile.

Using the Field Init Shorthand when parameters passed to the function and fields have the same name.

Rust provides the flexibility of using field init shorthand when both the variables and fields have the same name. There is no need of repetition of fields and variables.

```
fn create_employee(employee_name:String, employee_profile:String)
{
Employee{
employee_name,
employee_id:12,
employee_profile,
active:true,
}
}
```

In the above example, the name of parameters and fields are the same. Therefore, there is no need of writing **employee_name:employee_name**, it can be directly written as **employee_name**.

Update Syntax

Creating a new instance from other instances using Struct update syntax.

When a new instance uses most of the values of an old instance, then we can use the struct update syntax. Consider two employees employee1 and employee2.

- First, create the instance employee1 of Employee structure:

```
let employee1 = Employee{
employee_name : String::from("William"),
employee_id: 12,
employee_profile : String::from("Computer Engineer"),
active : true,
};
```

- Second, create the instance the employee2. Some values of the employee2 instance are the same as employee1. There are two ways of declaring the employee2 instance.

The first way is declaring the employee2 instance without syntax update.

```
let employee2 = Employee{
employee_name : String::from("John"),
employee_id: 11,
employee_profile : employee1.employee_profile,
active : employee1.active,
};
```

The second way is declaring the employee2 instance by using syntax update.

```
let employee2 = Employee{
employee_name : String::from("John"),
employee_id: 11,
..employee1
};
```

The syntax '..' specifies that the rest of the fields are not explicitly set and they have the same value as the fields in the given instance.

Let's see a simple example of Structure:

```
struct Triangle
```

```rust
{
base:f64,
height:f64,
}

fn main()
{
let triangle= Triangle{base:20.0,height:30.0};
print!("Area of a right angled triangle is {}", area(&triangle));
}

fn area(t:&Triangle)->f64
{
0.5 * t.base * t.height
}
```

Output:

```
Area of a right angled triangle is 300
```

In the above example, the structure of a triangle is created, and it contains two variables, i.e., base and height of a right-angled triangle. The instance of a Triangle is created inside the main() method.

Method Syntax

Methods are similar to functions as they contain the **fn** keyword at the starting and then function name. Methods also contain the parameters and return value. However, when the method is declared within the struct context, then the method syntax varies from the normal function. The first parameter of such methods is always **self**, which represents the instance on which the function is called upon.

Defining methods

Let's define the method when the method is declared in the struct context.

```
struct Square
{
a : u32,
}
impl Square
{
fn area(&self)->u32
{
self.a * self.a
}
}

fn main()
{
let square = Square{a:10};
print!("Area of square is {}", square.area());
}
```

Output:

```
Area of square is 100
```

When the method is declared within the struct context, then we defin the method inside the implementation block, i.e., impl block.

```
impl Square
{
fn area(&self)->u32
{
self.a * self.a
}
}
```

The first parameter is to be self in the signature and everywhere within the body.

Here, we use the method syntax to call the area() function. The method syntax is an instance followed by the dot operator, method name, parameter, and any arguments.

```
1. square.area();
```

Where the square is an instance and area() is the function name.

An Advantage of method syntax:

The main advantage of using method syntax over functions is that all the data related to the instance is placed inside the impl block rather than putting in different places that we provide.

Rust Enum

Enum is a custom data type which contains some definite values. It is defined with an enum keyword before the name of the enumeration. It also consists of methods.

The syntax of enum:

```
 enum enum_name
{
  variant1,
 variant2,
.
.
}
```

In the above syntax, enum_name is the name of the enum and variant1,variant2,.. are the enum values related to the enum name.

For example:

```
enum Computer_language

  C,
  C++,
  Java,

```

In the above example, computer_language is the enum name and C, C++, Java are the values of computer_language.

Enum values

Let's create the instance of each of the variants. It looks like:

```
let c = Computer_language :: C;
let cplus = Computer_language :: C++;
let java = Computer_language :: Java;
```

In the above scenario, we create the three instances, i.e., c, cplus, java containing the values C, C++, Java respectively. Each variant of enum has been namespaced under its identifier, and double colon is used. This is useful because Computer_language::C, Computer_language::C++, Computer_language::Java belongs to the same type, i.e., Computer_language.

- We can also define a function on a particular instance. Let's define the function that takes the instance of type Computer_language; then it looks like:

```
fn language_type(language_name::Computer_language);
```

This function can be called by either of any variant:

```
language_type(Computer_language :: C++);
```

Let's understand through an example.

```
#[derive(Debug)]
enum Employee {
   Name(String),
   Id(i32),
   Profile(String),
}
fn main() {

   let n = Employee::Name("Hema".to_string());
   let i = Employee::Id(2);
   let p = Employee::Profile("Computer Engineer".to_string());
   println!(" {:?} s {:?} b {:?}", n,i,p);
}
```

Output:

```
Name("Hema") s Id(2) b Profile("Computer Engineer")
```

In the above example, Employee is a custom data type which contains three variants such as Name(String), Id(i32), Profile(String). The ":?" is used to print the instance of each variant.

Match Operator

The **match operator** allows us to compare a value against a series of patterns, and executes the code whenever the match is found. The patterns can be literal values, variable names, wildcards and many other things.

Let's understand the match operator through a simple example:

```
enum Computerlanguage
{
 C,
 Cplus,
 Java,
 Csharp,
}
fn language(language:Computerlanguage)
{
 match language
 {
  Computerlanguage::C=> println!("C language"),
  Computerlanguage::Cplus=> println!("C++ language"),
  Computerlanguage::Java=> println!("Java language"),
  Computerlanguage::Csharp=> println!("C# language"),
 }
}
fn main()
{
 language(Computerlanguage::C);
 language(Computerlanguage::Cplus);
 language(Computerlanguage::Java);
 language(Computerlanguage::Csharp);
}
```

Output:

```
C language
C++ language
Java language
C# language
```

In the above example, Computerlanguage is a custom data type which consists of four variants are C, Cplus, Java, Csharp. The match operator matches the value of the language with the expressions given in the match operator block.

Matching with Option<T>

Option<T> is used when we want to get the inner value of T out of **some** case.

The Option<T> consists of two variants:

- **None:** It indicates the failure or lack of value.
- **Some(value):** It is a tuple struct that wraps the value with T.

Let's understand through an example:

```
 fn main()
{
 even_number(2);
 even_number(3);
}
fn even_number(n:i32)
{
 let num=n;
  match checked_even(n)
  {
    None=>println!("None"),

    Some(n)=>
    {
    if n==0
    {
    println!("{} is a even number",num);
    }
    else
    {
    println!("{} is a odd number",num);
    }},
  }
}
```

```
fn checked_even(number:i32)->Option<i32>
{

  Some(number%2)

}
```

Output:

```
2 is a even number
3 is a odd number
```

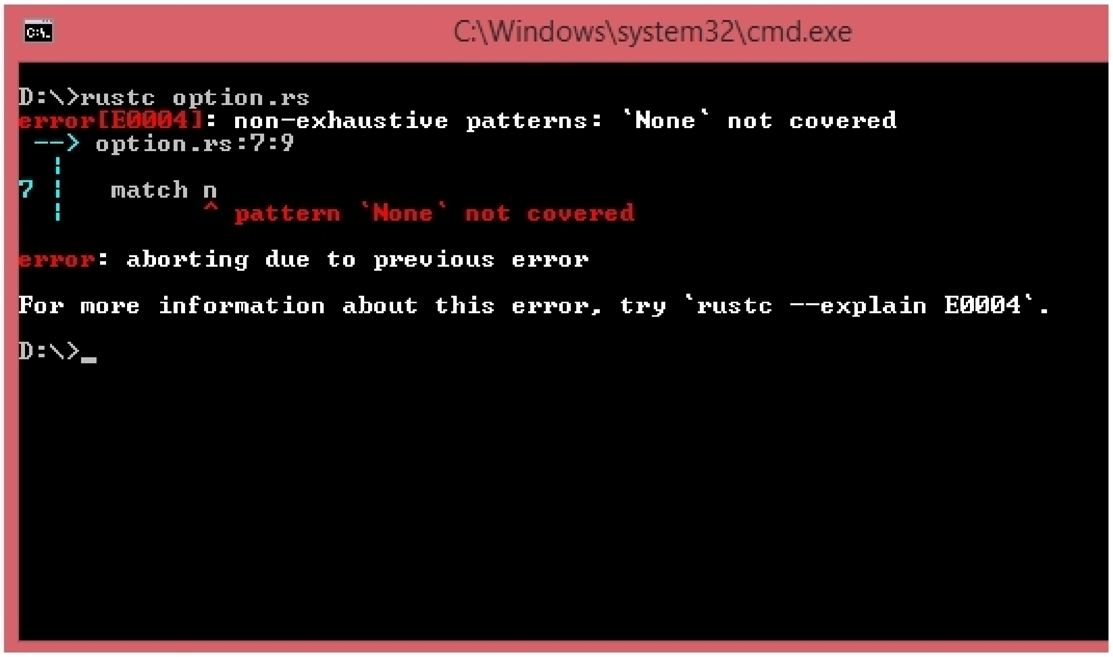
Matches are exhaustive

In Rust, matches are exhaustive, i.e., we should exhaust every possible case for the code to be valid. Suppose we forget to write the None case then the Rust compiler will show the bug that "pattern 'None' not covered".

Let's understand this case through an example:

```
 fn main()
{
 Some(5);
}
fn Value(n:Option<i32>)
{
  match n
  {
    Some(n)=>println!("{}is a Number",n),
  }
}
```

Output:



```
D:\>rustc option.rs
error[E0004]: non-exhaustive patterns: `None` not covered
 --> option.rs:7:9
  |
7 |     match n
  |           ^ pattern `None` not covered

error: aborting due to previous error

For more information about this error, try `rustc --explain E0004`.

D:\>_
```

Concise control flow with if let

The **if let** syntax is used to combine if and let which handles the values that matches one of the patterns while ignoring the rest of the code. The working of "match" operator and "if let" expression is similar.

Example of match operator

```
 fn main()
{
let a = Some(5);
match a {
    Some(5) => println!("five"),
    _ => (),
}}
```

Output:

```
five
```

In the above example, the match operator executes the code when the value is equal to Some(5). The **"_=>()"** expression satisfies the match expression after executing the first variant. If we use **if let** instead of **match**, then it reduces the length of the code.

Example of if let

```
 fn main()
{
let a=Some(3);
if let Some(3)=a{
 println!("three");
  }
  }
```

Output:

```
three
```

Rust Modules

A module is a namespace which contains the definitions of the functions or its types. A module is a collection of items such as functions, structs, traits, impl blocks. By default, the modifier of the module is private, but it can be overridden with the public modifier by using pub keyword.

Following are the keywords used in the modules:

- **mod keyword:** The "mod" keyword declares the new module.
- **pub keyword:** By default, all the functions, types, modules and constants have a private visibility modifier. The pub keyword makes the visibility modifier as public, and therefore, they are accessible outside the namespace.
- **use keyword:** The use keyword is used to import the module into local scope.

Module Definition

The module is defined by the mod keyword.

The syntax of Module:

```
mod module_name

  // body inside the module.

```

A Module can be categorized in three ways:

1. Single module: When the module appeared in a single file is known as a single module.

Let's understand this through an example:

```
 mod a
{
 pub fn single_module()
{
  println!("Single module");
}
}
fn main()
{
  a::single_module();
}
```

Output:

```
Single module
```

In the above example, module 'a' is defined, and every code defined in the block is inside the namespace 'a'. The function of module 'a' can be called by using the module name followed by namespace and then function name.

- We can also do the above example by using a separate file:

```
 mod module;
fn main()
{
  module::single_module();
}
```

```
pub fn single_module()
{
   println!("Single module");
}
```

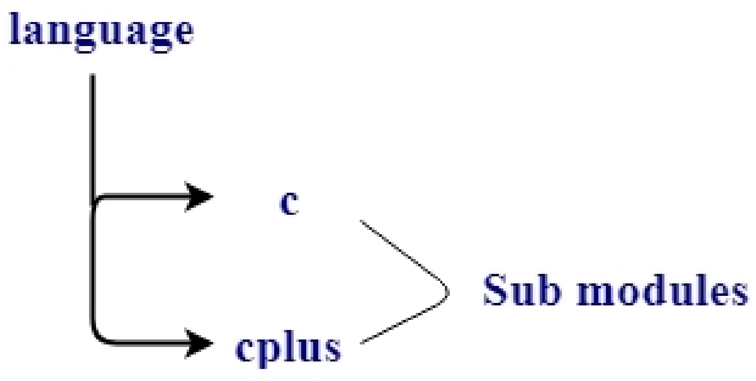
Output:

```
Single module
```

In the above two examples, we examine that ***mod X is defined either in curly braces*** or in a ***separate file named as X.rs or X/mod.rs***.

2. Sub-modules: In a single file, we can have multiple modules. Suppose the library name is "language" and it consists of two modules, i.e., C and Cplus.

Hierarchy of a "language" library is given below:



Let's understand through an example:

```
 mod c
{
 pub fn c()
  {
   println!("C is a structured programming language");
  }
}
mod cplus
{
 pub fn cplus()
 {
   println!("C++ is an object-oriented programming language");
 }
}
fn main()
{
 c::c();
 cplus::cplus();
}
```

Output:

```
C is a structured programming language
C++ is an object-oriented programming language
```

In the above example, the program consists of two modules, i.e., c and cplus and their respective functions are called by using c::c() and cplus::cplus().

3. Nested modules: Nested modules are those modules which consist of a module inside of modules, and they can be useful when the related modules are grouped together.

Let's understand this through an example:

```
mod a
{
pub fn a()
{
 println!("a module");
}
pub mod b
{
 pub fn a()
 {
  println!("b module");
 }
}
}
fn main()
{
a::a();
a::b::b();
}
```

Output:

```
a module
b module
```

In the above example, the program consists of two modules, i.e., 'a' and 'b' where 'b' is the inner module of 'a'. Both the modules consist the function with the same name but with different functionality. Both the functions are called by using a::a() and a::b::b() respectively. They both will not conflict with each other as they belong to different namespaces.

Filesystem

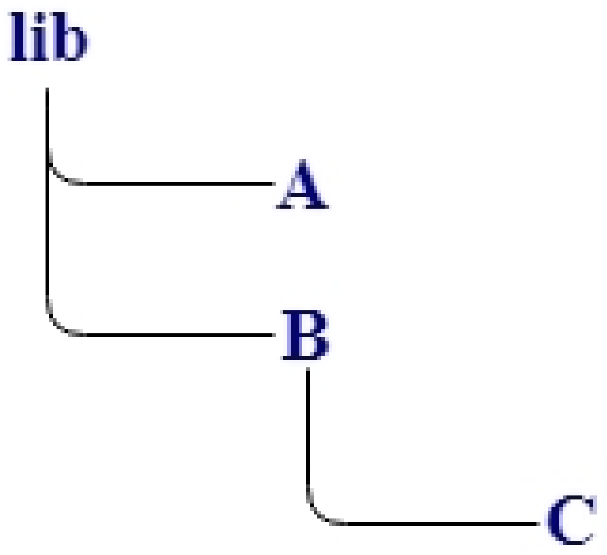
A module forms a hierarchical structure so that the project becomes more understandable. Rust module system is used to split the multiple files in such a way that not everything lies in the src/lib.rs or src/main.rs file.

Filename: src/lib.rs

```
 mod A
{
    fn a()
    {
        // block of statements.
    }
}
mod B
{
    fn b()
    {
        // block of statements.
    }
   mod C
  {
      fn c()
      {
         // block of statements.
      }
  }
}
```

In the above example, a program consists of three modules, i.e., A,B and C. C is an inner module of a B module.

Module hierarchy of a given file is:



If the module contains many functions and the functions are very lengthy, then it becomes difficult to find the code of a particular function. Rust provides the flexibility by providing the module system. We can have a separate file of each module rather than placing in the same file, i.e., src/lib.rs.

Steps to be followed:

Firstly, replace the block of a module 'A' with a semicolon.

```
 mod A;
mod B
{
    fn b()
    {
          // block of statements.
    }
  mod C
  {
      fn c()
      {
            // block of statements.
      }
  }
}
```

The semicolon ; tells the Rust to find the definition of a module 'A' into another location where the scope of module 'A' is defined.

- **mod A; looks like:**

```
 mod A
{
    fn a()
    {
        // block of statements.
    }
}
```

Now create the external file which contains the definition of module A. The name of the external file would be named as src/A.rs. After creating the file, write the definition of module A in this file which has been removed previously.

Filename: src/A.rs.

```
fn a()

```

```
    // block of statements.
```

In this case, we do not need to write the mod declaration as we mentioned in the src/lib.rs file. And, if we write the mod declaration here, then it becomes a submodule of module A.

Rust bydefault looks into the src/lib.rs file then this file determines which file is to be looked further.

Now, we will extract the module B from the file src/lib.rs and replace the body of module B with the semicolon.

Filename: src/lib.rs

```
mod A;
od B;
```

- **mod B; looks like:**

```
mod B

    fn b()
    {
          // block of statements.
    }
  mod C
 {
     fn c()
     {
           // block of statements.
     }
 }

```

Now create the external file which contains the definition of module B. The name of the external file would be named as src/B.rs. After creating the file, write the definition of module B in this file which have been removed previously.

Filename: src/B.rs

```
fn b()
    {
          // block of statements.
    }
```

```
 mod C
{
    fn c()
    {
        // block of statements.
    }
}
```

Now we will extract the module C from the file src/B.rs and replace the body of the module C with the semicolon.

```
fn b()
   {
        // block of statements.
   }
mod C;
```

- **mod C; looks like:**

```
 mod C
{
   fn c()
   {
       // block of statements.
   }
}
```

Now create the external file which contains the definition of module C. The name of the external file would be named as src/C.rs. After creating the file, write the definition of module C in this file which has been removed previously.

File name: src/C.rs

```
fn c()
    {
        // block of statements.
    }
```

Rules of Module filesystem:

- If the module named "server" and has no submodules, then all the declarations of the module can be placed in the file server.rs.
- If the module named "server" contains the submodules, then all the declarations of the module are to be placed in the file server/mod.rs.

Making a functioning public

The "pub" keyword is used at the starting of the declaration so that the function becomes accessible to the outside functions.

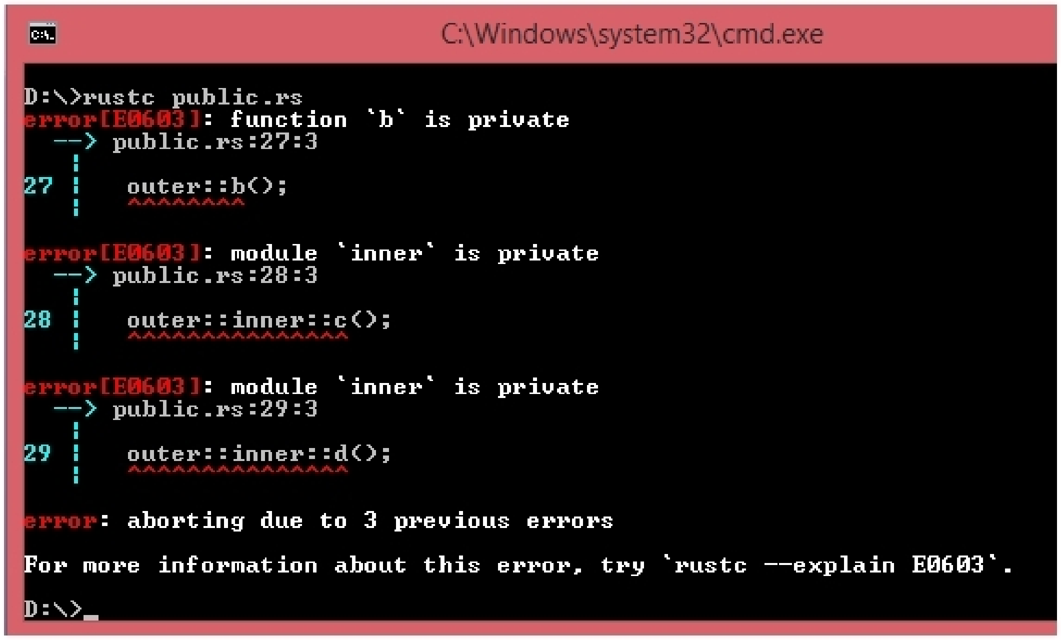
Following are the privacy rules:

- If any function or module is public, then it can be accessed by any of the parent modules.
- If any function or module is private, then it can be accessed either by its immediate parent module or by the parent's child module.

Let's understand this through a simple example:

```
 mod outer
{
  pub fn a()
  {
   println!("function a");
  }
  fn b()
  {
    println!("function b");
  }

mod inner
{
 pub fn c()
 {
  println!("function c");
 }
 fn d()
 {
  println!("function d");
 }
}
}
fn main()
{
 outer::a();
27.   outer::b();
28.   outer::inner::c();
29.   outer::inner::d();
30. }
```

Output:



```
C:\Windows\system32\cmd.exe

D:\>rustc public.rs
error[E0603]: function `b` is private
  --> public.rs:27:3
   |
27 |   outer::b();
   |   ^^^^^^^^
   |
error[E0603]: module `inner` is private
  --> public.rs:28:3
   |
28 |   outer::inner::c();
   |   ^^^^^^^^^^^^^^^
   |
error[E0603]: module `inner` is private
  --> public.rs:29:3
   |
29 |   outer::inner::d();
   |   ^^^^^^^^^^^^^^^
   |
error: aborting due to 3 previous errors

For more information about this error, try `rustc --explain E0603`.

D:\>_
```

In the above example, the main() function is the root module while an outer module is the current root module of our project. Therefore, the main() function can access the outer module.

The call to *outer::a()* will not cause any error as the function a() is public, but when the main() function tries to access the *outer::b()* function, then it causes the compilation error because it is a private function.

The main() function cannot access the inner module as it is private. An inner module has no child module, so it can be accessed only by its parent module, i.e., outer module.

Referring to names in different modules

When we call the function of a module, then we need to specify the full path.

Let's understand this concept through an example:

```
 pub mod a
{
 pub mod b
 {
   pub mod c
   {
     pub fn nested_modules()
     {
       println!("Nested Modules");
     }
   }
 }
}

fn main()
{
 a::b::c::nested_modules();
}
```

Output:

```
Nested Modules
```

In the above example, nested_modules() function is called by specifying the full path, i.e., **a::b::c::nested_modules()**.

use keyword

In the above scenario, we saw that the function calling is quite lengthy. Rust **"use keyword"** shortens the length of the function calling to bring the modules of a function in the scope. The use keyword brings only those modules which we have specified in the scope. Let's understand this through an example:

```
pub mod a
{
 pub mod b
 {
   pub mod c
   {
     pub fn nested_modules()
     {
       println!("Nested Modules");
     }
   }
 }
}

use a::b::c::nested_modules;
fn main()
{
 nested_modules();
}
```

Output:

```
Nested Modules
```

In the above example, the use keyword includes all the modules into the scope. Therefore, we can call the function directly without including the modules in the calling function.

An enum is also a form of a namespace like modules. Therefore, we can use the use keyword to bring the enum variants into the scope. In **use** statement, we can list the enum variants in the curly brackets and the commas in the last position.

Let's understand through an example:

```
#[derive(Debug)]
enum Flagcolor
{
 Orange,
 White,
 Green,
}
use Flagcolor::{Orange,White,Green};
fn main()
{
  let _o= Orange;
  let _w= White;
 let _g= Green;
 println!("{:?}",_o);
println!("{:?}",_w);
println!("{:?}",_g);
}
```

Output:

```
orange
white
green
```

In the above example, Flagcolor is the namespace whose variants are specified in the use statement. Therefore, we can directly use the enum variants without using enum name and namespace specifier.

Use of '*' operator

The * operator is used to bring all the items into the scope, and this is also known as glob operator. If we use the glob operator, then we do not need to specify the enum variants individually.

Let's understand this through an example:

```rust
#[derive(Debug)]
enum Color
{
  Red,
  Yellow,
  Green,
  Orange,
}

use Color::*;
fn main()
{
  let _red=Red;
  let _yellow=Yellow;
  let _green=Green;
  let _orange=Orange;
  println!("{:?}",_red);
  println!("{:?}",_yellow);
  println!("{:?}",_green);
  println!("{:?}",_orange);
}
```

Output:

```
Red
Yellow
Green
Orange
```

In the above example, the '*' operator has been used to include all the enum variants without specifying the list in the use statement.

Use of super keyword

The **super keyword** is used to access the grandparent module from the current module. It enables us to access the private functions of the parent module.

```
 mod a{
fn x() -> u8 {
    5
}

pub mod example {
    use super::x;

    pub fn foo() {
        println!("{}",x());
    }
}}

fn main()
{
  a::example::foo();
}
```

Output:

```
5
```

In the above example, the module example has used the super which refers its parent module. Due to this reason, foo() function of module example can access the private function of module a.

Vector

A vector is a single data structure which enables you to store more than one value next to each other in the memory. A vector is useful when we have a list of items such as items in a shopping cart.

Important points:

- A vector is used to store the value of the same type.
- Vector is denoted by **Vec<T>**.
- The **Vec<T>** is provided by the standard library which can hold the data of any type, where **T** determines the type of the vector.
- The data of the vector is allocated on the heap.
- A vector is a growable array means that the new elements can be added at the runtime.

Vec<T>: When a vector holds the specific type then it is represented in the angular brackets.

How to create a vector?

A vector can be created by using Vec::new() function. Let's look at this:

```
Let v : Vec<i32> = Vec::new();
```

In the above declaration, v is a vector of i32 type and it is created by using Vec::new() function.

- **There is another way to create the vector:**

Rust provides vec! macro to create the vector and hold the values that we provide.

For example:

```
let v = vec![10,20,30,40,50];
```

In the above declaration, vector v is created using a vector macro, i.e., vec!. In the case of vec!, Rust automatically infer the type of the vector v is Vec<i32> as the vector macro contains the integer values.

```
let v = vec![2 ; i];
```

In the above declaration, vector 'v' is created using vector macro which contains the value 2 'i' times.

Accessing elements

The particular element of a vector can be accessed by using the subscript operator [].

Let's understand through an example:

```
fn main()
{
let v =vec![20,30,40,50];
println!("first element of a vector is :{}",v[0]);
println!("Second element of a vector is :{}",v[1]);
println!("Third element of a vector is :{}",v[2]);
println!("Fourth element of a vector is :{}",v[3]);
}
```

Output:

```
first element of a vector is :20
Second element of a vector is :30
Third element of a vector is :40
Fourth element of a vector is :50
```

- The second way of accessing the vector elements is to use the get(index) method with the index of a vector is passed as an argument and it returns the value of type Option<&t>.

Let's understand through an example:

```
            fn value(n:Option<&i32>)
           {
              match n
              {
                  Some(n)=>println!("Fourth element of a vector is {}",n),
                  None=>println!("None"),
               }
}
fn main()
{
let v =vec![20,30,40,50];
let a: Option<&i32>=v.get(3);
value(a);
}
```

Output:

```
Fourth element of a vector is 50
```

In the above example, get() method is used to access the fourth element of the vector.

Difference between [] &get() method:

When we access the nonexistent element using [] operator, then it causes the program to panic. Therefore, the program is crashed when we try to access the nonexistent element. If we try to access the element by using get() method, then it returns None without panicking.

Let's understand this through an example:

- **get(index)**

```rust
fn value(n:Option<&i32>)
{
 match n
 {
   Some(n)=>println!("Fourth element of a vector is {}",n),
   None=>println!("None"),
 }
}
fn main()
{
let v =vec![20,30,40,50];
let a: Option<&i32>=v.get(7);
value(a);
}
```

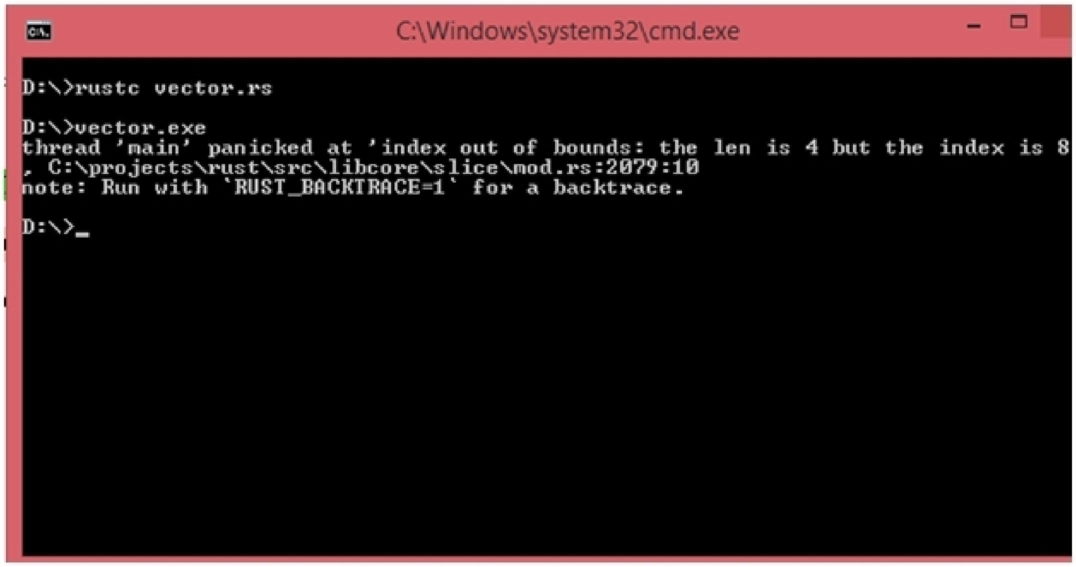
Output:

```
None
```

- **[ ] operator**

```rust
fn main()
{
let v =vec![20,30,40,50];
println!("{}",v[8]);
}
```

Output:



```
D:\>rustc vector.rs

D:\>vector.exe
thread 'main' panicked at 'index out of bounds: the len is 4 but the index is 8
, C:\projects\rust\src\libcore\slice\mod.rs:2079:10
note: Run with `RUST_BACKTRACE=1` for a backtrace.

D:\>_
```

Iterating over the values in a vector

If we want to access each element of a vector, then we can iterate over the elements of a vector rather than using the indexes to access a particular element of a vector.

We can use the 'for' loop to iterate over the mutable or immutable references.

Let's see a simple example of immutable references:

```
fn main()
{
let v =vec![20,30,40,50];
print!("Elements of vector are :");
for i in v
{
  print!("{} ",i);
}
            }
```

Output:

```
Elements of vector are :20 30 40 50
```

Let's see a simple example of mutable references:

```
fn main()
{
let mut v =vec![20,30,40,50];
print!("Elements of vector are :");
for i in &mut v
{
*i+=20;
  print!("{} ",i);
}
}
```

Output:

```
Elements of vector are :20 30 40 50
```

In the above example, we are changing the value of the vector. Therefore, the vector is a mutable reference. The dereference(*) operator is used before the 'i' variable to get the value of vector v.

Updating a vector

When we create the vector, then we insert the elements into the vector by using push() method. The push() inserts the new element at the end of the vector.

Let's see a simple example:

```
fn main()
{
 let mut v=Vec::new();
 v.push('j');
 v.push('a');
 v.push('v');
 v.push('a');
 for i in v
 {
  print!("{}",i);
 }
}
```

Output:

```
java
```

In the above example, push() function is used to insert the elements into the vector at the runtime. The vector 'v' is made mutable so that we can also change the value of a vector.

Dropping a vector

When a vector goes out of the scope, then it gets autonatically dropped or freed from the memory.

Let's understand this through a simple scenario:

```
fn main()
{
   let v = !vec[30,40,50];
   } => v is freed here as it goes out of the scope.
```

In the above scenario, a vector is freed when it goes out of the scope means that all the elements present in the vector will be removed.

Using Enum to store multiple types

Vectors can store the elements of the same type, and this is a big disadvantage of a vector. Enum is a custom data type which contains the variants of the various type under the same enum name. When we want to store the elements in a vector of a different type , then we use the enum type.

Let's understand this through an example:

```
#[derive(Debug)]
enum Values {
  A(i32),
  B(f64),
  C(String),
}

fn main()
{
    let v = vec![Values::A(5),
    Values::B(10.7),Values::C(String::from("Rustlang"))];
    for i in v
   {
     println!("{:?}",i);
    }
}
```

Output:

```
A(5)
B(10.7)
C(Rustlang)
```

Advantages of using enum in a vector:

- Rust knows the type of the elements of a vector at the compile time as to determine how much memory on the heap is required for each element.
- When a vector consists of elements of one or more type then the operations performed on the elements will cause the error but using

an **enum with the match** will ensure that every possible case can be handled at the runtime.

String

Rust contains the two types of strings: **&str** and **String**.

String:

- A string is encoded as a UTF-8 sequence.
- A string is allocated on the heap memory.
- A string is growable in size.
- It is not a null-terminated sequence.

&str

- '&str' is also known as a string slice.
- It is represented by &[u8] to point the UTP-8 sequence.
- '&str' is used to view the data present in the string.
- It is fixed in size, i.e., it cannot be resized.

Difference b/w 'String' and '&str'.

- A String is a mutable reference while &str is an immutable reference to the string, i.e., we can change the data of String, but the data of &str cannot be manipulated.
- A String contains the ownership on its data while &str does not have ownership, it borrows it from another variable.

Creating a new String

A String is created similarly as we create the vector. Let's look at this:

Creating an empty String:

```
Let mut s = String::new();
```

In the above declaration, String s is created by using new() function. Now, if we want to initialize the String at the time of declaration, we can achieve this by using the **to_string()** method.

- Implementing the to_string() method on the data:

```
let a = "Rustlang";
let s = a.to_string();
```

- We can also implement the to_string method directly on the string literal:

```
let s = "Rustlang".to_string();
```

Let's understand this through an example:

```
fn main()

 let data="Rustlang";
 let s=data.to_string();
 print!("{} ",s);
 let str="tutorial".to_string();
 print!("{}",str);
```

Output:

```
Rustlang tutorial
```

- The second way to create the String is to use String::from function, and this is equivalent to the String::new() function.

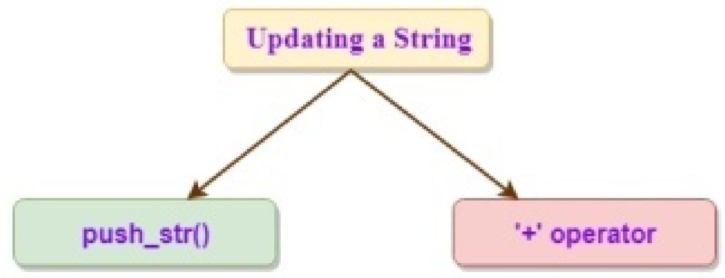
Let's understand this through a simple example:

```
fn main()
   {
       let str = String::from("Rustlang tutorial");
       print!("{}",str);
   }
```

Output:

Updating a String

We can change the size of the String and content of the String as well by pushing more data into the String. We can also use the '+' operator of the format macro! To concatenate the string values.



- **Appending to a string with push_str and push**

push_str() : We can grow the size of the String by using the push_str() function. It appends the content at the end of the string. Suppose s1 and s2 are two strings and we want to append the string s2 to the string s1.

```
s1.push_str(s2);
```

Let's understand this through a simple example:

```
fn main()
{
  let mut s=String::from("java is a");
  s.push_str(" programming language");
  print!("{}",s);
}
```

Output:

```
java is a programming language
```

The **push_str()** function does not take the ownership of the parameter. Let's understand this scenario through a simple example.

```
fn main()
{
  let mut s1 = String::from("Hello");
  let s2 = "World";
  s1.push_str(s2);
  print!("{}",s2);
}
```

Output:

```
World
```

If **push_str()** function takes the ownership of the parameter, then the last line of the program would not work, and the value of the s2 will not be printed.

push() : The push() function is used to add a single character at the end of the string. Suppose the string is s1 and character ch which is to be added at the end of the string s1.

```
s1.push(ch);
```

Let's see a simple example:

```
fn main()
{
 let mut s = String::from("java");
 s.push('c');
 print!("{}",s);
}
```

Output:

```
javac
```

- **Concatenation with the '+' operator or format macro**

'+' operator: The '+' operator is used to concatenate two strings. Let' look:

```
let s1 = String::from("Rustlang ");
let s2 = String::from("tutorial!!");
let s3 = s1+&s2;
```

Let's see a simple example:

```
 fn main()
{
let s1 = String::from("Rustlang");
let s2 = String::from(" tutorial!!");
let s3 = s1+&s2;
print!("{}",s3);
}
```

Output:

```
Rustlang tutorial!!
```

In the above example, s3 contains the result of the concatenation of two strings, i.e., Rustlang tutorial. The 's1' is no longer valid, and we use the reference of the s2, i.e., &s2 according to the signature of the method which is called when we use the '+' operator. The '+' operator calls the add() method whose declaration is given below:

```
fn add(self,s:&str)->String
{
 }
```

Firstly, s2 has **'&'** operator means that we are adding a reference to the s1. According to the signature of the add() function, we can add &str to a String, and we cannot add two string values together. But the type of s2 is &String not &str according to the second parameter specified in the add() method. But still, we able to use the s2 in the add method because the compiler coerces the &string into &str. Therefore, we can say that when we call the add() method, then Rust uses **deref coercion**.

Secondly, the first parameter of the add() function is self and add() takes the ownership of self. This means that s1 is no longer valid after the statement let **s3=s1+&s2;**

- **format! Macro**
  - When we want to concatenate multiple strings, then the use of '+' operator becomes very clumsy in this case. To concatenate the multiple strings, use of format macro is preferred.
  - The format macro works similarly as println! macro. The difference between format macro and println! macro is that format macro does not print on the screen, it returns the content of the string.

Let's understand this through a simple example:

```
fn main()
{
 let s1 = String::from("C");
 let s2 = String::from("is");
 let s3 = String::from("a");
 let s4 = String::from("programming");
 let s5 = String::from("language.");
 let s = format!("{} {} {} {} {}",s1,s2,s3,s4,s5);
 print!("{}",s);
}
```

Output:

```
C is a programming language.
```

Indexing into Strings

A String is encoded in a UTF-8 sequence. Therefore, the string cannot be indexed. Let's understand this concept through an example:

```
fn main()
{
    let s = String::from("Rustlang");
    print!("{}",s[1]);
}
```

Output:

```
error[E0277]: the trait bound `std::string::String: std::ops::Index<{integer}>` is not satisfied
--> jdoodle.rs:4:17
  |
4 |     print!("{}",s[1]);
  |                 ^^^^ the type `std::string::String` cannot be indexed by `{integer}`
  |
  = help: the trait `std::ops::Index<{integer}>` is not implemented for `std::string::String`

error: aborting due to previous error
```

Accessing through an index is very fast. But, the string is encoded in a UTF-8 sequence which can have multiple bytes and to find the nth character in a string will prove expensive operation.

Slicing Strings

Indexing is not provided in the string as it is not known about the return type of the indexing operation should have byte value, character or a string slice. Rust provides a more specific way to index the string by providing a range within [] rather than a single number.

Let's look:

```
let s = "Hello World";
let a = &s[1..4];
```

In the above scenario, s contains the string literal, i.e., Hello World. We specify [1..4] indices means that we are fetching the substring from a string s indexing from 1 to 3.

```
fn main() {

    let s = "Hello World";
    let a = &s[1..4];
    print!("{}",a);
}
```

Output:

```
ell
```

Methods for iterating over strings

We can access the string in other ways also. We can use the chars() method to iterate over each element of the string.

Let's see a simple example:

```
fn main()
{
    let s = "C is a programming language";
    for i in s.chars()
    {
      print!("{}",i);
    }
}
```

Output:

```
C is a programming language
```

Rust Error handling

- **Error handling** is a mechanism in which Rust determines the possibility of an error and acknowledge you to take some action before the code goes for compilation.
- This mechanism makes the program more robust as it enables you to discover and handles the errors before you deploy the code for production.
- Rust programming language does not contain the exceptions.

There are two types of errors in Rust:

- Unrecoverable error:
- Recoverable error
- **Recoverable Error**: Recoverable errors are the errors which are reported to the user and user can retry the operation. Recoverable errors are not very serious to stop the process entirely. It is represented by **Result<T,E>. Example of recoverable error is "file not found".**
  Where T & E are the generic parameters.
  T-> It is a type of value which is returned in a success case with an 'OK' variant.
  E-> It is a type of the error which is returned in a failure case with an 'Err' variant.
- **Unrecoverable Error**: When the Rust reports an unrecoverable error, then the panic! macro stops the execution of a program. **For example:** "Divide by zero" is an example of unrecoverable error.

Recoverable Error vs Unrecoverable Error

Recoverable Error is an error that can be recovered in some way while **Unrecoverable Error** is an error that cannot be recovered in any way.

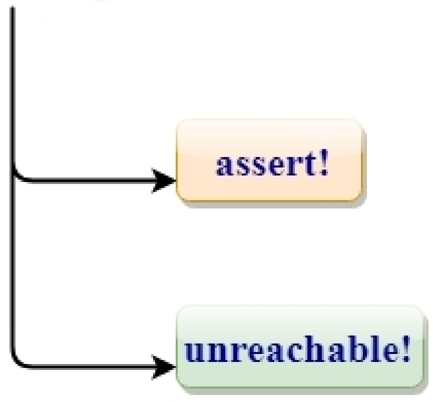
Let's see a scenario of expected behavior:

```
"100".parse();
```

In the above case, "100" is a string, so we are not confirmed whether the above case will work or not. This is the expected behavior. Therefore, it is a recoverable Error.

- **Unexpected behavior**

Unexpected behavior can be shown in two ways:

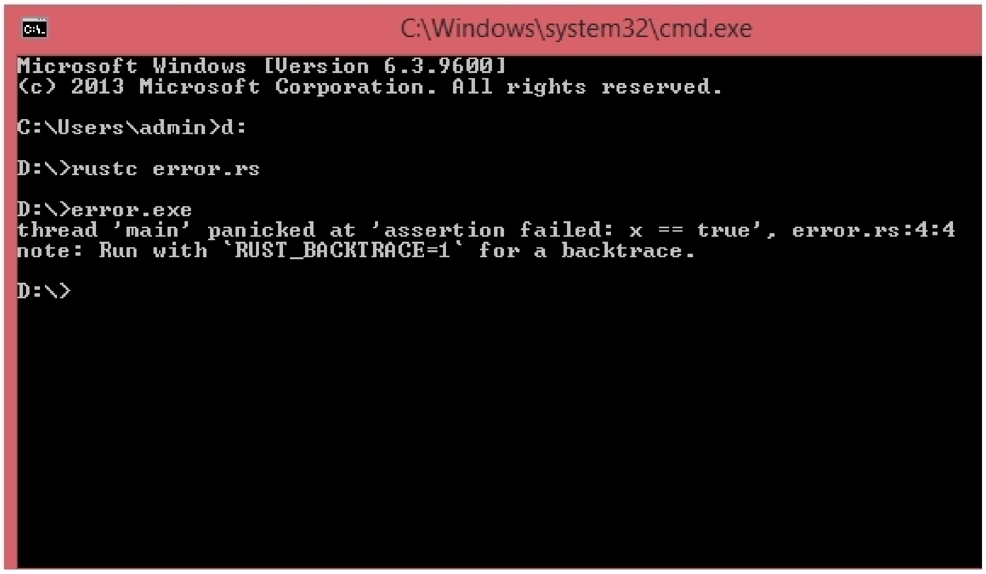


assert!: An **assert!** is used when we want to declare something that it is true. If it is not correct and wrong enough, then the program stops the execution. It invokes the **panic!** , if the expression is not evaluated as true at the runtime.

Let's see a simple example:

```
 fn main()
{
let x : bool = false;
assert!(x==true);
}
```

Output:



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>d:

D:\>rustc error.rs

D:\>error.exe
thread 'main' panicked at 'assertion failed: x == true', error.rs:4:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.

D:\>
```

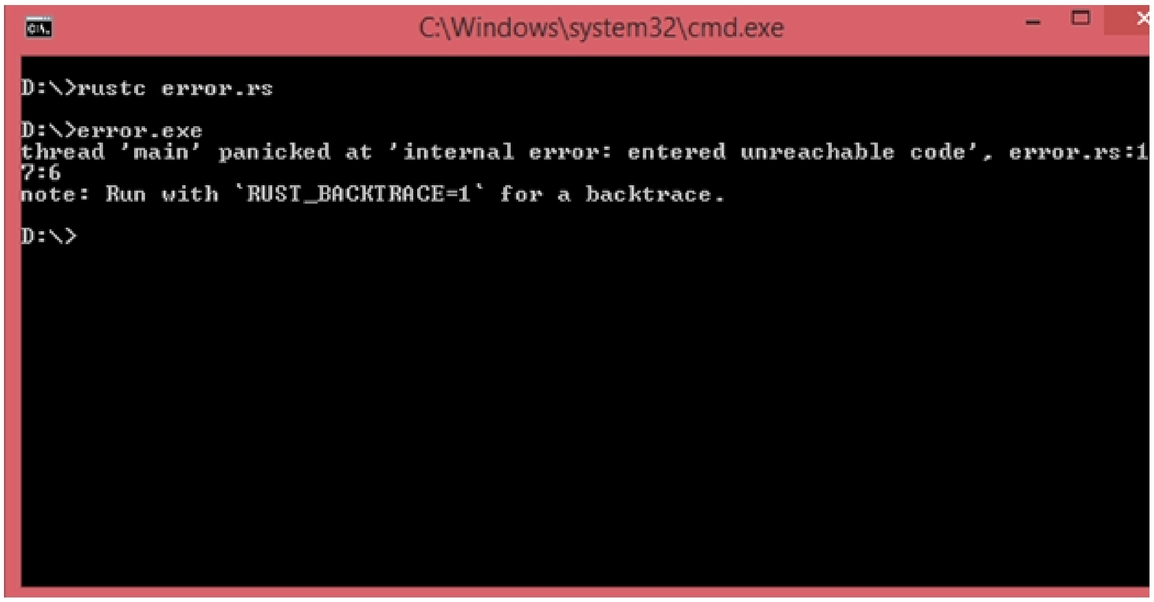
In the above example, the value of x is false and the condition within the assert! Macro is false. Therefore, an assert! Invoke the panic! at the runtime.

unreachable!: An unreachable! Macro is used for the unreachable code. This macro is useful as the compiler can not determine the unreachable code. Unreachable code is determined by the unreachable! at the runtime.

Let's see a simple example:

```
enum Value
{
  Val,
}

fn get_number(_:Value)->i32
{
   5
}
fn find_number(val:Value)-> &'static str
{
  match get_number(val)
  {
    7 => "seven",
    8=> "eight",
    _=> unreachable!()
  }
}

fn main()
{
  println!("{}", find_number(Value::Val));
}
```

Output:

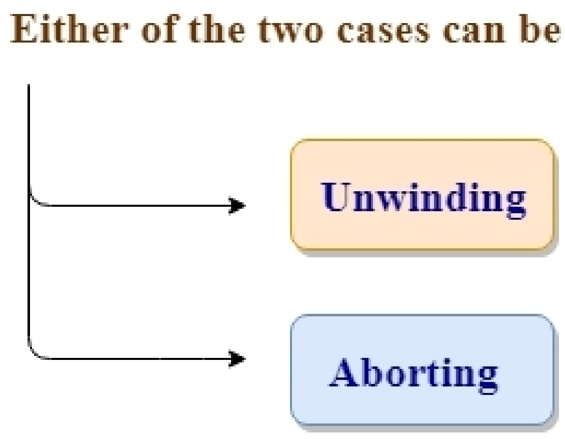


```
D:\>rustc error.rs

D:\>error.exe
thread 'main' panicked at 'internal error: entered unreachable code', error.rs:1
7:6
note: Run with `RUST_BACKTRACE=1` for a backtrace.

D:\>
```

In the above example, the value returned by the get_number() function is 5, and it is matched with each pattern, but it is not matched with any of the patterns. Therefore, the unreachable! macro calls the panic! macro.

Rust Unrecoverable Errors

Unrecoverable Error is an error which is detected, and the programmer can not handle it. When such kind of error occurs, then panic! macro is executed. The panic! prints the failure message. The **panic!** macro unwinds cleans up the stack and then quit.


Either of the two cases can be occurred in response to panic!:
Unwinding
Aborting

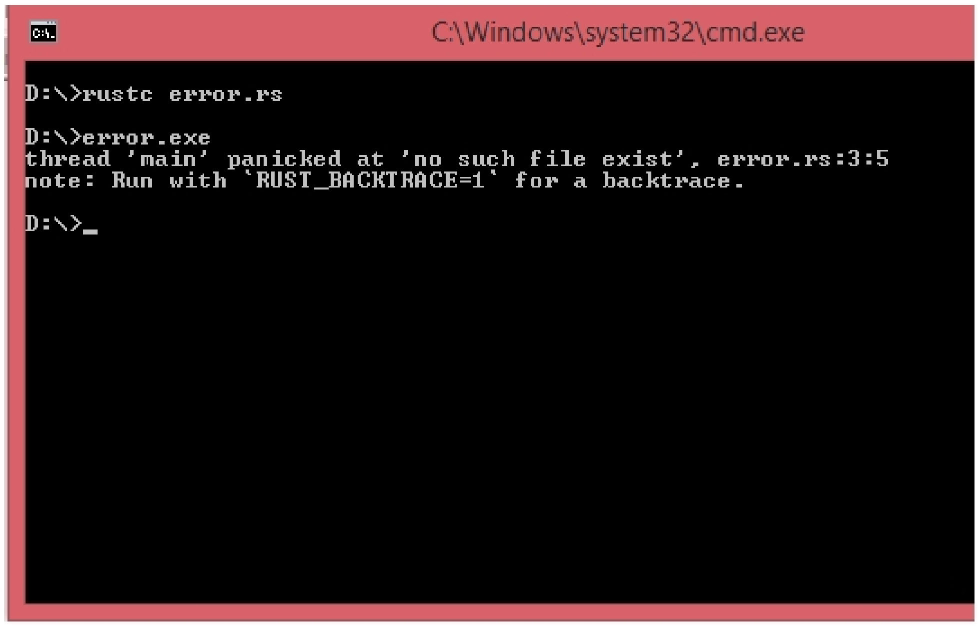
- **Unwinding**: Unwinding is a process of cleaning up the data from the stack memory of each function that it encounters. However, the process of unwinding requires a lot of work. The alternative of Unwinding is an Aborting.
- **Aborting**: Aborting is a process of ending the program without cleaning the data from the stack memory. The operating system will remove the data. If we switch from unwinding to aborting, then we need to add the following statement:

```
panic = 'abort';
```

Let's see a simple example of panic! macro:

```
fn main()

   panic!(?No such file exist?);

```

Output:



```
D:\>rustc error.rs

D:\>error.exe
thread 'main' panicked at 'no such file exist', error.rs:3:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.

D:\>_
```

In the above output, the first line shows the error message which conveys two information, i.e., the panic message and the location of the error. The panic message is "no such file exist" and error.rs:3:5 indicates that it is a third line and fifth character of our file error.rs:3:5 file.

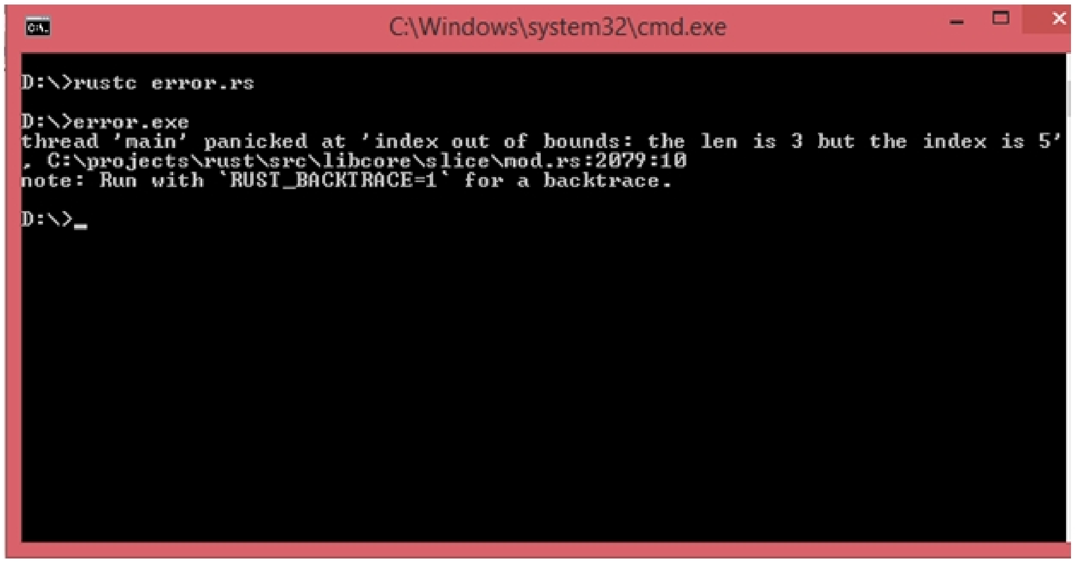
The Advantage of panic! macro

Rust language does not have a buffer overread issue. Buffer overread is a situation, when reading the data from the buffer and the program overruns the buffer, i.e., it reads the adjacent memory. This leads to the violation of the memory safety.

Let's see a simple example:

```
fn main()
{
  let v = vec![20,30,40];
  print!("element of a vector is :",v[5]);
}
```

Output:



```
D:\>rustc error.rs

D:\>error.exe
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 5'
, C:\projects\rust\src\libcore\slice\mod.rs:2079:10
note: Run with `RUST_BACKTRACE=1` for a backtrace.

D:\>_
```

In the above example, we are trying to access the sixth element which is at the index 5. In such a situation, Rust will panic as we are accessing the invalid index. Therefore, Rust will not return anything.

But, in the case of other languages such as C and C++, they would return something, eventhough the vector does not belong to that memory. This is known as Buffer overread, and it leads to the security issues.

Rust Backtrace

Rust Backtrace is the list of all the functions that have been called to know "**what happened to cause the errror.**" We need to set the RUST_BACKTRACE environment variable to get the backtrace.

Rust Recoverable Errors

- Recoverable errors are those errors which are not very serious to stop the program entirely. The errors which can be handled are known as recoverable errors.
- It is represented by Result<T, E>. The Result<T, E> is an enum consists of two variants, i.e., OK<T> and Err<E>. It describes the possible error.

OK<T>: The 'T' is a type of value which returns the OK variant in the success case. It is an expected outcome.

Err<E>: The 'E' is a type of error which returns the ERR variant in the failure. It is an unexpected outcome.

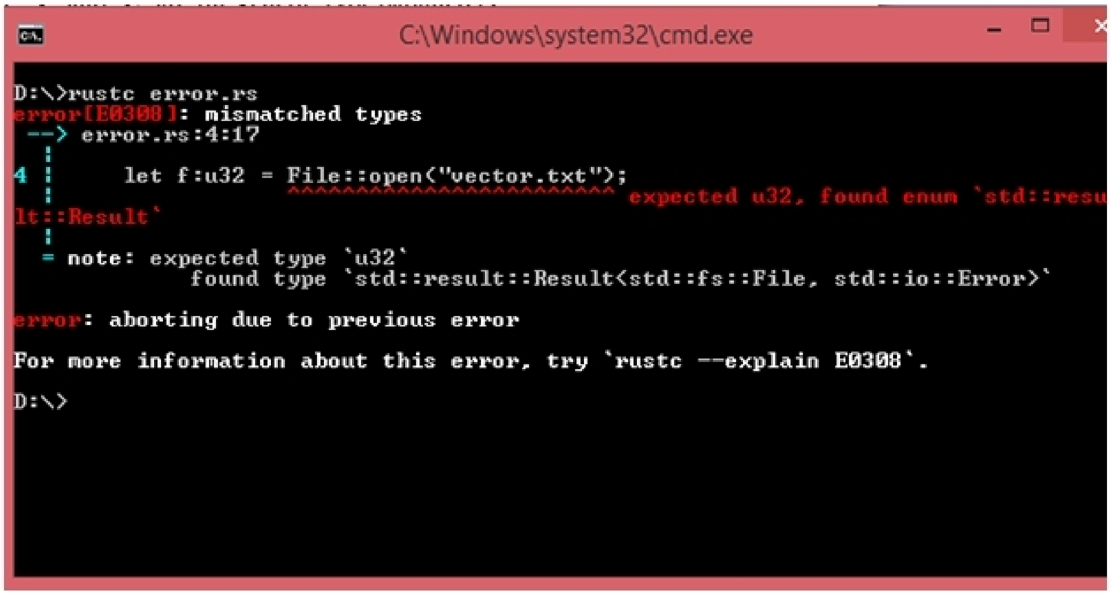
```
Enum Result<T,E>
{
   OK<T>,
   Err<E>,
}
```

- In the above case, Result is the enum type, and **OK<T> & Err<E>** are the variants of enum type where **'T'** and **'E'** are the generic type parameters.
- 'T' is a type of value which will be returned in the success case while 'E' is a type of error which will be returned in the failure case.
- The **Result** contains the generic type parameters, so we can use the Result type and functions defined in the standard library in many different situations where the success and failure values may vary.

Let's see a simple example that returns the Result value:

```
 use std::fs::File;
fn main()
{
    let f:u32 = File::open("vector.txt");
}
```

Output:



```
D:\>rustc error.rs
error[E0308]: mismatched types
 --> error.rs:4:17
  |
4 |     let f:u32 = File::open("vector.txt");
  |                 ^^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum `std::resu
lt::Result`
  |
  = note: expected type `u32`
             found type `std::result::Result<std::fs::File, std::io::Error>`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.

D:\>
```

In the above example, Rust compiler shows that type does not match. The 'f' is a u32 type while File:: open returns the Result<T, E>type. The above output shows that the type of the success value is std::fs:: File and the type of the error value is std::io:: Error.

Note:

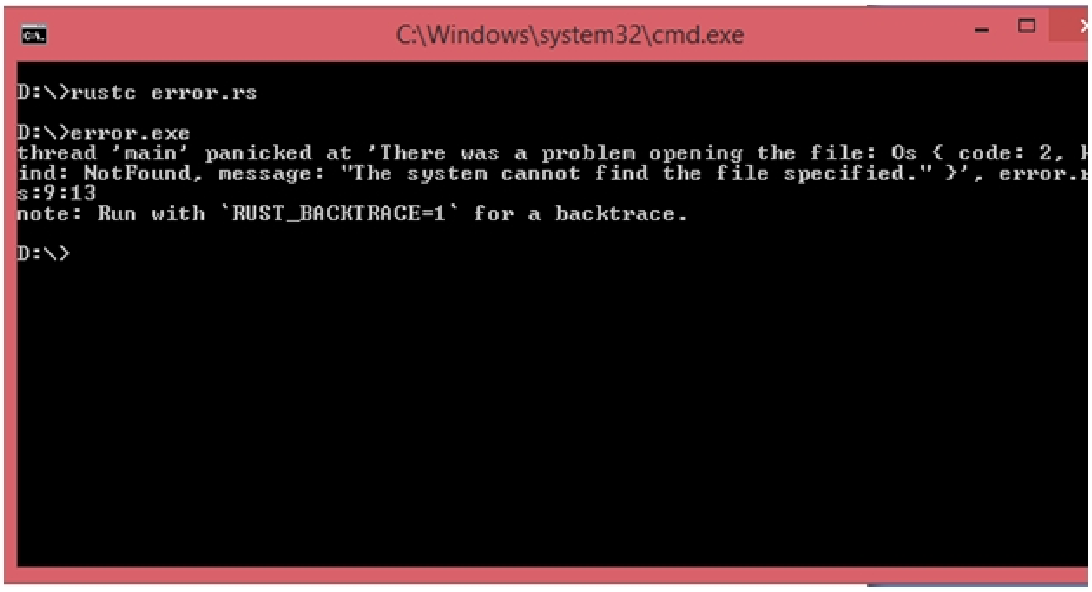
1. The return type of the File:: open is either a success value or failure value. If the file:: open succeeds, then it returns a file handle, and if file:: open fails, then it returns an error value. The Result enum provides this information.
2. **If File:: open succeed**, then f will have an OK variant that contains the file handle, and **if File:: open fails**, then f will have Err variant that contains the information related to the error.

Match Expression to handle the Result variants.

Let's see a simple example of match expression:

```
use std::fs::File;
n main()
{
  let f = File::open("vector.txt");
  match f
  {
    Ok(file) => file,
    Err(error) => {
    panic!("There was a problem opening the file: {:?}", error)
  },
};
```

Output:



```
D:\>rustc error.rs

D:\>error.exe
thread 'main' panicked at 'There was a problem opening the file: Os { code: 2, k
ind: NotFound, message: "The system cannot find the file specified." }', error.r
s:9:13
note: Run with `RUST_BACKTRACE=1` for a backtrace.

D:\>
```

Program Explanation

- In the above example, we can access the enum variants directly without using the **Result**:: before **OK** and **Err** variant.
- If the result is **OK**, then it returns the file and stores it in the 'f' variable. After the match, we can perform the operations in the file either reading or writing.
- The second arm of the match works on the Err value. If Result returns the Error value, then panic! runs and stops the execution of a program.

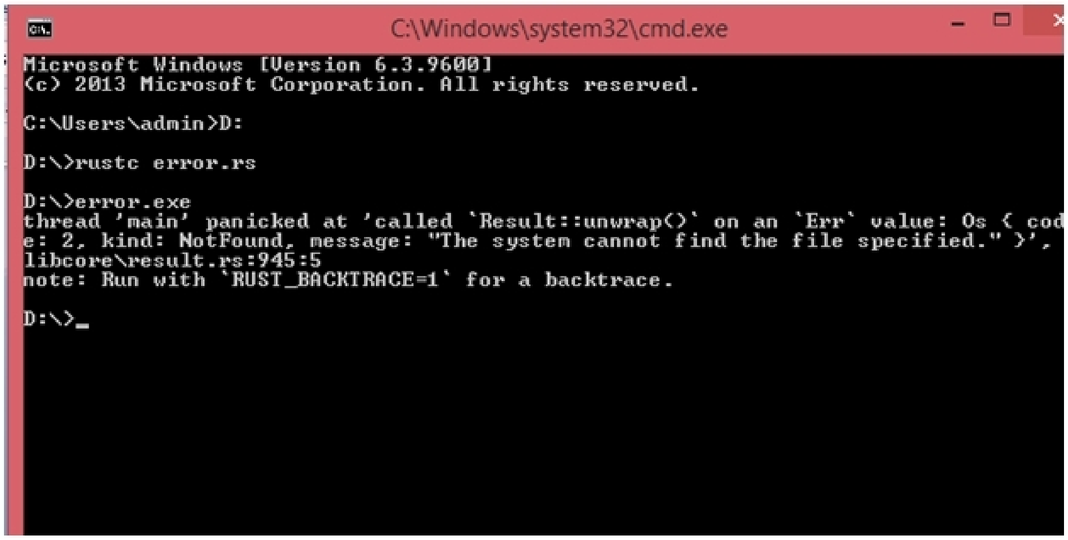
Panic on Error: unwrap()

- The Result<T, E> has many methods to provide various tasks. One of the methods is unwrap() method. The unwrap() method is a shortcut method of a match expression. The working of unwrap() method and match expression is the same.
- If the Result value is an **OK** variant, then the unwrap() method returns the value of the **OK** variant.
- If the Result value is **an Err** variant, then the unwrap() method calls the panic! macro.

Let's see a simple example:

```rust
use std::fs::File;

fn main()
{
    File::open("hello.txt").unwrap();
}
```

Output:



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>D:

D:\>rustc error.rs

D:\>error.exe
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Os { cod
e: 2, kind: NotFound, message: "The system cannot find the file specified." }',
libcore\result.rs:945:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.

D:\>_
```

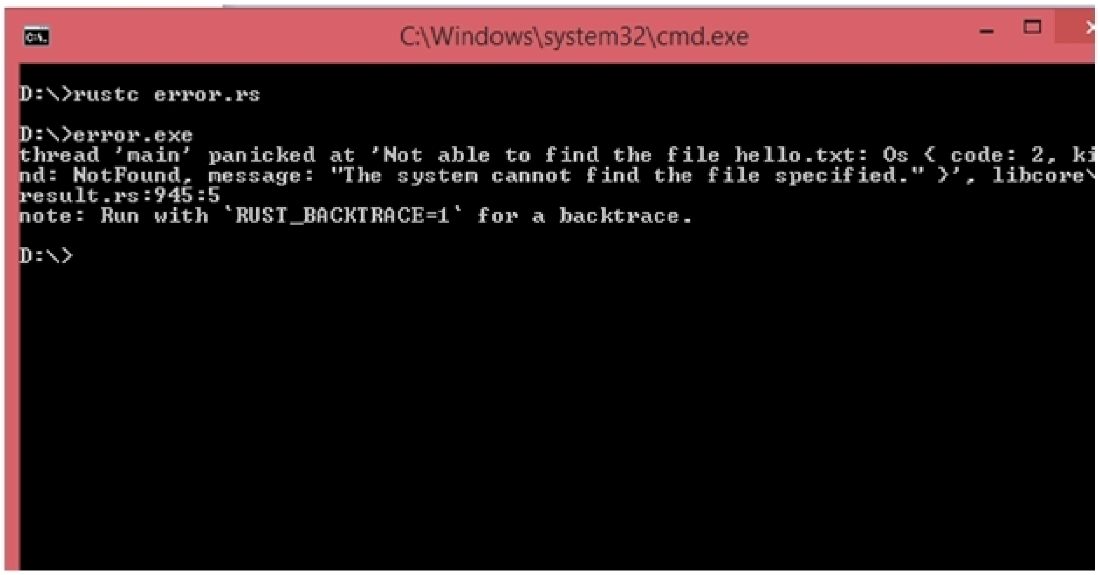
Panic on Error: expect()

- The expect() method behaves in the same way as the unwrap() method, i.e., both methods call the panic! to display the error information.
- The difference between the expect() and unwrap() method is that the error message is passed as a parameter to the expect() method while unwrap() method does not contain any parameter. Therefore, we can say that the expect() method makes tracking of the panic! source easier.

Let's see a simple example of expect()

```
use std::fs::File;
fn main()
{
    File::open("hello.txt").expect("Not able to find the file hello.txt");
}
```

Output:



```
D:\>rustc error.rs

D:\>error.exe
thread 'main' panicked at 'Not able to find the file hello.txt: Os { code: 2, ki
nd: NotFound, message: "The system cannot find the file specified." }', libcore\
result.rs:945:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.

D:\>
```

In the above output, the error message is displayed on the output screen which we specify in our program, i.e., "**Not able to find the file hello.txt**" and this makes easier for us to find the code from where the error is coming from. If we contain multiple unwrap() method, then it becomes difficult to find where the unwrap() method is causing panic! as as panic! shows the same error messages for all the errors.

Propagating Errors

Propagating error is a mechanism in which errors are forwarded from one function to other function. Errors are propagated to the calling function where more information is available so that the error can be handled. **Suppose we have a file named as 'a.txt' and it contains the text "Rustlang." We want to create a program that performs the reading operation on this file. Let's work on this example**.

Let's see a simple example:

```
 use std::io;
use std::io::Read;
use std::fs::File;
fn main()
{
 let a = read_username_from_file();
 print!("{:?}",a);
}
fn read_username_from_file() -> Result<String, io::Error>
{
   let f = File::open("a.txt");
   let mut f = match f {
   Ok(file) => file,
   Err(e) => return Err(e),
   };
   let mut s = String::new();
   match f.read_to_string(&mut s) {
      Ok(_) => Ok(s),
      Err(e) => Err(e),
   }
}
```

Program Explanation

- The read_username_from_file() function returns a value of the type Result<T, E> where 'T' is a type of String and 'E' is a type of io:Error.
- If the function succeeds, then it returns an OK value that holds a String, and if the function fails, then it returns an Err value.

- This function starts by calling the File:: open function. If the File:: open function fails, then the second arm of the match will return the Err value, and if the File:: open function succeeds, then it stores the value of the file handle in variable f.
- If the File:: open function succeeds, then we create the variable of a String. If read_to_string() method succeeds, then it returns the text of the file otherwise it returns the error information.
- Suppose we have an external file with a name 'a.text' and contains the text "Rustlang." Therefore, this program reads the file 'a.text' and displays the content of the file.

Shortcut for propagating the errors: the '?' operator

The use of '?' operator reduces the length of the code. The '?' operator is the replacement of the match expressions means that the '?' operator works in the same way as the match expressions do. **Suppose we have a file named as 'a.txt' and it contains the text "Rustlang." We want to create a program that performs the reading operation on this file. Let's work on this example**.

Let's see a simple example.

```
 use std::io;
use std::io::Read;
use std::fs::File;
fn main()
{
  let a = read_username_from_file();
  print!("{:?}",a);
}
fn read_username_from_file() -> Result<String, io::Error>
{
   let mut f = File::open("a.txt")?;
   let mut s = String::new();
   f.read_to_string(&mut s)?;
  Ok(s)
}
```

In the above example, '?' operator is used before the Result value type. If Result is OK, then it returns the value of OK variant, and if Result is an Err, then it returns the error information.

Difference b/w '?' operator & match expression

- The errors which are used with the '?' operator moves through the 'from' function and the 'from' function is defined in the from trait in the standard library.
- When the '?' operator calls the 'from' function, then this function converts the error type into the error type defined in the return type of the current function.
- If no error occurs, then the '?' operator at the end of any function returns the value of OK, and if the error occurs, then the value of Err is returned.
- It makes the implementation of the function simpler.

Chaining method calls after the '?' operator

We can even shorten the code of a program more by using the **chaining method calls after the '?' operator.**

Let's see a simple example:

```
 use std::io;
use std::io::Read;
use std::fs::File;
fn main()
{
  let a = read_username_from_file();
  print!("{:?}",a);
}
fn read_username_from_file() -> Result<String, io::Error>
{
   let mut s = String::new();
  File::open("a.txt")?.read_to_string(&mut s)?;
  Ok(s)
}
```

Program Explanation

In the above example, we have chained the call of read_to_string() to the result of the call of File::open("a.txt")?. We place the '?' operator at the end of the call of read_to_string(). It returns OK value if both the functions, i.e., read_to_string() and File::open("a.txt") succeeds otherwise it returns the error value.

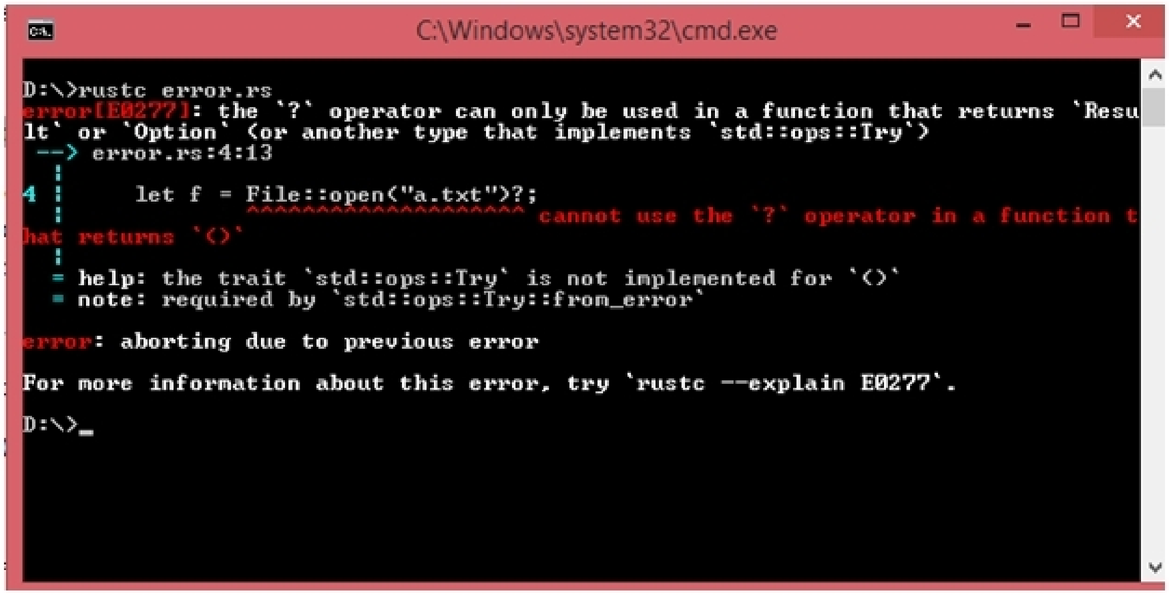
Limitation of '?' operator

The '?' operator can only be used in the functions that return the Result type value. As the '?' operator works similarly as the match expression. The match expression works only on the Result return type.

Let's understand this through a simple example.

```
use std::fs::File;
fn main()
{
   let f = File::open("a.txt")?;
}
```

Output:



```
D:\>rustc error.rs
error[E0277]: the `?` operator can only be used in a function that returns `Resu
lt` or `Option` (or another type that implements `std::ops::Try`)
 --> error.rs:4:13
  |
4 |     let f = File::open("a.txt")?;
  |             ^^^^^^^^^^^^^^^^^^^^ cannot use the `?` operator in a function t
hat returns `()`
  |
  = help: the trait `std::ops::Try` is not implemented for `()`
  = note: required by `std::ops::Try::from_error`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0277`.

D:\>_
```

Rust Generics

When we want to create the function of multiple forms, i.e., the parameters of the function can accept the multiple types of data. This can be achieved through generics. Generics are also known as **'parametric polymorphism' where poly is multiple, and morph is form.**

There are two ways to provide the generic code:

- Option<T>
- Result<T, E>

1. Option<T>: Rust standard library provides Option where 'T' is the generic data type. It provides the generic over one type.

```
enum Option<T>
{
   Some(T),
   None,
}
```

In the above case, **enum** is the custom type where <T> is the generic data type. We can substitute the 'T' with any data type. Let's look at this:

```
let x : Option<i32> = Some(10);  // 'T' is of type i32.
let x : Option<bool> = Some(true);  // 'T' is of type bool.
let x : Option<f64> = Some(10.5); // 'T' is of type f64.
let x : Option<char> = Some('b'); // 'T' is of type char.
```

In the above case, we observe that 'T' can be of any type, i.e., i32, bool, f64 or char. But, if the type on the left-hand side and the value on the right hand side didn't match, then the error occurs. Let's look at this:

```
let x : Option<i32> = Some(10.8);
```

In the above case, type on the left-hand side is i32, and the value on the right-hand side is of type f64. Therefore, the error occurs **"type mismatched"**.

2. Result<T,E>: Rust standard library provides another data type **Result<T,E> which is generic over two type, i.e., T &E:**

```
 enum Result<T,E>
```

```
{
    OK(T),
      Err(E),
}
```

Generic functions

Generics can be used in the functions, and we place the generics in the signature of the function, where the data type of the parameters and the return value is specified.

- **When the function contains a single argument of type 'T'**.

Syntax:

```
fn function_name<T>(x:T)

 // body of the function.
```

The above syntax has two parts:

- <T> : The given function is a generic over one type.
- (x : T) : x is of type T.

When the function contains multiple arguments of the same type.

```
fn function_name<T>(x:T, y:T)

 // body of the function.
```

When the function contains arguments of multiple types.

```
fn function_name<T,U>(x:T, y:U)

   // Body of the function.
```

```
 fn main()
{
 let a = vec![1,2,3,4,5];
 let b = vec![2.3,3.3,4.3,5.3];
 let result = add(&a);
 let result1 = add(&b);
 println!("The value of result is {}",result);
 println!("The value of result1 is {}",result1);
}

fn add<T>(list:&[T])->T
{
 let mut c =0;
 for &item in list.iter()
 {
   c= c+item;
 }
 c}
```

Struct Definitions

Structs can also use the generic type parameter in one or more fields using <> operator.

Syntax:

```
struct structure_name<T>

  // Body of the structure.
```

In the above syntax, we declare the generic type parameter within the angular brackets just after the structure_name, and then we can use the generic type inside the struct definition.

Let's see a simple example:

```
 struct Value<T>
{
  a:T,
  b:T,
}
fn main()
{
  let integer = Value{a:2,b:3};
  let float = Value{a:7.8,b:12.3};
  println!("integer values : {},{}",integer.a,integer.b);
  println!("Float values :{},{}",float.a,float.b);
}
```

Output:

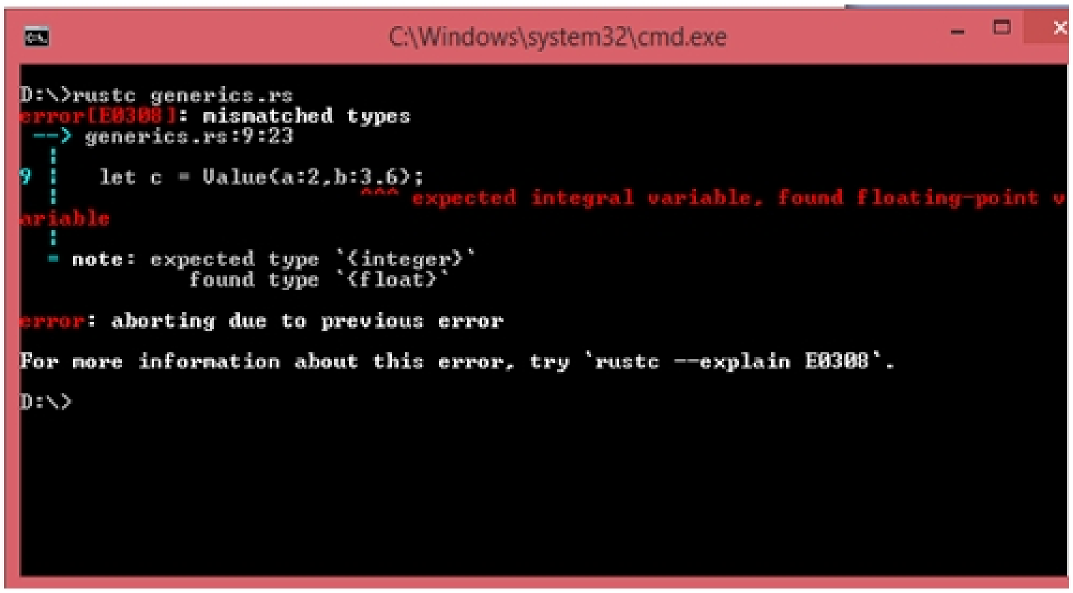
```
integer values : 2,3
Float values : 7.8,12.3
```

In the above example, Value<T> struct is generic over one type and a and b are of the same type. We create two instances integer and float. Integer contains the values of type i32 and float contains the values of type f64.

Let's see another simple example.

```
 struct Value<T>
{
  a:T,
  b:T,
}
fn main()
{
  let c = Value{a:2,b:3.6};
  println!("c values : {},{}",c.a,c.b);
}
```

Output:



```
D:\>rustc generics.rs
error[E0308]: mismatched types
 --> generics.rs:9:23
  |
9 |   let c = Value{a:2,b:3.6};
  |                       ^^^ expected integral variable, found floating-point v
ariable
  |
  = note: expected type `{integer}`
             found type `{float}`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.

D:\>
```

In the above example, Value<T> struct is generic over one type, and a and b are of the same type. We create an instance of 'c'. The 'c' contains the value of different types, i.e., i32 and f64. Therefore, the Rust compiler throws the "mismatched error".

Enum Definitions

An enum can also use the generic data types.Rust standard library provides the Option<T> enum which holds the generic data type. The Option<T> is an enum where 'T' is a generic data type.

- **Option<T>**

It consists of two variants, i.e., Some(T) and None.

Where Some(T) holds the value of type T and None does not contain any value.

Let's look:

```
enum Option<T>
{
   Some(T),
   None,
}
```

In the above case, Option is an enum which is generic over one type 'T'. It consists of two variants Some(T) and None.

- **Result<T, E>**: We can create the generic of multiple types. This can be achieved through Result<T, E>.

```
enum Result<T,E>
{
   OK(T),
   Err(E),
}
```

In the above case, Result<T, E> is an enum which is generic over two types, and it consists of two variants, i.e., OK(T) and Err(E).

OK(T) holds the value of type 'T' while Err(E) holds the value of type 'E'.

Method Definitions

We can implement the methods on structs and enums.

Let's see a simple example:

```
struct Program<T> {
   a: T,
   b: T,
}
impl<T> Program<T>
{
   fn a(&self) -> &T
{
      &self.a
   }
}
fn main() {
let p = Program{ a: 5, b: 10 };

   println!("p.a() is {}", p.a());
}
```

Output:

```
p.a() is 5
```

In the above example, we have implemented the method named as 'a' on the Program<T> that returns a reference to the data present in the variable a.

We have declared the 'T' after impl to specify that we are implementing the method on Program<T>.

Resolving Ambiquities

Rust compiler automatically infers the generic parameters. Let's understand this through a simple scenario:

```
Let mut v = Vec::new();   // creating a vector.
v.push(10); // inserts integer value into the vector. Therefore, v is of i32 type.
println!("{:?}", v); // prints the value of v.
```

In the above case, we insert the integer value into the vector. Therefore, the Rust compiler got to know that the vector v has the type i32.

If we delete the second last line, then it looks like;

```
Let mut v = Vec::new();   // creating a vector.
println!("{:?}", v); // prints the value of v.
```

The above case will throw an error that "it cannot infer the type for T".

- **We can solve the above case in two ways:**

1. We can use the following annotation:

```
let v : Vec<bool> = Vec::new();
println!("{:?}",v) ;
```

2. We can bind the generic parameter 'T' by using the 'turbofish' ::<> operator:

```
let v = Vec :: <bool> :: new();
println!("{:?}",v) ;
```

Rust Trait

- Rust trait is a feature of a Rust language that describes the functionality of each type that it can provide.
- A trait is similar to the feature of an interface defined in other languages.
- A trait is a way to group the method signatures to define a set of behaviors.
- A Trait is defined by using the trait keyword.

The Syntax of the trait:

```
trait trait_name

    //body of the trait.

```

In the above case, we declare the trait followed by the trait name. Inside the curly brackets, method signature is declared to describe the behavior of a type that implements the trait.

Let's see a simple example:

```
 struct Triangle
{
  base : f64,
  height : f64,
}
trait HasArea
{
  fn area(&self)->f64;
}

impl HasArea for Triangle
{
  fn area(&self)->f64
  {
    0.5*(self.base*self.height)
  }
}
```

```
fn main()
{
  let a = Triangle{base:10.5,height:17.4};
  let triangle_area = a.area();
  println!("Area of a triangle is {}",triangle_area);
}
```

Output:

```
Area of a triangle is 91.35
```

In the above example, trait named as HasArea is declared which contains the declaration of area() function. HasArea is implemented on the type Triangle. An area() function is simply called by using the instance of the structure, i.e., a.area().

Trait as Arguments

Traits can also be used as arguments of many different types.

The above example implements the HasArea trait, and it contains the definition of the area() function. We can define the calculate_area() function that calls the area() function, and the area() function is called using the instance of the type that implements the HasArea trait.

Let's look at the syntax:

```
fn calculate_area(item : impl HasArea)

  println!("Area of the triangle is : {}",item.area());
}
```

Trait bounds on Generic functions

Traits are useful because they describe the behavior of different methods. But, Generic functions does not follow this constraint. **Let's understand this through a simple scenario:**

```
fn calculate_area<T>( item : T)

   println!(?Area of a triangle is {}?, item.area());
```

In the above case, Rust compiler throws an "error that no method named found of type T". If we bound the trait to the generic T, then the following error can be overcome:

```
 fn calculate_area<T : HasArea> (item : T)
{
println!("Area of a triangle is {} ",item.area());


}
```

In the above case, <T: HasArea> means "T can be of any type that implements HasArea trait". Rust compiler got to know that any type that implements the HasArea trait will have an area() function.

Let's see a simple example:

```
 trait HasArea
{
  fn area(&self)->f64;
}
struct Triangle
{
  base : f64,
  height : f64,
}

impl HasArea for Triangle
{
  fn area(&self)->f64
  {
    0.5*(self.base*self.height)
  }
}
struct Square
{
  side : f64,
}

impl HasArea for Square
{
  fn area(&self)->f64
  {
    self.side*self.side
  }
}
fn calculate_area<T : HasArea>(item : T)
{
  println!("Area is : {}",item.area());
}

fn main()
{
  let a = Triangle{base:10.5,height:17.4};
  let b = Square{side : 4.5};
  calculate_area(a);
  calculate_area(b);
}
```

Output:

```
Area is : 91.35
Area is : 20.25
```

In the above example, calculate_area() function is generic over "T".

Rules for implementing traits

There are two limitations to implementing the trait:

- If the trait is not defined in your scope, then it cannot be implemented on any data type.

Let's see a simple example:

```
use::std::fs::File;
fn main()
{
 let mut f = File::create("hello.txt");
 let str = "Rustlang";
 let result = f.write(str);
}
```

Output:

```
error : no method named 'write' found.
        let result = f.write(str);
```

In the above case, Rust compiler throws an error, i.e., "no method named 'write' found" as use::std::fs::File; namespace does not contain the write() method. Therefore, we need to **use the Write trait to remove the compilation error**.

- The trait which we are implementing must be defined by us. For example: If we define the **HasArea** trait, then we can implement this trait for the type i32. However, we could not implement the **toString** trait defined by the Rust for the type i32 as both the type and trait are not defined in our crate.

Multiple trait bounds

- **Using '+' operator.**

If we want to bound the multiple traits, we use the + operator.

Let's see a simple example:

```
 use std::fmt::{Debug, Display};
fn compare_prints<T: Debug + Display>(t: &T)
 {
 println!("Debug: '{:?}'", t);
 println!("Display: '{}'", t);
}




fn main() {
   let string = "Rustlang";
   compare_prints(&string);
   }
```

Output:

```
Debug: ' "Rustlang"'
Display: ' Rustlang'
```

In the above example, Display and Debug traits are bounded to the type 'T' by using the '+' operator.

- **Using 'where' clause.**
  - A bound can be written using a 'where' clause which appears just before the opening bracket '{'.
  - A 'where' clause can also be applied to the arbitrary types.
  - When 'where' clause is used, then it makes the syntax more expressive than the normal syntax.

Let's look:

```
fn fun<T: Display+Debug, V: Clone+Debug>(t:T,v:V)->i32

   //block of code;

```

When 'where' is used in the above case:

```
fn fun<T, V>(t:T, v:V)->i32
  where T : Display+ Debug,
            V : Clone+ Debug

     //block of code;
```

In the above cases, the second case where we have used the 'where' clause makes the program more expressive and readable.

Let's see a simple example:

```
 trait Perimeter
{
  fn a(&self)->f64;
}
struct Square
{
  side : f64,
}
impl Perimeter for Square
{
  fn a(&self)->f64
  {
    4.0*self.side
  }
}
struct Rectangle
{
 length : f64,
 breadth : f64,
}
impl Perimeter for Rectangle

{
 fn a(&self)->f64
 {
   2.0*(self.length+self.breadth)
 }
}
fn print_perimeter<Square,Rectangle>(s:Square,r:Rectangle)
  where Square : Perimeter,
        Rectangle : Perimeter
        {
          let r1 = s.a();
          let r2 = r.a();
          println!("Perimeter of a square is {}",r1);
          println!("Perimeter of a rectangle is {}",r2);
        }
        fn main()
        {
          let sq = Square{side : 6.2};
          let rect = Rectangle{length : 3.2,breadth:5.6};
          print_perimeter(sq,rect);
        }
```

Output:

```
Perimeter of a square is 24.8
Perimeter of a rectangle is 17.6
```

Default methods

A default method can be added to the trait definition if the definition of a method is already known.

Let's look:

```
trait Sample

  fn a(&self);
  fn b(&self)
  {
      println!("Print b");
  }

```

In the above case, the default behavior is added to the trait definition. We can also override the default behavior. **Let' look at this scenario through an example:**

```
trait Sample
{
fn a(&self);
fn b(&self)
{
 println!("Print b");
}
}

struct Example
{
a:i32,
b:i32,
}



impl Sample for Example
{
  fn a(&self)
  {
   println!("Value of a is {}",self.a);
  }

  fn b(&self)
  {
   println!("Value of b is {}",self.b);
  }
}
fn main()
{
  let r = Example{a:5,b:7};
  r.a();
  r.b();
}
```

Output:

```
Value of a is : 5
Value of b is : 7
```

In the above example, the behavior of b() function is defined in the trait is overridden. Therefore, we can conclude that we can override the method which is defined in the trait.

Inheritance

The trait which is derived from another trait is known as inheritance. Sometimes, it becomes necessary to implement the trait that requires implementing another trait. If we want to derive 'B' trait from 'A' trait, then it looks like:

```
trait B : A;
```

Let's see a simple example:

```
 trait A
{
  fn f(&self);
}
trait B : A
{
  fn t(&self);
}
struct Example
{
  first : String,
  second : String,
}
impl A for Example
{
  fn f(&self)
  {

   print!("{} ",self.first);
  }

 }
 impl B for Example
 {
  fn t(&self)
  {
    print!("{}",self.second);
  }
}
fn main()
{
  let s = Example{first:String::from("Rustlang"),second:String::from("tutorial")};
  s.f();
  s.t();
}
```

Output:

```
Rustlang tutorial
```

In the above example, our program is implementing the 'B' trait. Therefore, it also requires to implement the 'A' trait. If our program does not implement the 'A' trait, then the Rust compiler throws an error.

Rust Lifetime

- Lifetime defines the scope for which reference is valid.
- Lifetimes are implicit and inferred.
- Rust uses the generic lifetime parameters to ensure that actual references are used which are valid.

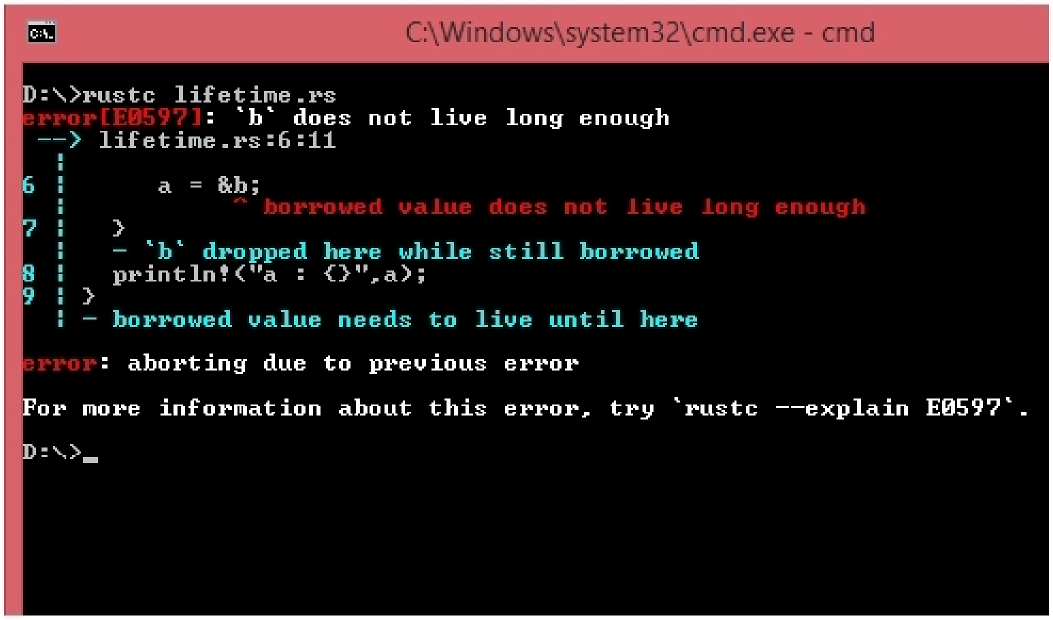
Preventing Dangling references with Lifetimes

When a program tries to access the invalid reference is known as a **Dangling reference**. The pointer which is pointing to the invalid resource is known as a **Dangling pointer**.

Let's see a simple example:

```
fn main()
{
 let a;
 {
   let b = 10;
    a = &b;
 }
 println!("a : {}",a);
}
```

Output:

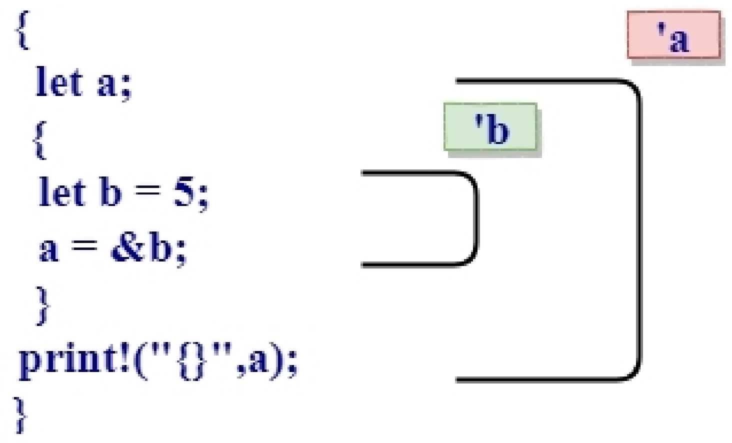


```
D:\>rustc lifetime.rs
error[E0597]: `b` does not live long enough
 --> lifetime.rs:6:11
  |
6 |       a = &b;
  |            ^ borrowed value does not live long enough
7 |    }
  |    - `b` dropped here while still borrowed
8 |    println!("a : {}",a);
9 | }
  | - borrowed value needs to live until here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0597`.

D:\>_
```

In the above example, the outer scope contains the variable whose named as 'a' and it does not contain any value. An inner scope contains the variable 'b' and it stores the value 10. The reference of 'b' variable is stored in the variable 'a'. When the inner scope ends, and we try to access the value of 'a'. The Rust compiler will throw a compilation error as 'a' variable is referring to the location of the variable which is gone out of the scope. Rust will determine that the code is invalid by using the **borrow checker**.

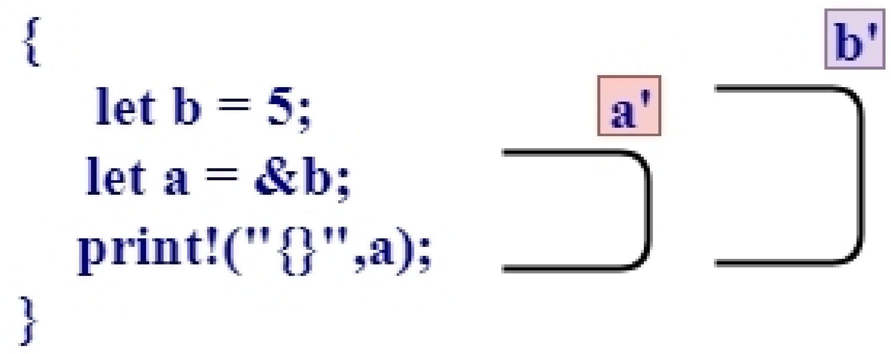
Borrow checker

The borrow checker is used to resolve the problem of dangling references. The borrow checker is used to compare the scopes to determine whether they are valid or not.



```
{
  let a;
  {
  let b = 5;
  a = &b;
  }
 print!("{}",a);
}
```

In the above example, we have annotated the lifetime of 'a' variable with the 'a and the lifetime of 'b' variable with the 'b. At the compile time, Rust will reject this program as the lifetime of 'a' variable is greater than the lifetime of 'b' variable. The above code can be fixed so that no compiler error occurs.



```
{
    let b = 5;
    let a = &b;
    print!("{}",a);
}
```

In the above example, the lifetime of 'a' variable is shorter than the lifetime of 'b' variable. Therefore, the above code runs without any compilation error.

Lifetime annotation syntax

- Lifetime annotation does not change how long any of the references live.
- Functions can also accept the references of any lifetime by using the generic lifetime parameter.
- Lifetime annotation describes the relationship among the lifetimes of multiple parameters.

Steps to be followed for the lifetime annotation syntax:

- The names of the lifetime parameters should start with (') apostrophe.
- They are mainly lowercase and short. For example: 'a.
- Lifetime parameter annotation is placed after the '&' of a reference and then space to separate the annotation from the reference type.

Some examples of lifetime annotation syntax are given below:

- &i32   // reference
- & 'a i32   // reference with a given lifetime.
- & 'a mut i32   // mutable reference with a given lifetime.

Lifetime Annotations in Function Signatures

The 'a represents the lifetime of a reference. Every reference has a lifetime associated with it. We can use the **lifetime annotations** in function signatures as well. The generic lifetime parameters are used between angular brackets <> , and the angular brackets are placed between the function name and the parameter list. Let's look:

```
1. fn fun<'a>(...);
```

In the above case, fun is the function name which has one lifetime, i.e., **'a. If a function contains two reference parameters with two different lifetimes, then it can be represented as:**

```
1. fn fun<'a,'b>(...);
```

If a function contains a single variable named as 'y'.

If 'y' is an immutable reference, then the parameter list would be:

```
1. fn fun<'a>(y : & 'a i32);
```

If 'y' is a mutable reference, then the parameter list would be:

```
1. fn fun<'a>(y : & 'a mut i32);
```

Both & 'a i32 and & 'a mut i32 are similar. The only difference is that 'a is placed between the & and mut.

& mut i32 means "mutable reference to an i32" .

& 'a mut i32 means "mutable reference to an i32 with a lifetime 'a".

Lifetime Annotations in struct

We can also use the explicit lifetimes in the struct as we have used in functions.

Let's look:

```
struct Example

   x : & 'a i32,  //  x is a variable of type i32 that has the lifetime 'a.

```

Let's see a simple example:

```
   struct Example<'a> {
    x: &'a i32,
}
fn main() {
    let y = &9;
    let b = Example{ x: y };
    println!("{}", b.x);
}
```

Output:

```
9
```

impl blocks

We can implement the struct type having a lifetime 'a using impl block.

Let's see a simple example:

```
 struct Example<'a> {
    x: &'a i32,
}
impl<'a> Example<'a>
{
fn display(&self)
{
  print!("Value of x is : {}",self.x);
}
}
fn main() {
    let y = &90;
    let b = Example{ x: y };
    b.display();
}
```

Output:

```
Value of x is : 90
```

Multiple Lifetimes

There are two possibilities that we can have:

- Multiple references have the same lifetime.
- Multiple references have different lifetimes.

When references have the same lifetime.

```
fn fun <'a>(x: & 'a i32 , y: & 'a i32) -> & 'a i32

   //block of code.
```

In the above case, both the references x and y have the same lifetime, i.e., 'a.

When references have the different lifetimes.

```
fn fun<'a , 'b>(x: & 'a i32 , y: & 'b i32)

   // block of code.

```

In the above case, both the references x and y have different lifetimes, i.e., 'a and 'b respectively.

'static

The lifetime named as 'static is a special lifetime. It signifies that something has the lifetime 'static will have the lifetime over the entire program. Mainly 'static lifetime is used with the strings. The references which have the 'static lifetime are valid for the entire program.

Let's look:

```
let s : & 'static str = "Rustlang tutorial" ;
```

Lifetime Ellision

Lifetime Ellision is an inference algorithm which makes the common patterns more ergonomic. Lifetime Ellision makes a program to be ellided.

Lifetime Ellision can be used anywhere:

- & 'a T
- & 'a mut T
- T<'a>

Lifetime Ellision can appear in two ways:

- **Input lifetime**: An input lifetime is a lifetime associated with the parameter of a function.
- **Output lifetime**: An output lifetime is a lifetime associated with the return type of the function.

Let's look:

```
1.  fn fun<'a>( x : & 'a i32);                  // input lifetime
2.  fn fun<'a>() -> & 'a i32;                   // output lifetime
3.  fn fun<'a>(x : & 'a i32)-> & 'a i32;       // Both input and output lifetime.
```

Rules of Lifetime Ellision:

- Each parameter passed by the reference has got a distinct lifetime annotation.

```
fn fun( x : &i32, y : &i32)
        {
        }
```

→

```
fn fun<'a , 'b>( x :& 'a i32, y : & 'b i32)
        {
        }
```

- If the single parameter is passed by reference, then the lifetime of that parameter is assigned to all the elided output lifetimes.

```
fn fun(x : i32, y : &i32) -> &i32
        {
        }
```

→

```
fn fun<'a>(x : i32, y : & 'a i32) -> & 'a i3
        {
        }
```

- If multiple parameters passed by reference and one of them is &self or &mut self, then the lifetime of self is assigned to all the elided output lifetimes.

```
fn fun(&self, x : &str)
        {
        }
→
fn fun<'a,'b>(& 'a self, x : & 'b str) -> & 'a str
        {
        }
```

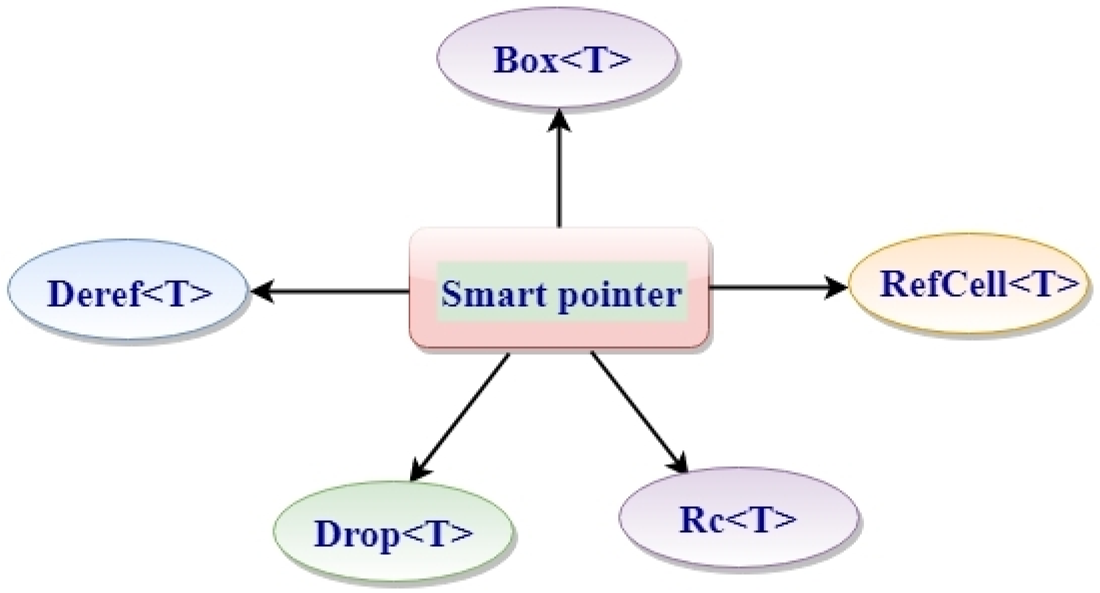
For Example:

```
fn fun( x : &str);                         // Elided form.
fn fun<'a>(x : & 'a str) -> & 'a str;    // Expanded form.
```

Rust Smart Pointers

- A Smart Pointer is a data structure that behaves like a pointer while providing additional features such as memory management or bound checking.
- Smart Pointers keep track of the memory that it points to, and is also used to manage other resources such as Fils handles and network connections.
- Smart pointers were first used in the C++ language.
- Reference is also a kind of pointer, but it does not have additional capabilities other than referring to the data. Reference is represented by '&' operator.
- A Smart Pointer provides the additional functionalities beyond that provided by the reference. The most common feature that smart pointer provides "reference counting smart pointer type". This feature enables us to have multiple owners of data by keeping track of the owners, and if no owner remains, then it cleans up the data.
- References are the pointers that only borrow the data while smart pointers are the pointers that own the data they point to.

Types of Smart pointers:



- **Box<T>**: The Box<T> is a smart pointer which points to the data allocated on the heap of type T where 'T' is the type of the data. It is used to store the data on the heap rather than on the stack.
- **Deref<T>**: The Deref<T> is a smart pointer which is used to customize the behavior of the dereference operator(*).
- **Drop<T>**: The Drop<T> is a smart pointer used to free the space from the heap memory when the variable goes out of the scope.
- **Rc<T>**: The Rc<T> stands for reference counted pointer. It is a smart pointer which keeps a record of the number of references to a value stored on the heap.
- **RefCell<T>**: The RefCell<T> is a smart pointer which allows you to borrow the mutable data even if the data is immutable. This process is known as interior mutability.

Box<T>

- Box<T> is a smart pointer that points to the data which is allocated on the heap of type T. Box<T> allow you to store the data on the heap rather than the stack.
- Box<T> is an owned pointer.
- Boxes do not have a performance overhead, other than storing the data on the heap.
- When the Box goes out of the scope, then the destructor is called to destroy all the inner objects and release the memory.

Using Box<T> to store the data on the heap.

Mainly, Box<T> is used to store the data on the heap. **Let's understand this through a simple example:**

```
fn main()
{
 let a = Box :: new(1);
 print!("value of a is : {}",a);
}
```

Output:

```
value of a is : 1
```

In the above example, a contains the value of Box that points to the data 1. If we access the value of Box, then the program prints '1'. When the program ends, then the Box is deallocated. The box is stored on the stack, and the data that it points to is stored on the heap.

Cons List

- Cons stand for **"Construct function"**.
- Cons list is a data structure which is used to construct a new pair from the two arguments, and this pair is known as a List.
- Suppose we have two elements x and y, then the cons function cons the "x onto y" means that we construct the new container by putting the element x first , and then followed by the element y.
- Cons list contains two elements, i.e., the current item and the last item. The last item of the cons list is Nil as Nil does not contain the next item.

Now, we create the enum that contains the cons list.

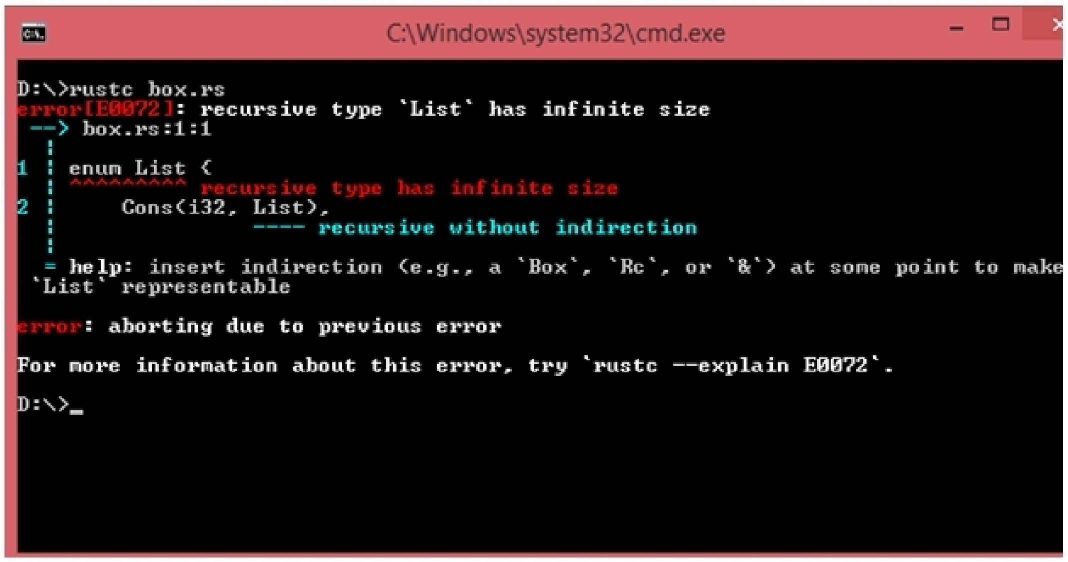
```
enum List
{
  cons(i32, List),
  Nil,
}
```

In the above code, we create the enum of List type which contains the cons list data structure of i32 values.

Now, we use the above List type in the following example:

```
enum List {
   Cons(i32, List),
   Nil,
}
use List::{Cons, Nil};
fn main()
{
 let list = List::Cons(1,Cons(2,Cons(3,Nil)));
 for i in list.iter()
 {
   print!("{}",i);
 }
}
```

Output:



```
D:\>rustc box.rs
error[E0072]: recursive type `List` has infinite size
 --> box.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |               ---- recursive without indirection
  |
  = help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to make
 `List` representable

error: aborting due to previous error

For more information about this error, try `rustc --explain E0072`.

D:\>_
```

In the above example, the Rust compiler throws an error "has infinite size" as List type contains the variant which is recursive. As a result, Rust is not able to find out how much space is required to store the List value. The problem of an infinite size can be overcome by using the Box<T>.

Using Box<T> to get the size of a recursive type

Rust cannot figure out how much space is required to store the recursive data types. The Rust compiler shows the error in the previous case:

```
= help: insert indirection (e.g., a 'Box', 'Rc', or '&') at some point to make 'List' representable
```

In the above case, we can use the Box<T> pointer as the compiler knows how much space Box<T> pointer requires. The size of Box<T> pointer will not change during the execution of a program. The Box<T> pointer points to the List value that will be stored on the heap rather than in the cons variant. The Box<T> pointer can be placed directly in the cons variant.

Let's see a simple example:

```rust
#[derive(Debug)]
enum List {
   Cons(i32, Box<List>),
   Nil,
}
use List::{Cons, Nil};
fn main()
{
 let list = Cons(1,Box::new(Cons(2,Box::new(Cons(3,Box::new(Nil))))));

   print!("{:?}",list);

}
```

Output:

```
Cons(1, Cons(2, Cons(3, Nil)))
```

Deref<T>

- Deref<T> trait is used to customize the behavior of dereference operator (*).
- If we implement the Deref<T> trait, then the smart pointer can be treated as a reference. Therefore, the code that works on the references can also be used on the smart pointers too.

Regular References

Regular reference is a kind of pointer that points to some value which is stored somewhere else. Let's see a simple example to create the reference of i32 type value and then we use the dereference operator with this reference.

```
fn main()
{
  let a = 20;
  let b = &a;
  if a==*b
  {
    println!("a and *b are equal");
  }

  else
  {
    println!("they are not equal");
  }
}
```

Output:

```
a and *b are equal
```

In the above example, a holds the i32 type value, 20 while b contains the reference of 'a' variable. If we use *b, then it represents the value, 20. Therefore, we can compare the variable a and *b, and it will return the true value. If we use &b instead of *b, then the compiler throws an error **"cannot compare {integer} with {&integer}"**.

Box<T> as a Reference

The Box<T> pointer can be used as a reference.

Let's see a simple example:

```
fn main()
{
  let a = 11;
  let b = Box::new(a);
  print!("Value of *b is {}",*b);
}
```

Output:

```
Value of *b is 11
```

In the above example, Box<T> behaves similarly as the regular references. The only difference between them is that b contains the box pointing to the data rather than referring to the value by using the '&' operator.

Smart Pointer as References

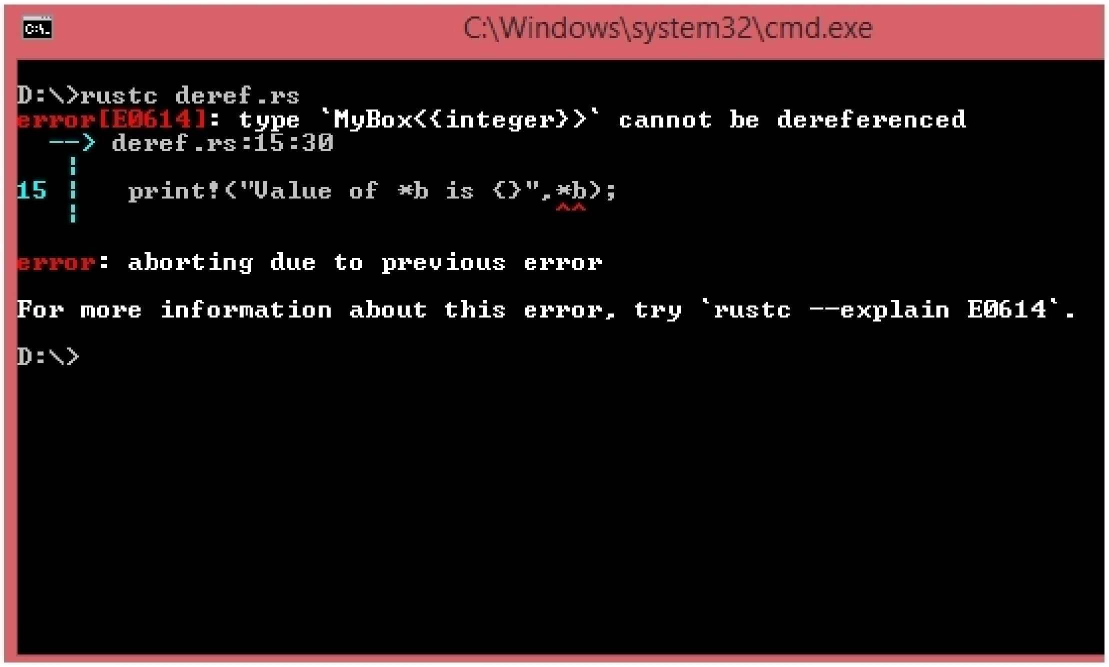
Now, we create the smart pointer similar to the Box<T> type, and we will see how they behave differently from the regular references.

- The Box<T> can be defined as the tuple struct with one element for example, MyBox<T>.
- After creating the tuple struct, we define the function on type MyBox<T>.

Let's see a simple example:

```
struct MyBox<T>(T);
impl<T> MyBox<T>
{
  fn example(y : T)->MyBox<T>
  {
    MyBox(y)
  }
}
fn main()
{
  let a = 8;
  let b = MyBox::example(a);
  print!("Value of *b is {}",*b);
}
```

Output:



```
D:\>rustc deref.rs
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
  --> deref.rs:15:30
   |
15 |    print!("Value of *b is {}",*b);
   |                               ^^

error: aborting due to previous error

For more information about this error, try `rustc --explain E0614`.

D:\>
```

In the above example, we create the smart pointer, b, but it cannot be dereferenced. Therefore, we conclude that the customized pointers which are similar to the Box<T> type cannot be dereferenced.

Implementing a Deref Trait

- The Deref trait is defined in the standard library which is used to implement the method named deref.
- The deref method borrows the self and returns a reference to the inner data.

Let's see a simple example:

```
struct MyBox<T>
{
 a : T,
}
use :: std::ops::Deref;
impl<T> Deref for MyBox<T>
{
 type Target = T;
 fn deref(&self) ->&T
 {
  &self.a
 }
}
fn main()
{
 let b = MyBox{a : 10};
 print!("{}",*(b.deref()));
}
```

Output:

```
10
```

Program Explanation

- The Deref trait is implemented on the MyBox type.
- The Deref trait implements the deref() method, and the deref() method returns the reference of 'a' variable.
- The type Target = T; is an associated type for a Deref trait. Associated type is used to declare the generic type parameter.

- We create the instance of MyBox type, b.
- The deref() method is called by using the instance of MyBox type, b.deref() and then the reference which is returned from the deref() method is dereferenced.

Deref Coercion

- Deref Coercion is a process of converting the reference that implements the Deref trait into the reference that Deref can convert the original type into.
- Deref Coercion is performed on the arguments of the functions and methods.
- Deref Coercion happens automatically when we pass the reference of a particular type to a function that does not match with the type of an argument in the function definition.

Let's see a simple example:

```
 struct MyBox<T>(T);
use :: std::ops::Deref;
impl<T> MyBox<T>
{
  fn hello(x:T)->MyBox<T>
  {
    MyBox(x)
  }
}
impl<T> Deref for MyBox<T>
{
  type Target = T;
  fn deref(&self) ->&T
  {
    &self.0
  }
}
fn print(m : &i32)
{
  print!("{}",m);
}
fn main()
{
  let b = MyBox::hello(5);

  print(&b);
}
```

Output:

```
5
```

In the above example, we are calling the print(&b) function with an argument &b, which is the reference of &Box<i32>. In this case, we implement the Deref trait that converts the &Box<i32> into &i32 through the process of Deref Coercion.

Interaction of Derif Coercion with mutability

Till now, we use the Deref Trait to override the * operator on immutable references, and we can use the DerefMut trait to override the * operator on mutable references.

Rust performs Deref coercion in the following three cases:

- When T: Deref<Target = U> where T and U are the immutable references , then &T is converted into &U type.
- When T: DerefMut<Target = U> where T and U are the mutable references, then &mut T is converted into &mut U.
- When T: Deref<Target = U> where T is a mutable reference and U is an immutable reference, then &mut T is converted into &U.

Drop trait

- Drop trait is used to release the resources like files or network connections when the value goes out of the scope.
- Drop trait is used to deallocate the space on the heap that the Box<T> points to.
- The drop trait is used to implement the drop() method that takes a mutable reference to the self.

Let's see a simple example:

```
struct Example
{
 a : i32,
 }
     impl Drop for Example
{
 fn drop(&mut self)
 {
  println!("Dropping the instance of Example with data : {}", self.a);
 }
}
     fn main()
{
 let a1 = Example{a : 10};
 let b1 = Example{a: 20};
 println!("Instances of Example type are created");
}
```

Output:

```
Instances of Example type are created
Dropping the instance of Example with data : 20
Dropping the instance of Example with data : 10
```

Program Explanation

- We have implemented the Drop trait on the type Example, and we define the drop() method inside the implementation of the Drop trait.
- Inside the main() function, we create the instances of the type Example and at the end of the main() function, instances go out of the scope.
- When the instances move out of the scope, then Rust calls the drop() method implicitly to drop the instances of type Example. First, it will drop the b1 instance and then a1 instance.

Dropping a value early with std::mem::drop

Sometimes, it becomes necessary to drop the value before the end of the scope. If we want to drop the value early, then we use the std::mem::drop function to drop the value.

Let's see a simple example to drop the value manually:

```
struct Example
{
 a : String,
}
impl Drop for Example
{
 fn drop(&mut self)
 {
  println!("Dropping the instance of Example with data : {}", self.a);
 }
}
fn main()
{
 let a1 = Example{a : String::from("Hello")};
 a1.drop();
 let b1 = Example{a: String::from("World")};
 println!("Instances of Example type are created");
}
```

In the above example, we call the drop() method manually. The Rust compiler throws an error that we are not allowed to call the drop() method explicitly. Instead of calling the drop() method explicitly, we call the std::mem::drop function to drop the value before it goes out of the scope.

- The syntax of std::mem::drop function is different from the drop() function defined in the Drop trait. The std::mem::drop function contains the value passed as an argument which is to be dropped before it goes out of the scope.

Let's see a simple example:

```
struct Example
{
  a : String,
}

impl Drop for Example
{
  fn drop(&mut self)
  {
    println!("Dropping the instance of Example with data : {}", self.a);
  }
}

fn main()
{
  let a1 = Example{a : String::from("Hello")};
  drop(a1);
  let b1 = Example{a: String::from("World")};
  println!("Instances of Example type are created");
}
```

Output:

```
Dropping the instance of Example with data : Hello
Instances of Example type are created
Dropping the instance of Example with data : World
```

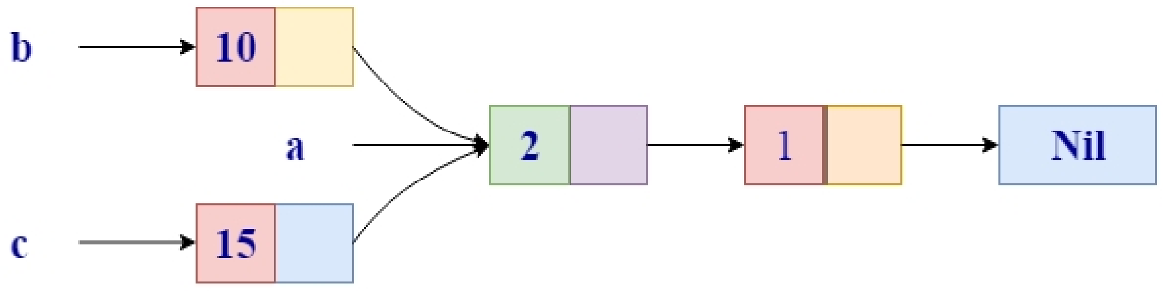
In the above example, the a1 instance is destroyed by passing the a1 instance as an argument in the drop(a1) function.

Rc<T>

- The Rc<T> stands for Reference Counted Smart Pointer.
- The Rc<T> smart pointer keeps track of the number of references to a value to determine whether the value is still in use or not and if there are zero references to a value, then the value can be cleaned up.
- The Rc<T> smart pointer is a single threaded reference-counting pointer.

Using Rc<T> to share data

Let's create the two lists that share the ownership of a third list.



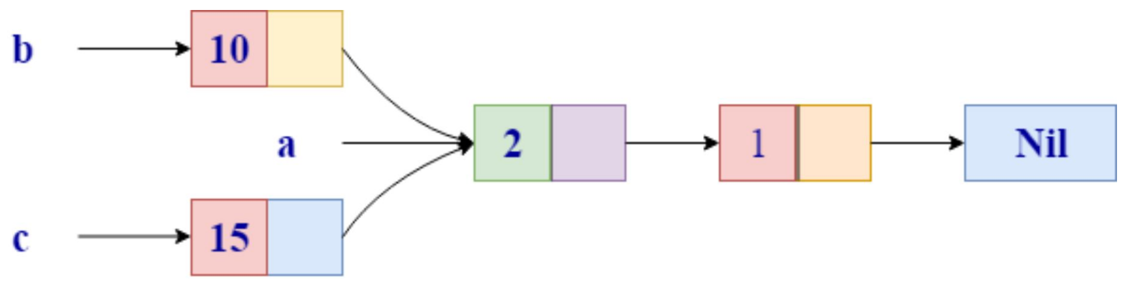
next →← prev

Rc<T>

- The Rc<T> stands for Reference Counted Smart Pointer.
- The Rc<T> smart pointer keeps track of the number of references to a value to determine whether the value is still in use or not and if there are zero references to a value, then the value can be cleaned up.
- The Rc<T> smart pointer is a single threaded reference-counting pointer.

Using Rc<T> to share data

Let's create the two lists that share the ownership of a third list.

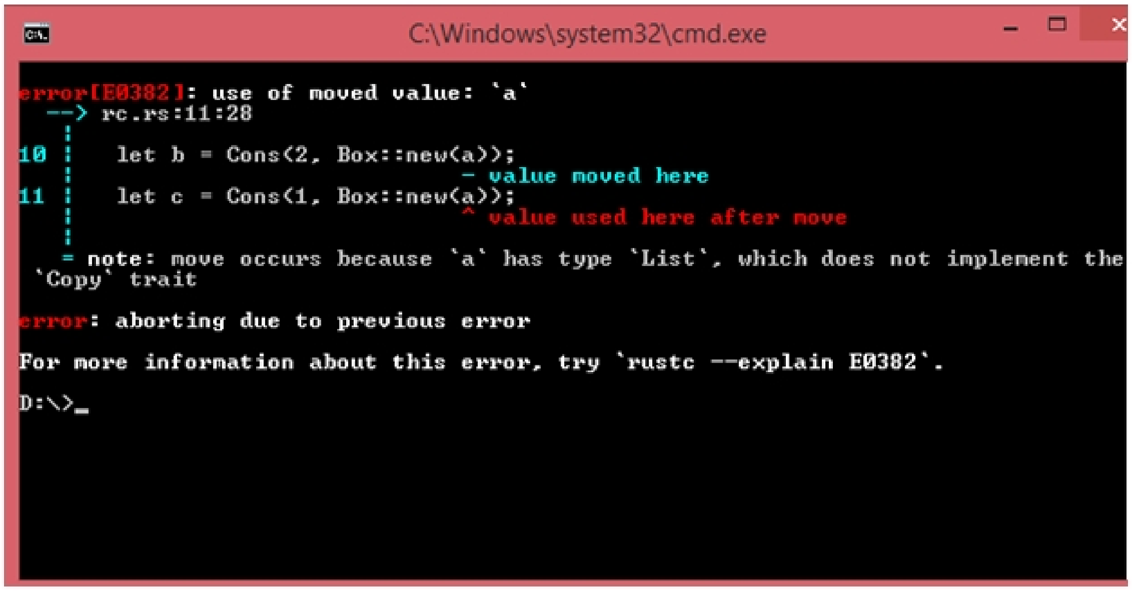


In the above figure, b and c are the two lists that share the ownership to the third list,a.

Let's implement the above scenario using Box<T> type.

```
enum List
{
  Cons(i32, Box<List>),
  Nil,
}
use List::{Cons,Nil};
fn main()
{
  let a = Cons(10, Box::new(Cons(15,Box::new(Nil))));
  let b = Cons(2, Box::new(a));
  let c = Cons(1, Box::new(a));
}
```

Output:



```
error[E0382]: use of moved value: `a`
  --> rc.rs:11:28
   |
10 |    let b = Cons(2, Box::new(a));
   |                             - value moved here
11 |    let c = Cons(1, Box::new(a));
   |                             ^ value used here after move
   |
   = note: move occurs because `a` has type `List`, which does not implement the
 `Copy` trait

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.

D:\>_
```

In the above example, cons variant consists of a data of type i32 and Box<T> pointing to a list. We create the list 'b' and the ownership of 'a' is moved to the 'b' list. Then, we try to move the' a' list to the 'c' list, but a list cannot be moved as 'a' list is already moved to the 'b' list.

How to overcome this problem

We can overcome this problem by changing the definition of the cons variant. Now, cons variant consists of a data that they hold and Rc<T> pointing to the List.

Let's see a simple example:

```
 enum List
{
  Cons(i32, Rc<List>),
  Nil,
}
use List::{Cons,Nil};
use std::rc::Rc;
fn main()
{
  let a = Rc::new(Cons(10, Rc::new(Cons(15,Rc::new(Nil)))));
  let b = Cons(2, Rc::clone(&a));
  let c = Cons(1, Rc::clone(&a));
}
```

In the above example, we need to add the use statement to bring the Rc<T> into the scope. Instead of taking the ownership of a, we will clone the Rc<T> list that a is holding and, therefore increasing the number of references from one to two as now, a and b are sharing the ownership of the data in that Rc<List>. We will again clone the Rc<List> when creating the c List, therefore increasing the references from two to three.

Cloning an Rc<T> Increases the Reference Count

Now, we will see how Rc<T> increases or drops the reference count when the list goes out of the scope.

Let's see a simple example:

```
enum List
{
 Cons(i32, Rc<List>),
 Nil,
}
use List::{Cons,Nil};
use std::rc::Rc;
fn main()
{
 let a = Rc::new(Cons(10, Rc::new(Cons(15,Rc::new(Nil)))));
 println!
("Reference count after creating a List : {}", Rc::strong_count(&a));
 let b = Cons(2, Rc::clone(&a));
 println!
("Reference count after creating b List : {}", Rc::strong_count(&a));
 {
 let c = Cons(1, Rc::clone(&a));
 println!
("Reference count after creating c List : {}",Rc::strong_count(&a));
 }
 println!
("Reference count when c goes out of the scope : {}",Rc::strong_count(
&a));
}
```

Output:

```
Reference count after creating a List : 1
Reference count after creating b List : 2
Reference count after creating c List : 3
Reference count when c goes out of the scope : 2
```

In the above example, we print the reference count by calling the **Rc::strong_count function**. An initial reference count of a in Rc<List> is 1 and when we call clone, then the reference count increases by 1. If variable goes out of the scope, then the reference count decreases by 1. Therefore, we can say that the **Drop trait** automatically decreases the reference count when an **Rc<T>/value goes out of the scope.**

RefCell<T>

Interior mutability pattern is a pattern is used to mutate the reference if we have an immutable reference. RefCell<T> can be used to achieve the interior mutability.

Important Points to remember:

- RefCell<T> represents the single ownership over the data that it holds.
- If we use RefCell<T>, then the invariants are enforced at the runtime.
- RefCell<T> is mainly used in the single-threaded scenario and will give an error if we use in a multithreaded case.
- RefCell<T> checks the mutable borrows at the runtime. Therefore, we can say that we can mutate the value even when the RefCell<T> value is immutable.

Interior Mutability

According to the borrowing rules, if we have an immutable value, then we cannot borrow mutably.

Let's see a simple example:

```
fn main()
{
 let a = 15;
 let b = &mut a;
}
```

In the above example, we have seen that the immutable value cannot be borrowed mutably. But, RefCell is the one way to achieve the interior mutability.

Keeping Track of Borrows at Runtime with RefCell<T>

RefCell<T> consists of two methods that keep track of borrows at runtime:

- **borrow():** The borrow() method returns the smart pointer of type Ref<T>.
- **borrow_mut():** The borrow_mut() method returns the smart pointer of type RefMut<<T>.

Some Important Points:

- The **RefCell<T>** keeps a record of how many Ref<T> and Refmut<T> smart pointers are currently active.
- Whenever the **borrow()** method is called, then the RefCell<T> increases the count of how many immutable borrows are active. When the Rc<T> goes out of the scope, then RefCell<T> decreases the count by one.
- The **RefCell<T>** lets us have many immutable borrows but one mutable borrow at a time, just as compile-time borrowing rules. If we violate this rule, then the RefCell<T> will panic at runtime.

borrow() method

The borrow() method borrows the immutable value. Multiple immutable borrows can be taken at the same time.

Syntax:

```
pub fn borrow(&self) -> Ref<T>
```

Let's see a simple example when multiple immutable borrow occurs:

```
 use std::cell::RefCell;
fn main()
{
  let a = RefCell::new(15);
  let b = a.borrow();
  let c = a.borrow();
  println!("Value of b is : {}",b);
  println!("Value of c is : {}",c);
}
```

Let's see a simple example of panic condition:

```
 use std::cell::RefCell;
fn main()
{
 let a = RefCell::new(10);
 let b = a.borrow();
 let c = a.borrow_mut(); // cause panic.
 println!("Value of b is : {}",b);
 println!("Value of c is : {}",c);
}
```

In the above example, program panics at runtime as immutable borrows and mutable borrows cannot occur at the same time.

borrow_mut() method

The borrow_mut() method borrows the mutable value. Mutable borrows can occur once.

Syntax:

```
pub fn borrow_mut(&self) -> RefMut<T>;
```

Let's see a simple example:

```
 use std::cell::RefCell;
fn main()
{
 let a = RefCell::new(15);
 let b = a.borrow_mut();
 println!("Now, value of b is {}",b);
}
```

Multiple owners of Mutable Data By combining Rc<T> and RefCell<T>

We can combine Rc<T> and RefCell<T> so that we can have multiple owners of mutable data. The Rc<T> lets you have multiple owners of a data, but it provides only immutable access to the data. The RefCell<T> lets you to mutate the data. Therefore, we can say that the combination of Rc<T> and RefCell<T> provides the flexibility of having multiple owners with mutable data.

Let's see a simple example:

```rust
 #[derive(Debug)]
enum List
{
 Cons(Rc<RefCell<String>>,Rc<List>),
 Nil,
}
use List:: {Cons,Nil};
use std::rc::Rc;
use std::cell::RefCell;
fn main()
{
  let val = Rc::new(RefCell::new(String::from("java")));
  let a = Rc::new(Cons(Rc::clone(&val),Rc::new(Nil)));
  let b = Cons(Rc::new(RefCell::new(String::from("C"))),Rc::clone(&a))
;
  let c = Cons(Rc::new(RefCell::new(String::from("C++"))),Rc::clone(&
a));
  *val.borrow_mut() = String::from("C# language");
  println!("value of a is : {:?}",a);
  println!("value of b is : {:?}",b);
  println!("value of c is : {:?}",c);
}
```

In the above example, we create a variable 'val' and store the value "java" to the variable 'val'. Then, we create the list 'a' and we clone the 'val' variable so that both the variable 'a' and 'val' have the ownership of 'java' value rather than transferring the ownership from 'val' to 'a' variable. After creating 'a' list, we create 'b' and 'c' list and clones the 'a' list. After creating the lists, we replace the value of 'val' variable with "C#" language" by using the borrow_mut() method.

GOLANG
FOR
BEGINNERS

LEARN TO CODE FAST

BY

TAM SEL

GO FOR BEGINNERS

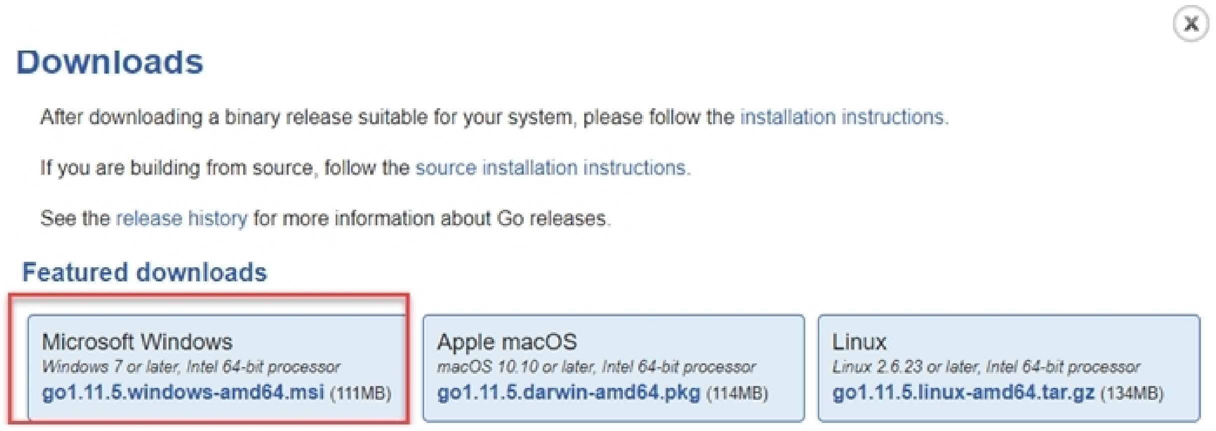
Go Programming Language

What is Go?

Go (also known as Golang) is an open source programming language developed by Google. It is a statically-typed compiled language. Go supports concurrent programming, i.e. it allows running multiple processes simultaneously. This is achieved using channels, goroutines, etc. Go has garbage collection which itself does the memory management and allows the deferred execution of functions.

How to Download and install GO

Step 1) Go to https://golang.org/dl/. Download the binary for your OS.



Downloads

After downloading a binary release suitable for your system, please follow the installation instructions.

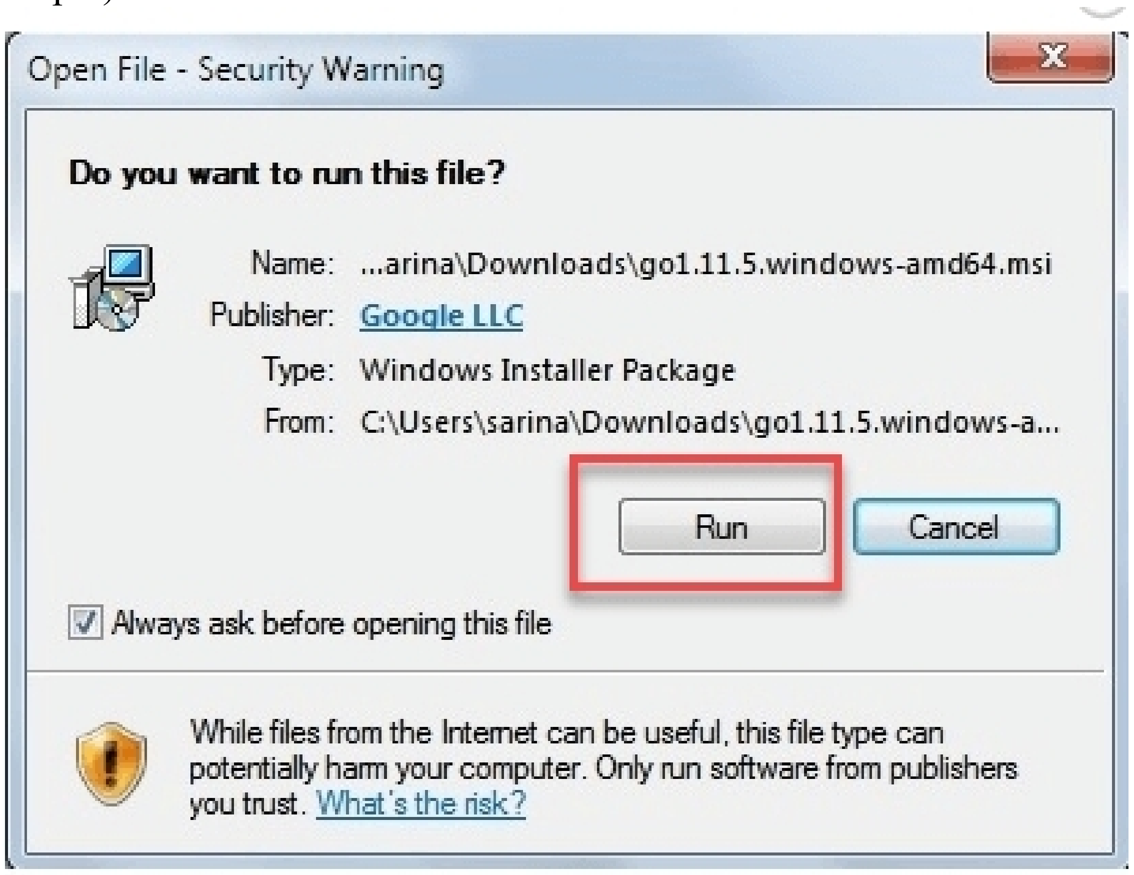
If you are building from source, follow the source installation instructions.

See the release history for more information about Go releases.

Featured downloads

Microsoft Windows Windows 7 or later, Intel 64-bit processor go1.11.5.windows-amd64.msi (111MB)	Apple macOS macOS 10.10 or later, Intel 64-bit processor go1.11.5.darwin-amd64.pkg (114MB)	Linux Linux 2.6.23 or later, Intel 64-bit processor go1.11.5.linux-amd64.tar.gz (134MB)

Step 2) Double click on the installer and click Run.



Open File - Security Warning

Do you want to run this file?

Name: ...arina\Downloads\go1.11.5.windows-amd64.msi

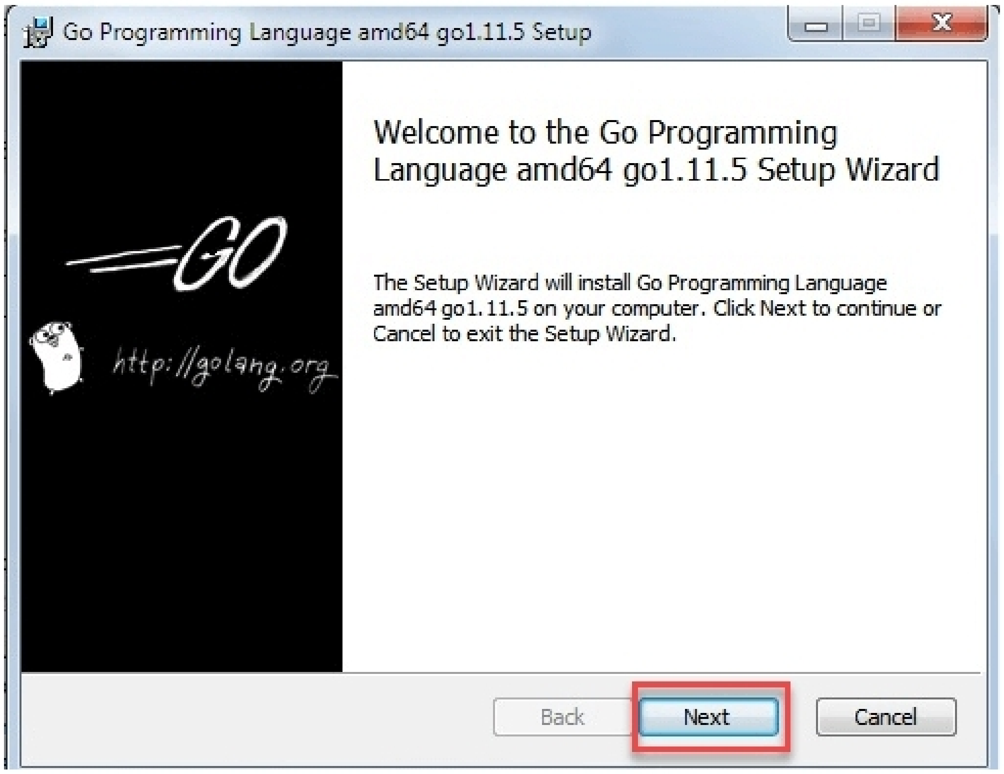
Publisher: Google LLC

Type: Windows Installer Package

From: C:\Users\sarina\Downloads\go1.11.5.windows-a...

While files from the Internet can be useful, this file type can potentially harm your computer. Only run software from publishers you trust. What's the risk?

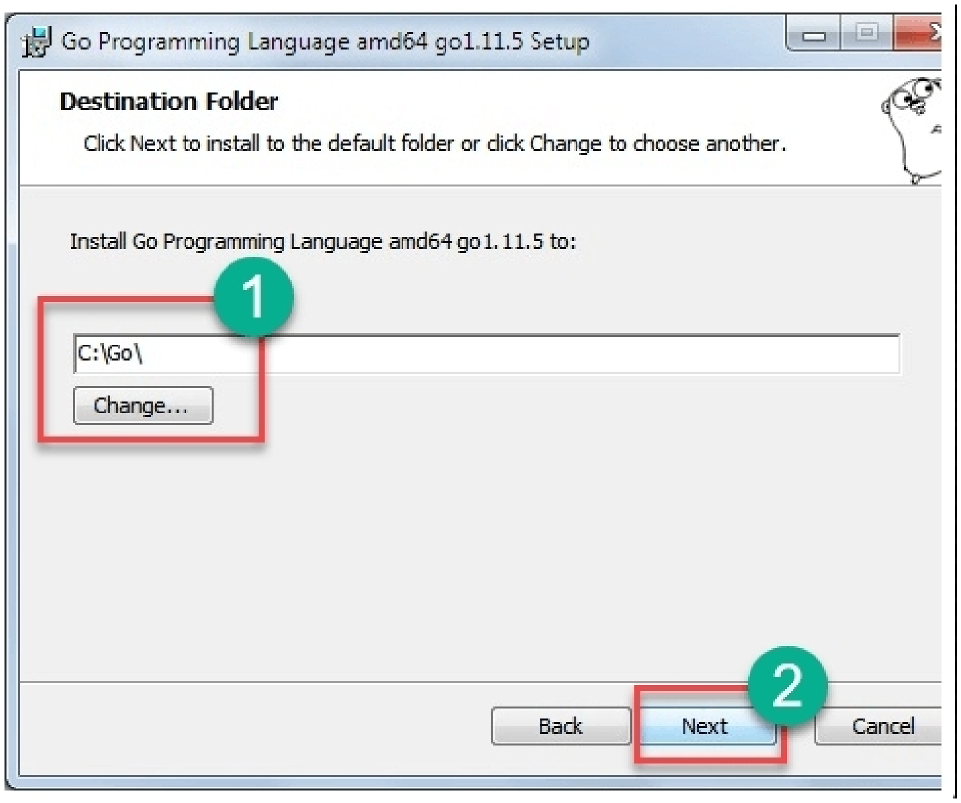
Step 3) Click Next


Go Programming Language amd64 go1.11.5 Setup

Welcome to the Go Programming Language amd64 go1.11.5 Setup Wizard

The Setup Wizard will install Go Programming Language amd64 go1.11.5 on your computer. Click Next to continue or Cancel to exit the Setup Wizard.

Step 4) Select the installation folder and click Next.


Go Programming Language amd64 go1.11.5 Setup

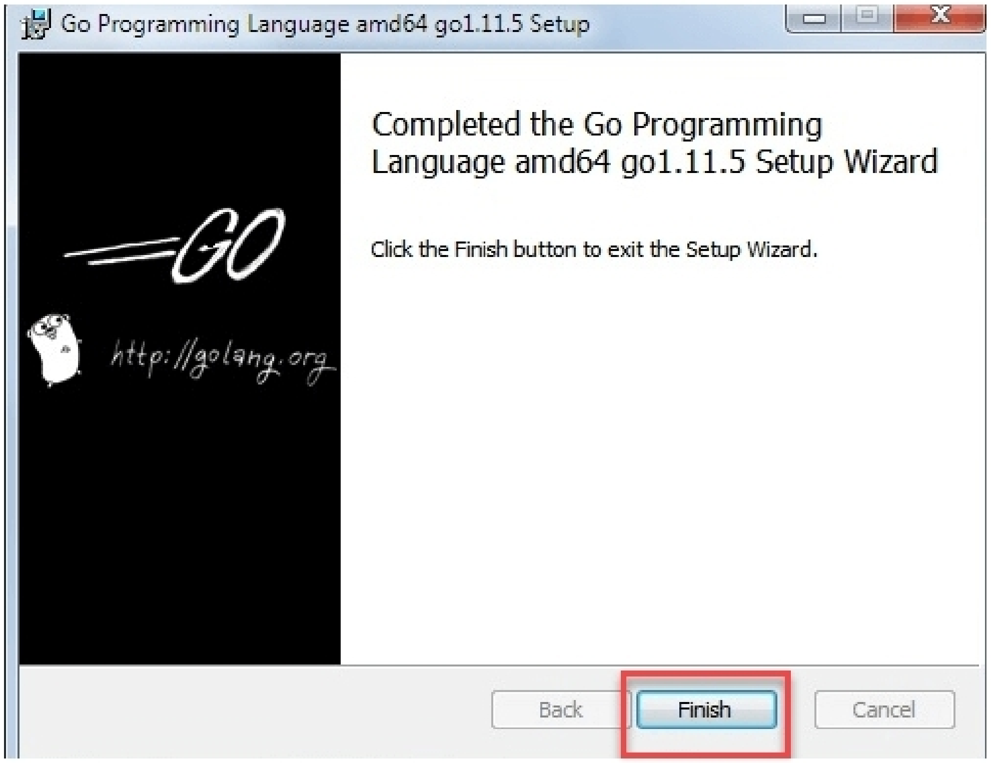
Destination Folder

Click Next to install to the default folder or click Change to choose another.

Install Go Programming Language amd64 go1.11.5 to:

C:\Go\

Step 5) Click Finish once the installation is complete.


Go Programming Language amd64 go1.11.5 Setup

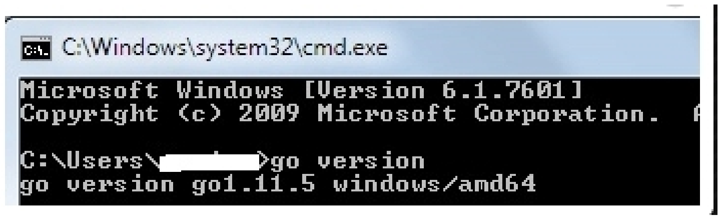
Completed the Go Programming Language amd64 go1.11.5 Setup Wizard

Click the Finish button to exit the Setup Wizard.

Step 6) Once the installation is complete you can verify it by opening the terminal and typing

go version

This will display the version of go installed



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  A

C:\Users\[redacted]>go version
go version go1.11.5 windows/amd64
```

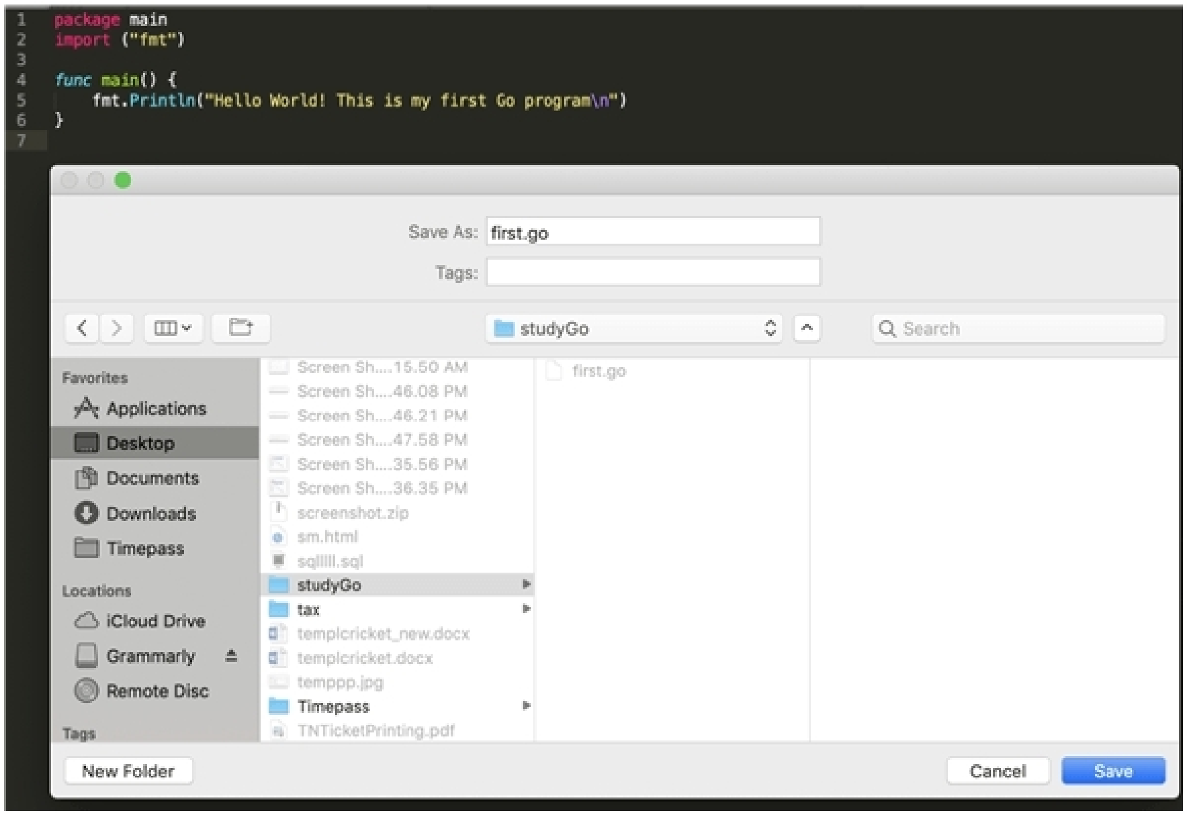
Your First Go program

Create a folder called studyGo. You will create our go programs inside this folder. Go files are created with the extension .go. You can run Go programs using the syntax

```
go run <filename>
```

Create a file called first.go and add the below code into it and save

```
package main
import ("fmt")
func main() {
        fmt.Println("Hello World! This is my first Go program\n")
}
```



```
package main
import ("fmt")

func main() {
    fmt.Println("Hello World! This is my first Go program\n")
}
```

Save As: first.go

Tags:

Navigate to this folder in your terminal. Run the program using the command

```
go run first.go
```

You can see the output printing

```
Hello World! This is my first Go program
```

Now let's discuss the above program.

package main - Every go program should start with a package name. Go allows us to use packages in another go programs and hence supports code reusability. Execution of a go program begins with the code inside the package named main.

import fmt - imports the package fmt. This package implements the I/O functions.

func main() - This is the function from which program execution begins. The main function should always be placed in the main package. Under the main(), You can write the code inside { }.

fmt.Println - This will print the text on the screen by the Println function of fmt.

Note: In the below sections when You mention execute/run the code, it means to save the code in a file with .go extension and run it using the syntax

```
go run <filename>
```

Data Types

Types(data types) represent the type of the value stored in a variable, type of the value a function returns, etc.
There are three basic types in Go.
Numeric types - Represent numeric values which includes integer, floating point, and complex values. Various numeric types are:
int8 - 8 bit signed integers.
int16 - 16 bit signed integers.
int32 - 32 bit signed integers.
int64 - 64 bit signed integers.
uint8 - 8 bit unsigned integers.
uint16 - 16 bit unsigned integers.
uint32 - 32 bit unsigned integers.
uint64 - 64 bit unsigned integers.
float32 - 32 bit floating point numbers.
float64 - 64 bit floating point numbers.
complex64 – has float32 real and imaginary parts.
complex128 - has float32 real and imaginary parts.
String types - Represents a sequence of bytes(characters). You can do various operations on strings like string concatenation, extracting substring, etc
Boolean types - Represents 2 values, either true or false.

Variables

Variables point to a memory location which stores some kind of value. The type parameter(in the below syntax) represents the type of value that can be stored in the memory location.
Variable can be declared using the syntax

```
  var <variable_name> <type>
```

Once You declare a variable of a type You can assign the variable to any value of that type.
You can also give an initial value to a variable during the declaration itself using

```
   var <variable_name> <type> = <value>
```

If You declare the variable with an initial value, Go an infer the type of the variable from the type of value assigned. So You can omit the type during the declaration using the syntax

```
   var <variable_name> = <value>
```

Also, You can declare multiple variables with the syntax

```
  var <variable_name1>, <variable_name2>  = <value1>, <value2>
```

The program below has some examples of variable declarations

```
package main
import "fmt"
func main() {
    //declaring a integer variable x
    var x int
    x=3 //assigning x the value 3
    fmt.Println("x:", x) //prints 3

    //declaring a integer variable y with value 20 in a single
statement and prints it
    var y int=20
    fmt.Println("y:", y)

    //declaring a variable z with value 50 and prints it
    //Here type int is not explicitly mentioned
    var z=50
    fmt.Println("z:", z)

    //Multiple variables are assigned in single line- i with an integer
and j with a string
    var i, j = 100,"hello"
    fmt.Println("i and j:", i,j)
}
```

The output will be

```
x: 3
y: 20
z: 50
i and j: 100 hello
```

Go also provides an easy way of declaring the variables with value by omitting the var keyword using

```
  <variable_name> := <value>
```

Note that You used := instead of =. You cannot use := just to assign a value to a variable which is already declared. := is used to declare and assign value.
Create a file called assign.go with the following code

```
package main
import ("fmt")
func main() {
        a := 20
        fmt.Println(a)
        //gives error since a is already declared
```

```
    a := 30
    fmt.Println(a)
}
```

Execute go run assign.go to see the result as

```
./assign.go:7:4: no new variables on left side of :=
```

Variables declared without an initial value will have of 0 for numeric types, false for Boolean and empty string for strings

Constants

Constant variables are those variables whose value cannot be changed once assigned. A constant in Go is declared by using the keyword "const"
Create a file called constant.go and with the following code

```
package main
import ("fmt")
func main() {
        const b =10
        fmt.Println(b)
        b = 30
        fmt.Println(b)
}
```

Execute go run constant.go to see the result as

```
.constant.go:7:4: cannot assign to b
```

Loops

Loops are used to execute a block of statements repeatedly based on a condition. Most of the programming languages provide 3 types of loops - for, while, do while. But Go supports only for loop.
The syntax of a for loop is

```
for initialisation_expression; evaluation_expression;
iteration_expression{
   // one or more statement
}
```

The initialisation_expression is executed first(and only once).
Then the evaluation_expression is evaluated and if it's true the code inside the block is executed.
The iteration_expression id is executed, and the evaluation_expression is evaluated again. If it's true the statement block gets executed again. This will continue until the evaluation_expression becomes false.
Copy the below program into a file and execute it to see the for loop printing numbers from 1 to 5

```
package main
import "fmt"
func main() {
var i int
for i = 1; i <= 5; i++ {
fmt.Println(i)
    }
}
```

Output is

```
1
```

If else

If else is a conditional statement. The syntax is

```
if condition{
// statements_1
}else{
// statements_2
}
```

Here the condition is evaluated and if it's true statements_1 will be executed else statements_2 will be executed.
You can use if statement without else also. You also can have chained if else statements. The below programs will explain more about if else.
Execute the below program. It checks if a number, x, is less than 10. If so, it will print "x is less than 10"

```
package main
import "fmt"
func main() {
   var x = 50
   if x < 10 {
      //Executes if x < 10
      fmt.Println("x is less than 10")
   }
}
```

Here since the value of x is greater than 10, the statement inside if block condition will not executed.
Now see the below program. We have an else block which will get executed on the failure of if evaluation.

```
package main
import "fmt"
func main() {
   var x = 50
   if x < 10 {
      //Executes if x is less than 10
      fmt.Println("x is less than 10")
   } else {
      //Executes if x >= 10
      fmt.Println("x is greater than or equals 10")
   }
}
```

This program will give you output

```
x is greater than or equals 10
```

Now we will see a program with multiple if else blocks(chained if else).Execute the below example. It checks whether a number is less than 10 or is between 10-90 or greater than 90.

```
package main
import "fmt"
func main() {
   var x = 100
   if x < 10 {
      //Executes if x is less than 10
      fmt.Println("x is less than 10")
   } else if x >= 10 && x <= 90 {
        //Executes if x >= 10 and x<=90
        fmt.Println("x is between 10 and 90")
    } else {
        //Executes if both above cases fail i.e x>90
        fmt.Println("x is greater than 90")
    }
}
```

Here first the if condition checks whether x is less than 10 and it's not. So it checks the next condition(else if) whether it's between 10 and 90 which is also false. So it then executes the block under the else section which gives the output

```
x is greater than 90
```

Switch

Switch is another conditional statement. Switch statements evaluate an expression and the result is compared against a set of available values(cases). Once a match is found the statements associated with that match(case) is executed. If no match is found nothing will be executed. You can also add a default case to switch which will be executed if no other matches are found. The syntax of the switch is

```
switch expression {
    case value_1:
        statements_1
    case value_2:
        statements_2
    case value_n:
        statements_n
    default:
        statements_default
    }
```

Here the value of the expression is compared against the values in each case. Once a match is found the statements associated with that case is executed. If no match is found the statements under the default section is executed.

Execute the below program

```
package main
import "fmt"
func main() {
    a,b := 2,1
    switch a+b {
```

```
    case 1:
        fmt.Println("Sum is 1")
    case 2:
        fmt.Println("Sum is 2")
    case 3:
        fmt.Println("Sum is 3")
    default:
        fmt.Println("Printing default")
    }
}
```

You will get the output as

```
Sum is 3
```

Change the value of a and b to 3 and the result will be

```
Printing default
```

You can also have multiple values in a case by separating them with a comma.

Arrays

Array represents a fixed size, named sequence of elements of the same type. You cannot have an array which contains both integer and characters in it. You cannot change the size of an array once You define the size.
The syntax for declaring an array is

```
var arrayname [size] type
```

Each array element can be assigned value using the syntax

```
arrayname [index] = value
```

Array index starts from 0 to size-1.
You can assign values to array elements during declaration using the syntax

```
arrayname := [size] type {value_0,value_1,…,value_size-1}
```

You can also ignore the size parameter while declaring the array with values by replacing size with … and the compiler will find the length from the number of values. Syntax is

```
arrayname :=  […] type {value_0,value_1,…,value_size-1}
```

You can find the length of the array by using the syntax

```
len(arrayname)
```

Execute the below example to understand the array

```
package main
import "fmt"
func main() {
   var numbers [3] string //Declaring a string array of size 3 and
adding elements
   numbers[0] = "One"
   numbers[1] = "Two"
   numbers[2] = "Three"
   fmt.Println(numbers[1]) //prints Two
   fmt.Println(len(numbers)) //prints 3
   fmt.Println(numbers) // prints [One Two Three]
```

```
    directions := [...] int {1,2,3,4,5} // creating an integer array and
the size of the array is defined by the number of elements
    fmt.Println(directions) //prints [1 2 3 4 5]
    fmt.Println(len(directions)) //prints 5
    //Executing the below commented statement prints invalid array
index 5 (out of bounds for 5-element array)
    //fmt.Println(directions[5])
}
```

Output

```
Two
3
[One Two Three]
[1 2 3 4 5]
5
```

Slice

A slice is a portion or segment of an array. Or it is a view or partial view of an underlying array to which it points. You can access the elements of a slice using the slice name and index number just as you do in an array. You cannot change the length of an array, but you can change the size of a slice. Contents of a slice are actually the pointers to the elements of an array. It means if you change any element in a slice, the underlying array contents also will be affected.

The syntax for creating a slice is

```
var slice_name [] type = array_name[start:end]
```

This will create a slice named slice_name from an array named array_name with the elements at the index start to end-1.

Execute the below program. The program will create a slice from the array and print it. Also, you can see that modifying the contents in the slice will modify the actual array.

```
package main
import "fmt"
func main() {
    // declaring array
    a := [5] string {"one", "two", "three", "four", "five"}
    fmt.Println("Array after creation:",a)
    var b [] string = a[1:4] //created a slice named b
    fmt.Println("Slice after creation:",b)
    b[0]="changed" // changed the slice data
    fmt.Println("Slice after modifying:",b)
    fmt.Println("Array after slice modification:",a)
}
```

This will print result as

```
Array after creation: [one two three four five]
Slice after creation: [two three four]
Slice after modifying: [changed three four]
Array after slice modification: [one changed three four five]
```

There are certain functions which you can apply on slices
len(slice_name) - returns the length of the slice
append(slice_name, value_1, value_2) - It is used to append value_1 and value_2 to an existing slice.
append(slice_nale1,slice_name2…) – appends slice_name2 to slice_name1
Execute the following program.

```
package main
import "fmt"
func main() {
        a := [5] string {"1","2","3","4","5"}
        slice_a := a[1:3]
        b := [5] string {"one","two","three","four","five"}
        slice_b := b[1:3]
   fmt.Println("Slice_a:", slice_a)
   fmt.Println("Slice_b:", slice_b)
   fmt.Println("Length of slice_a:", len(slice_a))
   fmt.Println("Length of slice_b:", len(slice_b))
   slice_a = append(slice_a,slice_b...) // appending slice
   fmt.Println("New Slice_a after appending slice_b :", slice_a)

   slice_a = append(slice_a,"text1") // appending value
   fmt.Println("New Slice_a after appending text1 :", slice_a)
}
```

The output will be

```
Slice_a: [2 3]
Slice_b: [two three]
Length of slice_a: 2
Length of slice_b: 2
New Slice_a after appending slice_b : [2 3 two three]
New Slice_a after appending text1 : [2 3 two three text1]
```

The program first creates 2 slices and printed its length. Then it appended one slice to other and then appended a string to the resulting slice.

Functions

A function represents a block of statements which performs a specific task. A function declaration tells us function name, return type and input parameters. Function definition represents the code contained in the function. The syntax for declaring the function is

```
func function_name(parameter_1 type, parameter_n type)
return_type {
//statements
}
```

The parameters and return types are optional. Also, you can return multiple values from a function.
Let's run the following example. Here function named calc will accept 2 numbers and performs the addition and subtraction and returns both values.

```
package main
import "fmt"
//calc is the function name which accepts two integers num1 and
num2
//(int, int) says that the function returns two values, both of integer
type.
func calc(num1 int, num2 int)(int, int) {
   sum := num1 + num2
   diff := num1 - num2
   return sum, diff
}
func main() {
   x,y := 15,10
   //calls the function calc with x and y an d gets sum, diff as output
   sum, diff := calc(x,y)
   fmt.Println("Sum",sum)
   fmt.Println("Diff",diff)
}
```

The output will be

```
Sum 25
Diff 5
```

Packages

Packages are used to organize the code. In a big project, it is not feasible to write code in a single file. Go allow us to organize the code under different packages. This increases code readability and reusability. An executable Go program should contain a package named main and the program execution starts from the function named main. You can import other packages in our program using the syntax

```
import package_name
```

We will see and discuss how to create and use packages in the following example.

Step 1) Create a file called package_example.go and add the below code

```
package main
import "fmt"
//the package to be created
import "calculation"
func main() {
        x,y := 15,10
        //the package will have function Do_add()
sum := calculation.Do_add(x,y)
fmt.Println("Sum",sum)
}
```

In the above program fmt is a package which Go provides us mainly for I/O purposes. Also, you can see a package named calculation. Inside the main() you can see a step sum := calculation.Do_add(x,y). It means you are invoking the function Do_add from package calculation.

Step 2) First, you should create the package calculation inside a folder with the same name under src folder of the go. The installed path of go can be found from the PATH variable.

For mac, find the path by executing echo $PATH
Step 3) Navigate to to the src folder(/usr/local/go/src for mac and C:\Go\src for windows). Now from the code, the package name is calculation. Go requires the package should be placed in a directory of the same name under src directory. Create a directory named calculation in src folder.
Step 4) Create a file called calc.go (You can give any name, but the package name in the code matters. Here it should be calculation) inside calculation directory and add the below code

```
package calculation

func Do_add(num1 int, num2 int)(int) {
   sum := num1 + num2
   return sum
}
```

Step 5) Run the command go install from the calculation directory which will compile the calc.go.
Step 6) Now go back to package_example.go and run go run package_example.go. The output will be Sum 25.
Note that the name of the function Do_add starts with a capital letter. This is because in Go if the function name starts with a capital letter it means other programs can see(access) it else other programs cannot access it. If the function name was do_add , then You would have got the error cannot refer to unexported name calculation.calc..

Defer and stacking defers

Defer statements are used to defer the execution of a function call until the function that contains the defer statement completes execution.
Lets learn this with an example:

```
package main
import "fmt"
func sample() {
   fmt.Println("Inside the sample()")
}
func main() {
   //sample() will be invoked only after executing the statements of
main()
   defer sample()
   fmt.Println("Inside the main()")
}
```

The output will be

```
Inside the main()
Inside the sample()
```

Here execution of sample() is deferred until the execution of the enclosing function(main()) completes.
Stacking defer is using multiple defer statements. Suppose you have multiple defer statements inside a function. Go places all the deferred function calls in a stack, and once the enclosing function returns, the stacked functions are executed in the Last In First Out(LIFO) order. You can see this in the below example.
Execute the below code

```
package main
import "fmt"
func display(a int) {
   fmt.Println(a)
}
func main() {
    defer display(1)
    defer display(2)
    defer display(3)
    fmt.Println(4)
}
```

The output will be

```
4
3
2
1
```

Here the code inside the main() executes first, and then the deferred function calls are executed in the reverse order, i.e. 4, 3,2,1.

Pointers

Before explaining pointers let's will first discuss '&' operator. The '&' operator is used to get the address of a variable. It means '&a' will print the memory address of variable a.
Execute the below program to display the value of a variable and the address of that variable

```
package main
import "fmt"

func main() {
	a := 20
	fmt.Println("Address:",&a)
	fmt.Println("Value:",a)
}
```

The result will be

```
Address: 0xc000078008
Value: 20
```

A pointer variable stores the memory address of another variable. You can define a pointer using the syntax

```
	var variable_name *type
```

The asterisk(*) represents the variable is a pointer. You will understand more by executing the below program

```
package main
import "fmt"
func main() {
	//Create an integer variable a with value 20
	a := 20
```

```
        //Create a pointer variable b and assigned the address of a
        var b *int = &a
        //print address of a(&a) and value of a
        fmt.Println("Address of a:",&a)
        fmt.Println("Value of a:",a)
        //print b which contains the memory address of a i.e. &a
        fmt.Println("Address of pointer b:",b)
        //*b prints the value in memory address which b contains i.e.
the value of a
        fmt.Println("Value of pointer b",*b)
        //increment the value of variable a using the variable b
        *b = *b+1
        //prints the new value using a and *b
        fmt.Println("Value of pointer b",*b)
        fmt.Println("Value of a:",a)}
```

The output will be

```
Address of a: 0x416020
Value of a: 20
Address of pointer b: 0x416020
Value of pointer b 20
Value of pointer b 21
Value of a: 21
```

Structures

A Structure is a user defined datatype which itself contains one more element of the same or different type.
Using a structure is a 2 step process.
First, create(declare) a structure type
Second, create variables of that type to store values.
Structures are mainly used when you want to store related data together.
Consider a piece of employee information which has name, age, and address. You can handle this in 2 ways
Create 3 arrays - one array stores the names of employees, one stores age and the third one stores age.
Declare a structure type with 3 fields- name, address, and age. Create an array of that structure type where each element is a structure object having name, address, and age.
The first approach is not efficient. In these kinds of scenarios, structures are more convenient.
The syntax for declaring a structure is

```
type structname struct {
  variable_1 variable_1_type
  variable_2 variable_2_type
  variable_n variable_n_type
}
```

An example of a structure declaration is

```
type emp struct {
   name string
   address string
   age int
}
```

Here a new user defined type named emp is created. Now, you can create variables of the type emp using the syntax

```
        var variable_name struct_name
```

An example is

```
var empdata1 emp
```

You can set values for the empdata1 as

```
empdata1.name = "John"
        empdata1.address = "Street-1, Sydney"
        empdata1.age = 30
```

You can also create a structure variable and assign values by

```
empdata2 := emp{"Sam", "Building-1, Zurich", 25}
```

Here, you need to maintain the order of elements. Sam will be mapped to name, next element to address and the last one to age.
Execute the code below

```
package main
import "fmt"
//declared the structure named emp
type emp struct {
        name string
        address string
        age int
}
//function which accepts variable of emp type and prints name
property
func display(e emp) {
        fmt.Println(e.name)
}
func main() {
// declares a variable, empdata1, of the type emp
var empdata1 emp
//assign values to members of empdata1
empdata1.name = "John"
empdata1.address = "Street-1, London"
empdata1.age = 30
//declares and assign values to variable empdata2 of type emp
empdata2 := emp{"Sam", "Building-1, Paris", 25}
//prints the member name of empdata1 and empdata2 using display
function
display(empdata1)
display(empdata2)
}
```

Output

```
John
Sam
```

Methods(not functions)

A method is a function with a receiver argument. Architecturally, it's between the func keyword and method name. The syntax of a method is

```
func (variable variabletype) methodName(parameter1
paramether1type) {
}
```

Let's convert the above example program to use methods instead of function.

```
package main
import "fmt"
//declared the structure named emp
type emp struct {
   name string
   address string
   age int
}
//Declaring a function with receiver of the type emp
func(e emp) display() {
   fmt.Println(e.name)
}
func main() {
   //declaring a variable of type emp
   var empdata1 emp

   //Assign values to members
```

```
    empdata1.name = "John"
    empdata1.address = "Street-1, London"
    empdata1.age = 30
    //declaring a variable of type emp and assign values to members
    empdata2 := emp {
        "Sam", "Building-1, Paris", 25}
    //Invoking the method using the receiver of the type emp
   // syntax is variable.methodname()
    empdata1.display()
    empdata2.display()
}
```

Go is not an object oriented language and it doesn't have the concept of class. Methods give a feel of what you do in object oriented programs where the functions of a class are invoked using the syntax objectname.functionname()

Concurrency

Go supports concurrent execution of tasks. It means Go can execute multiple tasks simultaneously. It is different from the concept of parallelism. In parallelism, a task is split into small subtasks and are executed in parallel. But in concurrency, multiple tasks are being executed simultaneously. Concurrency is achieved in Go using Goroutines and Channels.

Goroutines

A goroutine is a function which can run concurrently with other functions. Usually when a function is invoked the control gets transferred into the called function, and once its completed execution control returns to the calling function. The calling function then continues its execution. The calling function waits for the invoked function to complete the execution before it proceeds with the rest of the statements.
But in the case of goroutine, the calling function will not wait for the execution of the invoked function to complete. It will continue to execute with the next statements. You can have multiple goroutines in a program.
Also, the main program will exit once it completes executing its statements and it will not wait for completion of the goroutines invoked.
Goroutine is invoked using keyword go followed by a function call.
Example

```
go add(x,y)
```

You will understand goroutines with the below examples. Execute the below program

```
package main
import "fmt"

func display() {
	for i:=0; i<5; i++ {
	fmt.Println("In display")
	}
}
func main() {
	//invoking the goroutine display()
	go display()
	//The main() continues without waiting for display()
	for i:=0; i<5; i++ {
	fmt.Println("In main")
	}
}
```

The output will be

```
In main
In main
In main
In main
In main
```

Here the main program completed execution even before the goroutine started. The display() is a goroutine which is invoked using the syntax

```
go function_name(parameter list)
```

In the above code, the main() doesn't wait for the display() to complete, and the main() completed its execution before the display() executed its code. So the print statement inside display() didn't get printed.
Now we modify the program to print the statements from display() as well. We add a time delay of 2 sec in the for loop of main() and a 1 sec delay in the for loop of the display().

```
package main
import "fmt"
import "time"

func display() {
        for i:=0; i<5; i++ {
        time.Sleep(1 * time.Second)
        fmt.Println("In display")
        }
}
func main() {
        //invoking the goroutine display()
        go display()
        for i:=0; i<5; i++ {
        time.Sleep(2 * time.Second)
        fmt.Println("In main")
        }
}
```

The output will be somewhat similar to

```
In display
In main
In display
In display
In main
In display
In display
In main
In main
In main
```

Here You can see both loops are being executed in an overlapping fashion because of the concurrent execution.

Channels

Channels are a way for functions to communicate with each other. It can be thought as a medium to where one routine places data and is accessed by another routine.
A channel can be declared with the syntax

```
channel_variable := make(chan datatype)
```

Example:

```
    ch := make(chan int)
```

You can send data to a channel using the syntax

```
channel_variable <- variable_name
```

Example

```
  ch <- x
```

You can receive data from a channel using the syntax

```
  variable_name := <- channel_variable
```

Example

```
  y := <- ch
```

In the above examples of goroutine, you have seen the main program doesn't wait for the goroutine. But that is not the case when channels are involved. Suppose if a goroutine pushes data to channel, the main() will wait on the statement receiving channel data until it gets the data.
You will see this in below example. First, write a normal goroutine and see the behaviour. Then modify the program to use channels and see the behaviour.
Execute the below program

```
package main
import "fmt"
import "time"

func display() {
	time.Sleep(5 * time.Second)
	fmt.Println("Inside display()")
}
func main() {
	go display()
	fmt.Println("Inside main()")
}
```

The output will be

```
Inside main()
```

The main() finished the execution and did exit before the goroutine executes. So the print inside the display() didn't get executed.
Now modify the above program to use channels and see the behaviour.

```
package main
import "fmt"
import "time"

func display(ch chan int) {
	time.Sleep(5 * time.Second)
	fmt.Println("Inside display()")
	ch <- 1234
}
func main() {
	ch := make(chan int)
	go display(ch)
	x := <-ch
	fmt.Println("Inside main()")
	fmt.Println("Printing x in main() after taking from
channel:",x)
}
```

The output will be

```
Inside display()
Inside main()
Printing x in main() after taking from channel: 1234
```

Here what happens is the main() on reaching x := <-ch will wait for data on channel ch. The display() has a wait of 5 seconds and then push data to the channel ch. The main() on receiving the data from the channel gets unblocked and continues its execution.
The sender who pushes data to channel can inform the receivers that no more data will be added to the channel by closing the channel. This is mainly used when you use a loop to push data to a channel. A channel can be closed using

```
close(channel_name)
```

And at the receiver end, it is possible to check whether the channel is closed using an additional variable while fetching data from channel using

```
variable_name, status := <- channel_variable
```

If the status is True it means you received data from the channel. If false, it means you are trying to read from a closed channel
You can also use channels for communication between goroutines. Need to use 2 goroutines – one pushes data to the channel and other receives the data from the channel. See the below program

```
package main
import "fmt"
import "time"
//This subroutine pushes numbers 0 to 9 to the channel and closes
the channel
func add_to_channel(ch chan int) {
	fmt.Println("Send data")
	for i:=0; i<10; i++ {
	ch <- i //pushing data to channel
	}
	close(ch) //closing the channel
}
//This subroutine fetches data from the channel and prints it.
func fetch_from_channel(ch chan int) {
	fmt.Println("Read data")
	for {
	//fetch data from channel
x, flag := <- ch
	//flag is true if data is received from the channel
//flag is false when the channel is closed
if flag == true {
	fmt.Println(x)
	}else{
	fmt.Println("Empty channel")
	break
	}
	}
}
func main() {
	//creating a channel variable to transport integer values
	ch := make(chan int)
	//invoking the subroutines to add and fetch from the channel
	//These routines execute simultaneously
	go add_to_channel(ch)
	go fetch_from_channel(ch)
	//delay is to prevent the exiting of main() before goroutines
finish
	time.Sleep(5 * time.Second)
	fmt.Println("Inside main()")
}
```

Here there are 2 subroutines one pushes data to the channel and other prints data to the channel. The function add_to_channel adds the numbers from 0

to 9 and closes the channel. Simultaneously the function fetch_from_channel waits at
x, flag := <- ch and once the data become available, it prints the data. It exits once the flag is false which means the channel is closed.
The wait in the main() is given to prevent the exiting of main() until the goroutines finish the execution.
Execute the code and see the output as

```
Read data
Send data
0
1
2
3
4
5
6
7
8
9
Empty channel
Inside main()
```

Select

Select can be viewed as a switch statement which works on channels. Here the case statements will be a channel operation. Usually, each case statements will be read attempt from the channel. When any of the cases is ready(the channel is read), then the statement associated with that case is executed. If multiple cases are ready, it will choose a random one. You can have a default case which is executed if none of the cases is ready.
Let's see the below code

```
package main
import "fmt"
import "time"
//push data to channel with a 4 second delay
func data1(ch chan string) {
    time.Sleep(4 * time.Second)
    ch <- "from data1()"
}
//push data to channel with a 2 second delay
func data2(ch chan string) {
    time.Sleep(2 * time.Second)
    ch <- "from data2()"
}
func main() {
    //creating channel variables for transporting string values
    chan1 := make(chan string)
    chan2 := make(chan string)
        //invoking the subroutines with channel variables
    go data1(chan1)
    go data2(chan2)
        //Both case statements wait for data in the chan1 or chan2.
    //chan2 gets data first since the delay is only 2 sec in data2().
    //So the second case will execute and exits the select block
    select {
    case x := <-chan1:
        fmt.Println(x)
    case y := <-chan2:
        fmt.Println(y)
    }
}
```

Executing the above program will give the output:

```
from data2()
```

Here the select statement waits for data to be available in any of the channels. The data2() adds data to the channel after a sleep of 2 seconds which will cause the second case to execute.
Add a default case to the select in the same program and see the output. Here, on reaching select block, if no case is having data ready on the channel, it will execute the default block without waiting for data to be available on any channel.

```
package main
import "fmt"
import "time"
//push data to channel with a 4 second delay
func data1(ch chan string) {
    time.Sleep(4 * time.Second)
    ch <- "from data1()"
}
//push data to channel with a 2 second delay
func data2(ch chan string) {
    time.Sleep(2 * time.Second)
    ch <- "from data2()"
}
func main() {
    //creating channel variables for transporting string values
    chan1 := make(chan string)
    chan2 := make(chan string)

    //invoking the subroutines with channel variables
    go data1(chan1)
    go data2(chan2)
    //Both case statements check for data in chan1 or chan2.
    //But data is not available (both routines have a delay of 2 and 4
sec)
    //So the default block will be executed without waiting for data
in channels.
    select {
    case x := <-chan1:
        fmt.Println(x)
    case y := <-chan2:
        fmt.Println(y)
    default:
        fmt.Println("Default case executed")
    }
}
```

This program will give the output:

```
Default case executed
```

This is because when the select block reached, no channel had data for reading. So, the default case is executed.

Mutex

Mutex is the short form for mutual exclusion. Mutex is used when you don't want to allow a resource to be accessed by multiple subroutines at the same time. Mutex has 2 methods - Lock and Unlock. Mutex is contained in sync package. So, you have to import the sync package. The statements which have to be mutually exclusively executed can be placed inside mutex.Lock() and mutex.Unlock().

Let's learn mutex with an example which is counting the number of times a loop is executed. In this program we expect routine to run loop 10 times and the count is stored in sum. You call this routine 3 times so the total count should be 30. The count is stored in a global variable count.
First, You run the program without mutex

```
package main
import "fmt"
import "time"
import "strconv"
import "math/rand"
//declare count variable, which is accessed by all the routine
instances
var count = 0
//copies count to temp, do some processing(increment) and store
back to count
//random delay is added between reading and writing of count
variable
func process(n int) {
    //loop incrementing the count by 10
    for i := 0; i < 10; i++ {
    time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
    temp := count
    temp++
    time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
    count = temp
    }
    fmt.Println("Count after i="+strconv.Itoa(n)+" Count:",
strconv.Itoa(count))
}
func main() {
    //loop calling the process() 3 times
    for i := 1; i < 4; i++ {
    go process(i)
    }
    //delay to wait for the routines to complete
    time.Sleep(25 * time.Second)
    fmt.Println("Final Count:", count)
}
```

See the result

```
Count after i=1 Count: 11
Count after i=3 Count: 12
Count after i=2 Count: 13
Final Count: 13
```

The result could be different when you execute it but the final result won't be 30.
Here what happens is 3 goroutines are trying to increase the loop count stored in the variable count. Suppose at a moment count is 5 and goroutine1 is going to increment the count to 6. The main steps include
Copy count to temp
Increment temp
Store temp back to count
Suppose soon after performing step 3 by goroutine1; another goroutine might have an old value say 3 does the above steps and store 4 back, which is wrong. This can be prevented by using mutex which causes other routines to wait when one routine is already using the variable.
Now You will run the program with mutex. Here the above mentioned 3 steps are executed in a mutex.

```
package main
import "fmt"
import "time"
import "sync"
import "strconv"
import "math/rand"
//declare a mutex instance
var mu sync.Mutex
//declare count variable, which is accessed by all the routine
instances
var count = 0
//copies count to temp, do some processing(increment) and store
back to count
//random delay is added between reading and writing of count
variable
```

```
func process(n int) {
	//loop incrementing the count by 10
	for i := 0; i < 10; i++ {
	time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
	//lock starts here
	mu.Lock()
	temp := count
	temp++
	time.Sleep(time.Duration(rand.Int31n(2)) * time.Second)
	count = temp
	//lock ends here
	mu.Unlock()
	}
	fmt.Println("Count after i="+strconv.Itoa(n)+" Count:",
strconv.Itoa(count))
}
func main() {
	//loop calling the process() 3 times
	for i := 1; i < 4; i++ {
	go process(i)
	}
	//delay to wait for the routines to complete
	time.Sleep(25 * time.Second)
	fmt.Println("Final Count:", count)
}
```

Now the output will be

```
Count after i=3 Count: 21
Count after i=2 Count: 28
Count after i=1 Count: 30
Final Count: 30
```

Here we get the expected result as final output. Because the statements reading, incrementing and writing back of count is executed in a mutex.

Error handling

Errors are abnormal conditions like closing a file which is not opened, open a file which doesn't exist, etc. Functions usually return errors as the last return value.
The below example explains more about the error.

```
package main
import "fmt"
import "os"
//function accepts a filename and tries to open it.
func fileopen(name string) {
    f, er := os.Open(name)
    //er will be nil if the file exists else it returns an error object
    if er != nil {
        fmt.Println(er)
        return
    }else{
        fmt.Println("file opened", f.Name())
    }
}
func main() {
    fileopen("invalid.txt")
}
```

The output will be:

```
open /invalid.txt: no such file or directory
```

Here we tried to open a non-existing file, and it returned the error to er variable. If the file is valid, then the error will be null

Custom errors

Using this feature, you can create custom errors. This is done by using New() of error package. We will rewrite the above program to make use of custom errors.
Run the below program

```
package main
import "fmt"
import "os"
import "errors"
//function accepts a filename and tries to open it.
func fileopen(name string) (string, error) {
    f, er := os.Open(name)
    //er will be nil if the file exists else it returns an error object
    if er != nil {
        //created a new error object and returns it
        return "", errors.New("Custom error message: File name is
wrong")
    }else{
         return f.Name(),nil
    }
}
func main() {
    //receives custom error or nil after trying to open the file
    filename, error := fileopen("invalid.txt")
    if error != nil {
        fmt.Println(error)
    }else{
         fmt.Println("file opened", filename)
    }
}
```

The output will be:

```
Custom error message:File name is wrong
```

Here the area() returns the area of a square. If the input is less than 1 then area() returns an error message.

Reading files

Files are used to store data. Go allows us to read data from the files
First create a file, data.txt, in your present directory with the below content.

```
Line one
Line two
Line three
```

Now run the below program to see it prints the contents of the entire file as output

```
package main
import "fmt"
import "io/ioutil"
func main() {
    data, err := ioutil.ReadFile("data.txt")
    if err != nil {
        fmt.Println("File reading error", err)
        return
    }
    fmt.Println("Contents of file:", string(data))
}
```

Here the data, err := ioutil.ReadFile("data.txt") reads the data and returns a byte sequence. While printing it is converted to string format.

Writing files

You will see this with a program

```
package main
import "fmt"
import "os"
func main() {
    f, err := os.Create("file1.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    l, err := f.WriteString("Write Line one")
    if err != nil {
        fmt.Println(err)
        f.Close()
        return
    }
    fmt.Println(l, "bytes written")
    err = f.Close()
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Here a file is created, test.txt. If the file already exists then the contents of the file are truncated. Writeline() is used to write the contents to the file. After that, You closed the file using Close().

GO Interview Questions

1. What is Go?

Go is a general-purpose language designed with systems programming in mind.It was initially developed at Google in year 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. It is strongly and statically typed, provides inbuilt support for garbage collection and supports concurrent programming. Programs are constructed using packages, for efficient management of dependencies. Go programming implementations use a traditional compile and link model to generate executable binaries.

2. What are the benefits of using Go programming?

Following are the benefits of using Go programming −

- Support for environment adopting patterns similar to dynamic languages. For example type inference (x := 0 is valid declaration of a variable x of type int).
- Compilation time is fast.
- InBuilt concurrency support: light-weight processes (via goroutines), channels, select statement.
- Conciseness, Simplicity, and Safety.
- Support for Interfaces and Type embedding.
- Production of statically linked native binaries without external dependencies.

3. What is static type declaration of a variable in Go?

Static type variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable declaration at the time of linking of the program.

4. What is dynamic type declaration of a variable in Go?

A dynamic type variable declaration requires compiler to interpret the type of variable based on value passed to it. Compiler don't need a variable to have type statically as a necessary requirement.

5. How to print type of a variable in Go?

Following code prints the type of a variable −

```
var a, b, c = 3, 4, "foo"
fmt.Printf("a is of type %T\n", a)
```

6. What is a pointer?

It's a pointer variable which can hold the address of a variable.

For example −

```
var x = 5
var p *int
p = &x
fmt.Printf("x = %d", *p)
```

Here x can be accessed by *p.

7. What is the purpose of break statement?

break terminates the for loop or switch statement and transfers execution to the statement immediately following the for loop or switch.

8. Explain the syntax to create a function in Go.

The general form of a function definition in Go programming language is as follows −

```
func function_name( [parameter list] ) [return_types] {
   body of the function
}
```

A function definition in Go programming language consists of a function header and a function body. Here are all the parts of a function −

- func func starts the declaration of a function.
- Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- Return Type − A function may return a list of values. The return_types is the list of data types of the values the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the not required.

- Function Body − The function body contains a collection of statements that define what the function does.

9. In how many ways you can pass parameters to a method?
While calling a function, there are two ways that arguments can be passed to a function −

- Call by value − This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- Call by reference − This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

10. What is the default way of passing parameters to a function?
By default, Go uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.
11. What do you mean by function as value in Go?
Go programming language provides flexibility to create functions on the fly and use them as values. We can set a variable with a function definition and use it as parameter to a function.
12. What are the function closures?
Functions closure are anonymous functions and can be used in dynamic programming.
13. What are methods in Go?
Go programming language supports special types of functions called methods. In method declaration syntax, a "receiver" is present to represent the container of the function. This receiver can be used to call function using "." operator.
14. What is the difference between actual and formal parameters?
The parameters sent to the function at calling end are called as actual parameters while at the receiving of the function definition called as formal parameters.
15. What is the difference between variable declaration and variable definition?

Declaration associates type to the variable whereas definition gives the value to the variable.
16. Explain modular programming.
Dividing the program in to sub programs (modules/function) to achieve the given task is modular approach. More generic functions definition gives the ability to re-use the functions, such as built-in library functions.
17. What is a token?
A Go program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol.
18. What is a nil Pointers in Go?
Go compiler assign a Nil value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned nil is called a nil pointer. The nil pointer is a constant with a value of zero defined in several standard libraries.
19. What is a pointer on pointer?
It's a pointer variable which can hold the address of another pointer variable. It de-refers twice to point to the data held by the designated pointer variable.

```
var a int
var ptr *int
var pptr **int
a = 3000
ptr = &a
pptr = &ptr

fmt.Printf("Value available at **pptr = %d\n", **pptr)
```

Therefore 'a' can be accessed by **pptr.
20. What is structure in Go?
Structure is another user defined data type available in Go programming, which allows you to combine data items of different kinds.
21. How to define a structure in Go?
To define a structure, you must use type and struct statements. The struct statement defines a new data type, with more than one member for your

program. type statement binds a name with the type which is struct in our case.
The format of the struct statement is this −

```
type struct_variable_type struct {
   member definition;
   member definition;
   ...
   member definition;

}
```

22. What is slice in Go?
Go Slice is an abstraction over Go Array. As Go Array allows you to define type of variables that can hold several data items of the same kind but it do not provide any inbuilt method to increase size of it dynamically or get a sub-array of its own. Slices covers this limitation. It provides many utility functions required on Array and is widely used in Go programming.
23. How to get a sub-slice of a slice?
Slice allows lower-bound and upper bound to be specified to get the subslice of it using[lower-bound:upper-bound].
24. What is range in Go?
The range keyword is used in for loop to iterate over items of an array, slice, channel or map. With array and slices, it returns the index of the item as integer. With maps, it returns the key of the next key-value pair.
25. What are maps in Go?
Go provides another important data type map which maps unique keys to values. A key is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a Map object. After value is stored, you can retrieve it by using its key.

And finally, if you liked the book, I would like to ask you to do me a favor and leave a review for the book on Amazon. Just go to your account on Amazon and write a review for this book. Thank you and good luck!