Rust Programming

For Beginners
Quick Start Guide

Ray Yao

Rust Programming

Rust Exercises

Rust

Programming

For Beginners

Quick Start Guide

Ray Yao

About the Author: Ray Yao

Certified PHP engineer by Zend, USA

Certified JAVA programmer by Sun, USA

Certified SCWCD developer by Oracle, USA

Certified A+ professional by CompTIA, USA

Certified ASP**.** NET expert by Microsoft, USA

Certified MCP professional by Microsoft, USA

Certified TECHNOLOGY specialist by Microsoft, USA

Certified NETWORK+ professional by CompTIA, USA

www.amazon.com/author/ray-yao

Recommended Books by Ray Yao

Advanced C++ Programming

Advanced Java Programming

AngularJs Programming

Asp.Net Programming

C# Exercises

C# Programming

C++ Exercises

C++ Programming

Dart Programming

Data Science Programming

Django Programming

Git Programming

Go Programming

Html Css Exercises

Html Css Programming

Java Exercises

Java Programming

JavaScript Exercises

JavaScript Programming

JQuery Exercises

JQuery Programming

Jsp Servlets Programming

Kotlin Programming

Linux Command Line

Linux Exercises

Lua Programming

Machine Learning Programming

Matlab Programming

MySql Programming

Node.Js Programming

Numpy Programming

Pandas Programming

Perl Programming

Php Exercises

Php MySql Programming

PowerShell Programming

Python Exercises

Python Programming

R Programming

React.Js Programming

Ruby Programming

Rust Programming

Scala Programming

Shell Scripting Programming

Swift Programming

TypeScript Programming

Visual Basic Exercises

Visual Basic Programming

Vue.Js Programming

Xml Json Programming

Preface

“Rust Programming & Exercises” covers all essential Rust language knowledge**.** You can learn complete primary skills of Rust programming fast and easily**.**

The book includes many practical examples for beginners and includes exercises for the college exam, the engineer certification exam, and the job interview exam**.**

Note:

This book is suitable for Rust programming beginners; it is not for the Rust experience programmers.

Table of Content

Chapter 1

What is Rust Language?

Rust is a programming language developed by Mozilla that focuses on security and concurrency. The Rust language is jointly developed by the leader of the web language, Brendan Eich, Dave Herman, and Graydon Hoare of Mozilla.

Rust is an excellent language for security, concurrency, and utility, for multi-core systems and absorbs important features of other dynamic languages such as: it doesn’t need to manage memory, without Null Pointers, and so on.

The Rust's syntax is similar to that of C++.

Rust is free open source software that is freely available to anyone and publicly shared so that people can also improve the design of the software.

The great feature of Rust language is the Security.

The Feature of Rust

1. Rust uses the abstract functionality without affecting the runtime performance of the code
2. Rust's error messages are clear and easy to understand, appear in a neat, color-coded format, and suggest spell checking in the program**.**
3. Rust provides data-type checking functionality, which means it can automatically determine the type of an expression**.**
4. Rust language can replace a copy operation with a move operation**.**
5. Rust provides thread functionality without data contention**.**
6. Rust uses the match pattern to better manage the program's control flow**.**
7. Rust guarantees memory security by using the concept of ownership**.**
8. Rust language can communicate with c language easily**.**
9. Rust programmers can clearly control the memory allocation in release of memory locations and time**.**

Install Rust

(1) Install C++ Build Tools

Before you install Rust, you must install **C++** Build Tools**.**

1. Click the following link to download C++ Build Tools**:**
2. https://visualstudio.microsoft.com/downloads/#build-tools-for-visual-studio-2017
3. You can find**:**

4. Click **Download** button on the right side, start downloading**.**
5. After downloading the Build Toos installer, please install **Build Tools for Visual Studio 2017**……
6. During installation, select options that relate to C++ to install**.**
7. After the installation is complete, please restart the computer**.**

(2) Install Rust

If you have already installed Build Toos, then continue to install Rust**.**

1. Click the following link to download Rust
2. https://www.rust-lang.org/tools/install
3. Please download the Rust installer "**RUSTUP-INIT.EXE**"**.**

4. After downloading the "**rustup-init.exe**" installer, please double click it to install Rust**.**
5. You can see the following message**:**

6. Type **1**, press **Enter** key to install Rust**.**
7. When the installation is complete, please press **Enter** key to confirm**.**

(3) Rust Test

Test Rust to check if the installation is successful**.**

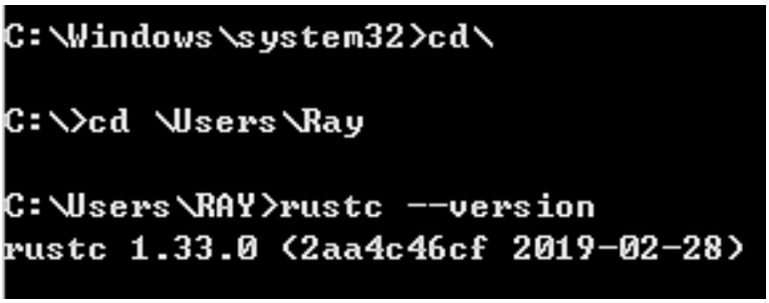
The Rust working folder is **C: \Users\Your_Name.** For example, my Rust working folder is **C: \Users\RAY.**

Rust working folder always includes three sub-folders**:** (1)**.**rustup folder, (2)**.**multirust folder, (3)**.**cargo folder**.**

Now we need to go to working folder first, and test Rust program**.**

Please run **cmd** command, open the command line editor**.**

1. Type **cd** to move to the root directory**.**
2. Type **cd \Users\Your_Name** to move to the working folder**.**
3. Run **rustc --version** command, you can see the output**:**



Congratulation**!** The Rust installation is successful**!**

Note:

In different Windows OS, the working folder is different**.**

For example, your working folder may be C**:**\Users\Admin**.**

Then you must find out the folder that contains three sub-folders**.** (**.**rustup folder, **.**multirust folder, **.**cargo folder)**.** And make sure this folder as a working folder**.**

The First Rust Program

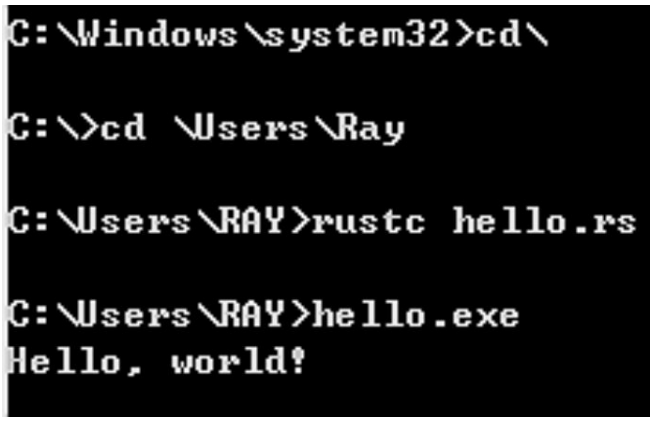
Please open the Notepad, and write the following code:

Example 1.1

```
fn main() {
        println!("Hello, world!");
}
```

Save the file as “hello.rs” in the working folder, using .rs as extension name.

Run “cmd” command, open the command line editor. Enter the following commands.



Output:

Hello, world!

Explanation:

fn main( ) defines a function named “main”**.**

println**!**() is a command used to output content**.**

The extension name of Rust file is “**.**rs”**.**

Each statement ends with a semicolon “**;**”

“cd\” moves to root directory**.**

“C**:** \>cd \Users\RAY” moves to my working folder**.**

“**rustc hello.rs**” compiles the Rust file “hello**.**rs”

“C**:**\Users\RAY>**hello.exe**” runs the hello**.**exe program**.**

Comment

```
//  Single line comment
/*      */   Multi line comment
```

The comments are always ignored by the compiler**.**

Example 1.2

```
fn main() {  // define a function named main
        println!("Hello, world!");
/* println!( ) is a command used to output
contents or text.
The comments are always ignored by the
compiler.
*/
}
```

Output:

Hello, world**!**

Explanation:

“//” is used in a single line comment**.**

“/* */” is used in multi line comments**.**

The comments are always ignored by the compiler**.**

Summary

fn main( ) defines a function named “main”**.**

println**!**() is a command used to output content**.**

The extension name of Rust file is “**.**rs”**.**

Each statement ends with a semicolon “**;**”

“//” is used in a single line comment**.**

“/* */” is used in multi line comments**.**

The comments are always ignored by the compiler**.**

Chapter 2

Data Type

Rust is a language with strong data type, and all variables and constants must have an explicit data type declaration;

Rust has data structures of integer, floating point, Boolean, string, array, tuple, enumeration, and struct**.** For example

Integer**:** i8, i16, i32, i64, isize, u8, u16, u32, u64, usize

Floating-point**:** f32, f64,

Boolean**:** true or false

String**:** “C# in 8 hours”

Array**:** a = [1,2,3]

Slices**:** &a[element1**…**element2]

Tuple**:** t = (1, 2, 3)

Char**:** ‘c’

"u" means unsigned data and "i" means signed data**.**

"u32" represents an unsigned 32-bit integer, and "i64" represents a signed 64-bit integer**.**

“isize” and “usize” are types that can vary in size**.**

Variables

The syntax to define a variable is as follows**:**

```
let variable_name = value
```

Example 2.1

```
fn main(){
let var = "OK";
println!("The value of var is: { }", var);
}
```

Output:

The value of var is OK**.**

Explanation:

let var = "OK"; assigns a value “OK” to the variable “var”**.**

println**!**("{ }", var); is a typical output format in Rust**.** “{ }” cannot be omitted**.**

Initialization with Data Type

Using a "**:**" to define a variable with a data type**.**

```
let variable_name : type = value;
```

Example 2.2

```
fn main(){
let var: i32 = 100;
let str: String = "Good".to_string();
println!("The value of var is: { }", var);
println!("The value of str is: { }", str);
}
```

Output:

The value of var is 100

The value of str is Good

Explanation:

"let var**:** i32 = 100;" defines the type of var as i32**.**

"let str**:** String = "Good"**.**to_string();" defines the type of str as String**.**

"**.**to_string()" converts the value to String type**.**

Output Format

```
println!(“{ }, variable”);     // print content and then change line
print!(“{ } { }”, variable1, variable2);   // print in the same line
```

Example 2.3

```
fn main(){
let x = 100;
let y = 200;
let z = 300;
println!("{ }", x);
print!("{ } { }", y, z);
}
```

Output:

100

200 300

Explanation:

“println!(“{ }, variable”);” print content and then the change line

“print!(“{ } { }”, variable1, variable2);” print text in the same line

Constants

The syntax to define a constant is as follows**:**

```
const  identifier : type = value
```

Example 2.4

```
const  NUM: i32 = 100;
fn main() {
println!("The value of NUM is {}", NUM);
}
```

Output:

The value of NUM is 100

Explanation:

“const NUM**:** i32 = 100;” defines a constant NUM, its value is 100**.**

Date Type Conversion

```
as new_type
```

“as” is a keyword which is used to convert data type**.**

Example 2.5

```
fn main(){
let var1: f32 = 100.88;
let var2: i32 = var1 as i32;      // var1 becomes i32
println!("{}", var1);
println!("{}", var2);
}
```

Output:

100.88

100

Explanation:

“var1 as i32;” means that the type of var1 is converted to i32**.**

Reserved Words

abstract alignof as become box
break const continue crate do
else enum extern false final
fn for if impl in
let loop macro match mod
move mut offsetof override priv
proc pub pure ref return
Self self sizeof static struct
super trait true type typeof
unsafe unsized use virtual where
while yield

Rust reserved words cannot be used when choosing identifier names for variables, functions, properties**.**

Function

(1) The syntax to define a function is as follows**:**

```
fn function_name(parameters) {
}
```

(2) The syntax to call a function is as follows**:**

```
function(parameters);
```

Example 2.6

```
fn main() {
    funt(100, 200);    // calls the function
}
fn funt(x: i32, y: i32) {  // define a function
    println!("The sum is: { }", x + y);
}
```

Output:

The sum is 300

Explanation:

“funt(100, 200)” calls the function named funt( ){…}, and pass the parameters 100 and 200 to the funt( ){…}.

(x: i32, y: i32) declares the data type of parameters are i32.

Return Type

The syntax to specify a return type for a function is as follows**:**

```
fn function_name(parameters)  ->  return_type {
}
```

Example 2.7

```
fn main() {
    let num = funt(100);   // calls the function
    println!("The value of num is: { }", num);
}
fn funt(num: i32)  ->  i32 {  // specify a return type
    num + 200      // return a value to the caller
}
```

Output:

The value of num is 300

Explanation:

“funt(100);” is a caller with a parameter 100**.**

“ -> **i32**” specifies the return type is i32

“num+200” returns a value to the caller**.**

Note:

The body of the function ends without a semicolon “**;**”;

In here, “num+200” is the last statement of function, so it has no semicolon “**;**”.

Example 2.8

```
fn foo()  ->  bool {   // specify a return type
  return true    // return a value to the caller
}
fn main() {
    let b = foo();   // foo() is a caller
    println!("The result is: { }", b);
}
```

Output:

The result is true

Explanation:

“ -> **bool**” specifies a return type as Boolean**.**

“return true” returns a value to the caller**.**

“foo();” calls the function “foo(){}”**.**

Summary

The syntax to define a variable is as follows**:**

let variable_name = value;

let variable_name **:** type = value;

“println**!**(“{ }, variable”);” print content and then the change line

“print**!**(“{ } { }”, variable1, variable2);” print text in the same line

The syntax to define a constant is as follows**:**

const identifier **:** type = value

The syntax of date type conversion**:** as new_type

Rust reserved words cannot be used when choosing identifier names for variables, functions, properties**.**

(1) The syntax to define a function is as follows**:**

fn function_name(parameters) {

}

(2) The syntax to call a function is as follows**:**

function(parameters);

The syntax to specify a return type for a function is as follows**:**

fn function_name(parameters) -> return_type {

}

Chapter 3

Variable-Binding

(1) For the sake of security, the value of a bound variable is immutable by default**.**

```
let (var1, var2) = (value1, value2);
```

After a variable binding, the var1 is equal to value1, var2 is equal to value2**.** The values are unchangeable**.**

Example 3.1

```
fn main(){
let (x, y) = (100, 200);   // variable-binding
println!("The value of x is {}", x);
println!("The value of y is {}", y);
}
```

Output:

The value of x is 100

The value of y is 200

Explanation:

"**let (x, y) = (100, 200);**" binds two immutable values**.**

(2) If you want the value of the bound variable mutable, you can use the "**mut**" keyword**.** But it is the lack of security**.**

```
let mut variable = value;
```

Example 3.2

```
fn main(){
let mut a = 100;     // variable-binding
let mut b = 200;
a = a + 300;
b = b + 400;
println!("Finally a is {}", a);
println!("Finally b is {}", b);
}
```

Output:

Finally a is 400

Finally b is 600

Explanation:

"let mut a = 100;" binds a mutable value**.**

String Assignment

There are three methods to define strings

```
let x = “hello”.to_string();     // convert text to a string
let y = String::from("hello");    // get text directly
let z:&str = "hello";      // reference a text
```

Example 3.3

```
fn main(){
let x = "hello".to_string();
let y = String::from("hello");
let z:&str = "hello";
print!("{}  {}  {} ", x, y, z);
}
```

Output:

hello hello hello

Explanation:

There are three methods to assign a value to a string variable.

Arithmetical Operators

Operators	Running
+	add
-	subtract
*	multiply
/	divide
%	get modulus

% modulus operator divides the first number by the second number and returns the remainder. e.g. 9%2=1.

Example 3.4

```
fn main() {
    println!("10 + 2 = {}", 10 + 2);
    println!("10 - 2 = {}", 10 - 2);
    println!("10 * 2 = {}", 10 * 2);
    println!("10 / 2 = {}", 10 / 2);
    println!("10 % 2 = {}", 10 % 2);
}
```

Output:

10 + 2 = 12

10 – 2 = 8

10 * 2 = 20

10 / 2 =5

10 % 2 = 0

Explanation:

In “10 % 2 = 0”, % modulus operator divides the first number by the second number and returns the remainder**.** e**.**g**.** 9%2=1**.**

Logical Operators

Operators	Equivalent
&&	and
||	or
!	not

After using logical operators, the result will be true or false**.**

Example 3.5

```
fn main() {
println!("true AND false is {}", true && false);
println!("true OR false is {}", true || false);
println!("NOT true is {}", ! true);
}
```

Output:

true AND false is false

true OR false is true

NOT true is false

Explanation:

true && true; returns true;	true && false; returns false;	false &&false; returns false;
true II true; returns true;	true II false; returns true;	false II false; return false;
! false; returns true;	**!** true; returns false;	

Comparison Operators

Operators	Running
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
==	equal
!=	not equal

After using comparison operators, the result will be true or false.

Example 3.6

```
fn main() {
let x:i32 = 100;
let y:i32 = 200;
println!("x is greater than y : {}", x > y);
println!("x is less than y : {}", x < y);
println!("x is unequal to y : {}", x != y);
println!("x is greater/equal to y : {}", x >= y);
println!("x is less/equal to y : {}", x <= y);
println!("x is completely equal to y : {}", x == y);
}
```

Output:

x is greater than y **:** false

x is less than y **:** true

x is unequal to y **:** true

x is greater/equal to y false

x is less/equal to y **:** true

x is completely equal to y **:** false

Explanation:

After using comparison operators, the result will be true or false**.**

Array

An array is a kind of variable that includes multiple values**.**

(1) The first way to create an array**:**

```
let mut array: [type; length] = [default; length];
```

Example 3.7

```
fn main() {
let mut a: [i32; 4] = [8; 4];    // create an array
a[1] = 10;
a[2] = 20;
println!("{} {} {} {}",a[0], a[1], a[2], a[3]);
}
```

Output:

8 10 20 8

Explanation:

“let mut a**:** [i32; 4] = [8; 4];” creates an array

The array type is i32, Array length is 4 elements**.** The default value is 8**.**

“a[1]=10” assigns the value 10 to the element a[1]**.**

“println**!**(“{}”, a[0])” prints the first element**.**

(2) The second way to create an array**:**

```
let array: [type; length] = [val1, val2, val3……];
```

Example 3.8

```
fn main(){
let a:[f32; 4] = [0.1, 0.2, 0.3, 0.4];    // create an
array
println!("{} {} {} {}",a[0],a[1],a[2],a[3])
}
```

Output:

0.1 0.2 0.3 0.4

Explanation:

The array is assigned four values when defining**.**

Slice

The slice is a part of an array**.** Extracting a portion of an array can create a slice**.**

The syntax to create a slice is as follows**:**

```
let slice = &array[ start..last-1];
```

Extracting the element from start element to last element-**1** can create a slice**.**

For example:

```
let a = [0, 10, 20, 30, 40, 50, 60];    // create an array
let slice = &a[2..5];  /* extract three elements from a[2] to a[4], now the slice contains three elements */
```

Example 3.9

```
fn main(){
let a = [0, 10, 20, 30, 40, 50, 60];   // create an array
let slice = &a[2..5];     // extract from a[2] to a[4]
println!("{}",slice[0] );     // show slice's elements
println!("{}",slice[1] );
println!("{}",slice[2] );
}
```

Output:

20

30

40

Explanation:

“let slice = &a[2**..**5];” extracts the array “a” from a[2] to a[4], and creates a slice with three elements**.**

Summary

The syntax of the variable-binding**:**

let (var1, var2) = (value1, value2);

let mut variable = value;

let x = “hello”**.**to_string(); // convert text to a string

let y = String**::**from("hello"); // get text directly

let z**:**&str = "hello"; // reference a text

Rust has operators such as**:** arithmetical operators, logical operators, comparison operators,

The syntax to create an array**:**

let mut array**:** [type; length] = [default; length];

let array**:** [type; length] = [val1, val2, val3……];

The syntax to create a slice is as follows**:**

let slice = &array[start**..**last-1];

Chapter 4

If Statement

```
if condiction {   // if true do this;   }
```

“if statement” executes codes inside { … } only if a specified condition is true**.**

Example 4.1

```
fn main(){
let num=10;
if num==10{
println!("num is equal to 10");
}
}
```

Output:

num is equal to 10

Explanation:

“num==10” is true, so if statement can be executed**.**

If-else Statement

```
if ( test-expression) {   // if true do this;
} else {  // if false do this;
}
```

“if-else statement” runs some code if a condition is true, and runs another code if the condition is false

Example 4.2

```
fn main(){
let x=100;
let y=200;
if x>y{
println!("x is greater than y");
} else {
println!("x is smaller than y");
} }
```

Output: x is smaller than y

Explanation:

“x>y” is false, so the else statement has been executed.

Let-If Statement

```
Let variable = if condition{
// if true do this
} else {
 // if false do this
}
```

The value of the if expression will be assigned to the let statement**.**

Example 4.3

```
fn main(){
let num = if true{          // let-if statement
100
} else {
200
};
println!("The value of num is {}", num)
}
```

Output: The value of num is 100

Explanation:

“let num = if true{…}” is a let-if statement**.**

Loop – Break Statement

The loop statement executes the code repeatedly until the break statement stops it**.**

```
fn main(){
loop{
……
break;
}
}
```

Example 4.4

```
fn main(){
let mut num=5;
loop{     // loop statement
println!("C# in {} Hours", num );
if num == 8 {
break;     // break statement
}
num=num+1;
}}
```

Output:

C# in 5 Hours

C# in 6 Hours

C# in 7 Hours

C# in 8 Hours

Explanation:

Loop statement run repeatedly until num is equal to 8, it will stop**.**

For Statement

A for loop is a conditional loop, running a specified number of times until the condition is false**.**

```
for var in condition {
}
```

Example 4.5

```
fn main(){
for num in 5..9 {   // "5..9" contains numbers from 5 to 8
println!("Java in {} Hours", num);
}}
```

Output:

Java in 5 Hours
Java in 6 Hours
Java in 7 Hours
Java in 8 Hours

Explanation:

"num1**..**num2" is a range expression, which contains numbers from num1 to num2**-1.** Note that "num1**..** num2" does not include num2**.**

"5**..**9" is a range expression, which contains numbers from 5 to 8**.** Note that "5**..**9" does not include 9**.**

While Statement

The while statement is a conditional loop, and when the condition is true, it executes the loop, otherwise it terminates the loop**.**

```
while condition{
// if true do this;
}
```

Example 4.6

```
fn main(){
let mut num=0;
while num<=8 {       // while statement
print!("{} ", num);
num=num+1;
}}
```

Output:

0 1 2 3 4 5 6 7 8

Explanation:

When num is less than or equal to 8, the while loop will run repeatedly**.**

Tuples

A tuple is a collection with different type elements.

(1) The syntax to create a tuple is as follows:

```
let tuple = (val1, val2, val3…);
```

Note: In Tuple, the type of the values can be different, using ( ) parentheses.

(2) Access the elements of the tuple

```
tuple.index
```

For example: “tuple.0” accesses the first element of the tuple.

Example 4.7

```
fn main(){
let t = ("Python in", 8, "Hours", true);   // create a tuple
print!("{} {} {} {}", t.0, t.1, t.2, t.3);   // access the elements
}
```

Output:

Python in 8 Hours true

Explanation:

“let t = ("Python in", 8, "Hours", true);” creates a tuple with different type elements**.**

“t**.**0, t**.**1, t**.**2, t**.**3” accesses four elements of the tuple “t”**.**

Note**:** In Tuple, the type of the values can be different, using ( ) parentheses**.**

Match

The Match statement valuates the given expression first, and compares the each value, and runs the code when the value matches**.**

```
match variable {
value1 => code1,
value2 => code2,
……
_default => code,    // if don't match above values, run this.
}
```

Example 4.8

```
fn main(){
let num:i32 = 3;    // given expression
match num {
1 => println!("one"),
2 => println!("two"),
3 => println!("three"),    // match this
4 => println!("four"),
_ => println!("something else"),
}}
```

Output:

three

Explanation:

The given expression is 3, which matches the value 3**.**

“_ => println**!**("something else"),” means that if don't match above values, run this**.**

Summary

“if statement” executes codes inside { … } only if a specified condition is true**.**

The syntax of Let-If Statement

Let variable = if condition{

} else {

}

The loop statement executes the code repeatedly until the break statement stops it**.**

The syntax of For statement**:**

for var in condition {……}

The syntax of While statement**:**

while condition{……}

(1) The syntax to create a tuple is as follows**:**

let tuple = (val1, val2, val3…);

(2) Access the elements of the tuple

tuple**.**index

The Match statement valuates the given expression first, and compares the each value, and runs the code when the value matches**.**

Chapter 5

Struct

Struct is a user-defined data type that is defined by using the struct keyword**.** The members of the struct contain members' names and types which are enclosed in braces**.** The struct members are called fields**.** There are three steps to use struct**:**

(1) Create a struct

```
struct Struct_Name {
member1: type,
member2: type,
.....
}
```

(2) Initialize the struct

```
let object = Struct_Name {
member1: value1,
member2: value2,
……
}
```

(3) Access the member

```
object. member
```

Example 5.1

```
struct Member {   // create a struct
id: i32,                  // member: type
name: String,
working: bool,
}
fn main() {
let clerk = Member {   // initialize the struct
id: 016320,                  // member: value
name: "Smith".to_string(),
working: true,
};
println!("ID is {}", clerk.id);   // access the members
println!("Name is {}", clerk.name);
println!("Working is {}", clerk.working);
}
```

Output:

ID is 016320

Name is Smith

Working is true

Explanation:

“struct Member {…}” creates a struct

“let clerk = Member” {…} initializes the struct

“**clerk.id**”, “**clerk.name**” and “**clerk.working**”accesse the members**.**

Example 5.2

```
struct Square {     // create a struct
len: i32,
wid: i32,
}
fn main() {
let table = Square { len: 10, wid: 8 };   // initialization
println!("The area is {}", table.len * table.wid);   // access
}
```

Output:

The area is 80

Explanation:

“struct Square{…}” creates a struct**.**

“let table = Square { len**:** 10, wid**:** 8 };” initializes the struct**.**

“table**.**len*table**.**wid” accesses the members of the struct**.**

Enum

Enumeration is a custom data type that contains certain values**.** Use the enum keyword to define**.**

(1) Define an enum

```
enum Enum_Name{
   member 1,
   member 2,
    ……
}
```

(2) Access to member

```
Enum_Name::member
```

Example 5.3

```
enum Language{    // define an enum
JS,      // member
GO,
VB,
}
fn program(var:Language){
match var{     // using match statement
Language::JS=> println!("JS in 8 Hours"),
```

```
Language::GO=> println!("GO in 8 Hours"),
Language::VB=> println!("VB in 8 Hours"),
}
}
fn main(){
program(Language::JS);    // access the member
program(Language::GO);
program(Language::VB);
}
```

Output:

JS in 8 Hours

GO in 8 Hours

VB in 8 hours

Explanation:

“enum Language{…}” defines an enum

“Language::JS” accesses the member.

Ownership

When a variable binding owns a resource, it is called ownership**.** But when a variable binding goes out of scope, it releases the resource, and loses ownership**.**

Example 5.4

```
fn main(){
let x = String::from("try");   // x owns “try”
let y = x;      // Warning! The ownership of x moves to y
println!("{}", x);    // Error!   x is no longer available
}
```

Output:

Error message……

Explanation:

“let x = String**::**from("try");” means that “x” owns the value “try”, “x” is an owner**.**

“let y = x” moves the ownership to y**.** Then x has no ownership**.**

“println**!**(“{}”, x)” print an error message**.** x is no longer available**.**

Example 5.5

```
fn main()  {
let s=String::from("R in 8 Hours");  // s owns “R in 8 Hours”
let n=cal(s);   // Warning! s will lose the ownership after used
println!("Value of the string is: {}",s); // s is no longer available
println!("Length of the string is: {}",n);
}
fn cal(s:String)  ->  usize {
s.len()    // get the length of the string
}
```

Output:

Error message……

Explanation:

“let s=String::from("R in 8 Hours");” means that “s” owns the “R in 8 Hours”. “s” is an owner.

“let n=cal(s);” calls the function “cal”, and passes the parameter “s”. After “s” is used, “s” will lose the ownership of the “R in 8 Hours”.

“println!("Value of the string is: {}",s);” will print error messages, because “s” is no longer available.

Reference

After a variable is referenced by other variables, the ownership of its value remains and will not be lost**.**

(1) Reference a variable**.**

```
&variable
```

(2) Reference a parameter

```
parameter: &type
```

Example 5.6

```
fn main()  {
let s=String::from("R in 8 Hours");
let n=cal(&s);     // reference
println!("Value of the string is: {}",s);
println!("Length of the string is: {}",n);
}
fn cal(s:&String) ->  usize {      // reference
s.len()    // get the length of the string
}
```

Output:

Value of the string is**:** R in 8 Hours

Length of the string is**:** 12

Explanation:

“let n=cal(**&s**);” calls the function “cal”**.** After “s” is referenced, “s” still owns the value of “R in 8 Hours”**.**

“fn cal(s**:&String**) -> usize” means that “s” is a referenced string type**.**

“usize” is the type whose size can vary**.**

Summary

(1) Create a struct

struct Struct_Name {

member1**:** type,

member2**:** type,

.....

}

(2) Initialize the struct

let object = Struct_Name {

member1**:** value1,

member2**:** value2,

……

}

(3) Access the member

object**.** member

(1) Define an enum

enum Enum_Name{

member 1,

member 2,

……

}

(2) Access to member

Enum_Name::member

About Ownership:

When a variable binding owns a resource, it is called ownership. But when a variable binding goes out of scope, it releases the resource, and loses ownership.

(1) Reference a variable.

&variable

(2) Reference a parameter

parameter: &type

Chapter 6

Module

A module is a namespace that contains functions, structures, traits, and impl projects**.** By default, the module modifier is private, but the public attributes can be set using the “pub” keyword**.**

The module is defined by the “mod” keyword**.**

(1) Define a module

```
mod module_name{
   pub function(){ }
}
```

(2) Run the module

```
module_name::function();
```

Example 6.1

```
mod my_module{       // define a module
pub fn test(){     // pub means public attribute
println!("Hello My Friends!");
}
}
fn main(){
```

```
my_module::test();      // run the module
}
```

Output:

Hello My Friends**!**

Explanation:

“mod my_module{**…**}” defines a module**.**

“my_module**::**test();” runs the module and function**.**

Embedded Module

(1) Define an embedded module

```
mod m1{
   mod m2{
      pub function(){ }
   }
}
```

(2) Run the embedded module

```
m1::m2::function();
```

Example 6.2

```
mod m1{
  pub fn a(){
  println!("m1 module");
  }
     pub mod m2{       // embedded module
        pub fn b(){
        println!("m2 module");
        }
     }
}
```

```
fn main(){
m1::a();
m1::m2::b();    // runs the embedded module & function
}
```

Output:

m1 module

m2 module

Explanation:

“pub mod m2{ }” defines an embedded module**.**

“m1**::**m2**::**b();” runs the embedded module & function**.**

External File

Typically, if you want to reference an external file, you can apply the “mod” keyword to load that file as a module, and “use” keyword to load the external function**.**

(1) First step, create an external file named “ex_file**.**rs” **.**

Example 6.3

```
pub fn ex_fun() {
println!("{}", "External Text");
}
```

Save the file as “ex_file**.**rs”**.**

The syntax to Load the external file and function is**:**

```
mod extern_file;     // “mod “ loads an external file
use extern_file::extern_fun;    // “use” loads an external function
```

(2) Second step, create a main file named “prime.rs”.

Example 6.4

```
mod ex_file;     // loads an external file
use ex_file::ex_fun;      // loads an external function
fn main () {
  ex_fun();      // calls the external function
}
```

Save the file as “prime.rs” in the same folder with “ex_file.rs” file, then compile the prime.rs and run the prime.exe.

Output:

External Text

Explanation:

“mod ex_file;” loads an external file “ex_file.rs”

“use ex_file::ex_fun;” loads an external function “ex_fun(){ }”.

Private Function

In Rust language, all function is private by default**.**

If any function or module is private, it can be accessed through its direct parent module or the module itself**.**

Example 6.5

```
mod my_module  {
    pub fn a()  {    // function is public
    println!("function a");
    }
    fn b(){     // function b is private
    println!("function b");
    }
}
fn main()  {
my_module::a();
my_module::b();     // call a private function
}
```

Output:

Error message……

Explanation:

“fn b(){ }” is a private function**.**

“my_module**::**b()” calls function b from outside the module**.** Therefore an error occurred**.**

Example 6.6

```
mod my_module  {
    pub fn a()  {
    println!("function a");
   b();     // call a private function b
   }
    fn b() {     // function b is private
    println! ("function b");
    }
}
fn main()  {
my_module::a();
}
```

Output:

function a

function b

Explanation:

“b();” calls a private “function b(){}” in the same module my_module, therefore no error occurred.

Super

The super keyword is used to access the parent module from the current module, even access to the parent module's private functions**.**

```
use:: super:: parent_function;
```

Example 6.7

```
mod sup_module{      // parent module
  fn a()  ->  i32 {
   100
   }
    pub mod sub_module {    // child module
   use super::a;    // access parent function a
     pub fn b() {
     println!("{}",a());    // calls parent function a
     }
   }
}
fn main()  {
sup_module::sub_module::b();  // call function b
}
```

Output:

100

Explanation:

“mod sup_module{ }” is a parent module

“pub mod sub_module { }” is a child module

“use super**::**a;” enable to access parent function a**.**

“println**!**("{}",a());” calls parent function a**.**

“sup_module**::**sub_module**::**b();” calls function b**.**

Summary

(1) Define a module

```
mod module_name{
   pub function(){ }
}
```

(2) Run the module

```
module_name::function();
```

(1) Define an embedded module

```
mod m1{
    mod m2{
      pub function(){ }
    }
}
```

(2) Run the embedded module

```
m1::m2::function();
```

```
mod extern_file;     // “mod “ loads an external file
use extern_file::extern_fun;    // “use” loads an external function
```

In Rust language, all function is private by default**.**

If any function or module is private, it can be accessed through its direct parent module or the module itself**.**

```
use:: super:: parent_function;   // accese parent function
```

Chapter 7

Vector

A vector is actually a dynamic or mutable array**.** It is a single data structure that can store multiple values in memory**.**

(1) The first method to create a vector

```
let vecter_name = vec! [val1, val2, val3……];
```

Example 7.1

```
fn main()  {
let v =vec![100, 200, 300, 400];    // create a vector
println!("First element is :{}",v[0]);    // access the first element
println!("Second element is :{}",v[1]);
println!("Third element is :{}",v[2]);
println!("Fourth element is :{}",v[3]);
}
```

Output:

First element is**:** 100

Second element is**:** 200

Third element is**:** 300

Fourth element is**:** 400

Explanation:

v[0] accesses the first element of the vector

v[1] accesses the second element of the vector**.**

(2) The second method to create a vector

```
let v = vec![val; repeat];
```

Example 7.2

```
fn main() {
let v =vec![8; 3];    // repeat three times
println!("First element is :{}",v[0]);
println!("Second element is :{}",v[1]);
println!("Third element is :{}",v[2]);
}
```

Output:

First element is**:** 8

Second element is**:** 8

Third element is**:** 8

Explanation:

“let v =vec**!**[8; 3];” repeats three times to set the values 8**.**

(3) The third method to create a vector

```
let mut v=Vec::new();  // use “new” keyword
v.push('value')   // set value to vector
```

Example 7.3

```
fn main() {
let mut v=Vec::new(); // create a vector
v.push('R');    // set R as the first element of vector
v.push('U');
v.push('B');
v.push('Y');
    for n in v{
    print!("{}",n);
    }
}
```

Output:

RUBY

Explanation:

“let mut v=Vec::new();” creates a vector

“v.push('R');” sets R as the first element of vector

Multiple Patterns

You can use | to match multiple patterns**.**

```
pattern1 | pattern2
```

Example 7.4

```
fn main(){
let num = 3;
match num {
1 => println!("one"),
2 | 3 => println!("two or three"),   // multiple patterns
_ => println!("others"),
}
}
```

Output:

two or three

Explanation:

“**2 | 3**” gets the value two or three**.**

Range

The symbol "**…**" can match the values within the specified range**.**

```
number1 … number2
```

Example 7.5

```
fn main(){
let x = 3;
match x {
2 ... 6 => println!("from 2 to 6"),   // match from 2 to 6
_ => println!("others"),
}
}
```

Output:

from 2 to 6

Explanation:

"2**…**6" matches the number from 2 to 6**.**

Binding a Range

@ can bind a variable to a range

```
variable @ range
```

Example 7.6

```
fn main(){
let x = 5;
match x {
var @ 2 ... 6 => println!("{}",var),    // binding
_ => println!("others"),
}
}
```

Output:

5

Explanation:

“var @ 2…6” binds var to the range 2…6.

Generics

The arguments in a function can accept multiple types of data.

This can be done with generics. Generics are also known as parametric polymorphism.

The Rust standard library provides Options for generics.

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

'T' is a generic data type. It provides a type of generics.

For example:

```
let x : Option<i32> = Some(100);  // 'T' type is i32.
let x : Option<bool> = Some(true);  // 'T' type is bool.
let x : Option<f64> = Some(100.5); // 'T' type is f64.
let x : Option<char> = Some('A');   // 'T' type is char.
```

Example 7.7

```
fn main(){
  let x: Option<bool> = Some(true);     // generic
parameters
  let y: Option<i32> = Some(10);
  let z: Option<f64> = Some(20.88);
  let n: Option<i32> = None;
  match x {
    Some(x) => { println!("x = {}", x) },
    None => { println!("x = None") },
  }
  match y {
    Some(y) => { println!("y = {}", y) },
    None => { println!("y = None") },
  }
  match z {
    Some(z) => { println!("z = {}", z) },
    None => { println!("z = None") },
  }
  match n {
    Some(n) => { println!("n = {}", n) },
    None => { println!("n = None") },
  }
}
```

Output:

x = true

y = 10

z = 20.88

n = None

Explanation:

<T> can be one of the <bool>, <i32>, <f64> types.

Summary

(1) The first method to create a vector

let vecter_name = vec**!** [val1, val2, val3……];

(2) The second method to create a vector

let v = vec**!**[val; repeat];

(3) The third method to create a vector

let mut v=Vec**::**new(); // use “new” keyword

v**.**push('value') // set value to vector

You can use | to match multiple patterns**.**

pattern1 | pattern2

The symbol “**…**” can match the values within the specified range**:**

number1 **…** number2

@ can bind a variable to a range**:**

variable @ range

The Rust standard library provides Options for generics**:**

enum Option<T> {

Some(T),

None,

}

Chapter 8

Method

A function in struct, enum, trait is called as a method.

```
impl Struct/Enum {    // implement Struct or Enum
fn method_name(&self)  ->  type {   // define a method
self.member       // access the member variable
}
}
```

The &self indicates that the caller's reference is taken as a parameter.

The first parameter of the method is &self.

Example 8.1

```
struct Circle{    // create a struct type
radius: f32,      // struct member
}
impl Circle{    // implement the struct
fn area(&self)  ->  f32{    // define a method
std::f32::consts::PI * self.radius * self.radius
}     // method body
}
fn main(){
let obj = Circle { radius : 2000.00};  // create a struct
object
```

```
println!("The Circle area is: {}", obj.area());  // call
the method
}
```

Output:

The Circle area is: 12566371

Explanation:

“impl Circle{ }” implements the struct

“fn area(&self) -> f32{ }” defines a method

“std::f32::consts::PI” imports PI from Rust library.

“self.radius” accesses the struct member “radius”.

Method can also be used in Trait, which will be discussed next.

Trait

Trait is an interface in Rust, it defines a train method without the body and is implemented by an “impl” method which provides various usages **.**

(1) Define a trait method

```
trait Trait_Name{     // define a trait
  fn  trait_method(&self);     // define a trait method
}
```

The first letter of the trait name should be capitalized**.**

The trait method has no the body, just like an interface in Java**.**

(2) Implement the trait method

```
impl Trait_Name for Struct/Enum{  // implement the trait
  fn trait_method(&self){  // implement the trait method
  self.member       // access the member variable
  }
}
```

Example 8.2

```
struct Circle{    // create a struct type
radius: f32,      // struct member
}
trait Calculate{       // define a trait
fn area(&self)  ->  f32;      // define a trait method
```

```
}
impl Calculate for Circle{    // implement the trait
fn area(&self)  ->  f32{    // implement the trait method
std::f32::consts::PI * self.radius * self.radius
}
}
fn main(){
let obj = Circle { radius : 2000.00};  // create a struct object
println!("The Circle area is: {}", obj.area());  // call the method
}
```

Output:

The Circle area is: 12566371

Explanation:

“trait Calculate{ }” defines a trait.

“fn area(&self) -> f32;” defines a trait method.

“impl Calculate for Circle{ }” implements the trait.

“fn area(&self) -> f32{ }” implements the trait method.

Example 8.3

```
pub trait Show {    // define a trait
  fn show(&self);      // define a trait method
}
impl<T> Show for T    // implement the trait with generic
where T: ToString{     // specify the String type
  fn show (&self){       // implement the trait method
  print!("{}",self.to_string());
  }
}
fn main(){
String::from("C# in 8 Hours").show();     // call method
}
```

Output:

C# in 8 Hours

Explanation:

“pub trait Show { }” defines a trait.

“fn show(&self);” defines a trait method.

“impl<T> Show for T” implements the trait with generic

“where T: ToString” specifies the String type

“fn show (&self){ }” implements the trait method

Drop() Method

When the drop () method is automatically invoked, it decrements the reference count, and if the total number of references is zero, it cleans up the associated excess resources**.**

The feature of drop() method**:** “Last in, First out”**.**

```
fn drop(&mut self){
}
```

“&mut self” is used for mutable parameters in different type**.**

Example 8.4

```
struct Game {
number: i32,
}
impl Drop for Game {
fn drop(&mut self) {    // define a drop method
println!("The #{ } Winner.", self.number);
}
}
fn main() {
let _baseball = Game { number: 3 };
let _football = Game { number: 2 };
let _basketball = Game { number: 1 };
```

```
}
```

Output:

The #1 Winner**.**

The #2 Winner**.**

The #3 Winner**.**

Explanation:

“fn drop(&mut self) { }” defines a drop() method**.**

When running the last command, the drop() will be automatically executed, and the sequence of output is “last in first out”**.**

Closure

The closure is the anonymous function of Rust Language**.**

(1) Create a closure

```
let closure_name = | parameter | {   };
```

(2) Call the closure

```
closure_name(parameter);
```

Example 8.5

```
fn main(){
let my_closure = | num: i32 | { num + 200 };  // create a closure
let num = 100;
println!("{}", my_closure(num));    // call the closure
}
```

Output: 300

Explanation:

Closure is an anonymous function in Rust Language**.**

“my_closure” is a closure name**.**

| num**:** i32 | is an argument using | | rather than ( )**.**

{ num + 200 }; is the closure body.

Example 8.6

```
fn main() {
let mut capacity = "Hard disk capacity:
5000".to_string();
{
let mut my_closure = | c: char |{capacity.push(c)};
// closure
my_closure('G');   // call the closure
}
println!("{:?}", capacity);   // {:?} is used to output a
string
}
```

Output: Hard disk capacity: 5000G

Explanation:

“let mut my_closure = | c: char |{capacity.push(c)};” creates a closure

“my_closure('G');” calls the closure.

“push()” adds a character.

Summary

A function in struct, enum, trait is called as a method.

```
impl Struct/Enum {    // implement Struct or Enum
fn method_name(&self)  ->  type {   // define a method
self.member       // access the member variable
}}
```

(1) Define a trait method

```
trait Trait_Name{     // define a trait
  fn  trait_method(&self);     // define a trait method
}
```

(2) Implement the trait method

```
impl Trait_Name for Struct/Enum{  // implement the trait
  fn trait_method(&self){  // implement the trait method
  self.member       // access the member variable
}}
```

The feature of drop() method: “Last in, First out”.

```
fn drop(&mut self){
}
```

(1) Create a closure

```
let closure_name = | parameter | {  };
```

(2) Call the closure

```
closure_name(parameter);
```

Appendix 1

Error Checking

“assert!” is used to check errors. Suppose you declare a variable to be true first, and if it is false after comparison, the program stops executing.

```
assert! (variable == true/false)
```

Example A1

```
fn main()  {
let check : bool = true;    // suppose it is true
assert!(check == true);   // check the error
print!("{}", check);
}
```

Output:

true

Explanation:

The above is a correct program without any error.

Example A2

```
fn main()  {
let check : bool = false;   // suppose it is false
assert!(check == true);   // check the error
print!("{}", check);
}
```

Output:

Error message……

Explanation:

“let check : bool = false;” supposes that the code is false.

“assert!(check == true);” returns false, so the program terminates running immediately, and output the error message.

Appendix 2 Q & A

Questions

Please choose the correct answer.

(01)

```
fill in main() {    // define a function
        println!("Hello, world!");
}
```

A. function

B. def

C. void

D. fn

(02)

```
fn main(){
let var: i32 = 100;
let str: String = "Good".fill in;   // define string type
println!("The value of var is: { }", var);
println!("The value of str is: { }", str);
}
```

A. String

B. toString()

C. to_String()

D. String()

(03)

```
fn main(){
fill in (x, y) = (100, 200);   // variable-binding
println!("The value of x is {}", x);
println!("The value of y is {}", y);
}
```

A. set

B. get

C. pet

D. let

(04)

```
if ( test-expression) {   // if true do this;
} fill in {  // if false do this;
}
```

A. then

B. else

C. otherwise

D. if

(05)

fill in Name { // custom type

member1**:** type,

member2**:** type,

.....

}

A. struct

B. structure

C. let struct =

D. let structure =

(06)

fill in **my_module**{ // define a module

pub fn test(){ // pub means public attribute

println**!**("Hello My Friends**!**");

}

```
}
fn main(){
my_module::test();       // run the module
}
```

A. module

B. mode

C. mod

D. model

(07)

```
fn main()  {
let v = fill in [100, 200, 300, 400];    // create a vector
println!("First element is :{}",v[0]);    // access the first element
println!("Second element is :{}",v[1]);
println!("Third element is :{}",v[2]);
println!("Fourth element is :{}",v[3]);
}
```

A. vector

B. vector!

C. vec

D.vec!

(08)

```
impl Struct/Enum {    // implement Struct or Enum
fn method_name( fill in )  ->  type {   // define a method
self.member       // access the member variable
}
}
```

A. self

B.&self

C.arg

D. &arg

(09)

```
fn main(){
let var1: f32 = 100.88;
let var2: i32 = var1 fill in i32;      // convert data type
println!("{}", var1);
println!("{}", var2);
}
```

A. as

B. convert

C.cast

D.type

(10)

```
fn main(){
let x = "hello".to_string();
let y = String:: fill in ("hello");   // assign a string value
let z:&str = "hello";
print!("{}  {}  {} ", x, y, z);
}
```

A. set

B. get

C. let

D. from

(11)

```
fn main(){
let mut num=5;
fill in {     // loop statement
println!("C# in {} Hours", num );
if num == 8 {
break;     // break statement
```

}

num=num+1;

}

}

A. while

B. do

C. loop

D. for

(12)

fill in Name { // custom type

member1**:** type,

member2**:** type,

.....

}

A. enum

B. enumeration

C. let enum =

D. let enumeration =

(13)

```
mod ex_file;     // loads an external file
fill in ex_file::ex_fun;      // loads an external function
fn main () {
  ex_fun();      // calls the external function
}
```

A. load

B. use

C. import

D. include

(14)

```
fn main(){
let num = 3;
match num {
1 => println!("one"),
2 fill in 3 => println!("two or three"),   // multiple patterns
_ => println!("others"),
}
}
```

A. &

B. $

C. #

D. |

(15)

```
impl Trait_Name for Struct/Enum{  // implement the trait
  fn trait_method(&self){  // implement the trait method
  fill in .member       // access the member variable
  }
}
```

A. enum

B. trait

C. self

D. struct

(16)

```
fn main() {
    let num = funt(100);   // calls the function
    println!("The value of num is: { }", num);
}
fn funt(num: i32)  ->  fill in {  // specify a return type
    num + 200      // return a value to the caller
}
```

A. f64

B. string

C. bool

D. i32

(17)

```
fn main(){
let a:[f32; 3] = fill in 0.1, 0.2, 0.3 fill in ;  // create array
println!("{} {} {} {}",a[0],a[1],a[2])
}
```

A. ( )

B. [ ]

C. < >

D. { }

(18)

```
fn main(){
let t = fill in "R in", 8, "Hours", true fill in ;  // create tuple
print!("{} {} {} {}", t.0, t.1, t.2, t.3);   // access the elements
}
```

A. ( )

B. [ ]

C. < >

D. { }

(19)

```
fn main()  {
let s=String::from("R in 8 Hours");
let n=cal( fill in );     // reference
println!("Value of the string is: {}",s);
println!("Length of the string is: {}",n);
}
fn cal(s: fill in ) ->  usize {      // reference
s.len()    // get the length of the string
}
```

A. String s

B. s String

C. &String &s

D. &s &String

(20)

```
mod sup_module{      // parent module
```

```
   fn a()  ->  i32 {
    100
    }
     pub mod sub_module {    // child module
    use fill in ::a;    // access parent function a
      pub fn b() {
      println!("{}",a());    // calls parent function a
      }
    }
}
fn main()  {
sup_module::sub_module::b();  // call function b
}
```

A. parent

B. sub

C. super

D.upper

(21)

```
fn main(){
let x = 5;
match x {
```

```
var fill in 2 ... 6 => println!("{}",var),  // bind a variable to a range
_ => println!("others"),
}
}
```

A. &

B. $

C. #

D. @

(22)

```
fn main(){
let my_closure = fill in num: i32 fill in { num + 200 };
// create a closure
let num = 100;
println!("{}", my_closure(num));    // call the closure
}
```

A. &

B. $

C. #

D. |

(23)

let x **:** Option< **fill in** > = Some(100); // 'T' type is ?

let x **:** Option< **fill in** l > = Some(true); // 'T' type is ?

let x **:** Option< **fill in** > = Some(100.5); // 'T' type is ?

let x **:** Option< **fill in** > = Some('A'); // 'T' type is ?

A.	bool	char	i32	f64
B.	i32	bool	f64	char
C.	f64	i32	char	bool
D.	char	f64	bool	i32

(24)

fn main() {

let var **:** bool = **false**; // suppose it is false

fill in (var == true); // check the error

print**!**("{}", var);

}

A. check**!**

B. check

C. assert**!**

D. assert

Answers

01. D	13. B
02. C	14. D
03. D	15. C
04. B	16. D
05. A	17. B
06. C	18. A
07. D	19. D
08. B	20. C
09. A	21. D
10. D	22. D
11. C	23. B
12. A	24. C

Recommended Books

Recommended Books by Ray Yao

Advanced C++ Programming

Advanced Java Programming

AngularJs Programming

Asp.Net Programming

C# Exercises

C# Programming

C++ Exercises

C++ Programming

Dart Programming

Data Science Programming

Django Programming

Git Programming

Go Programming

Html Css Exercises

Html Css Programming

Java Exercises

Java Programming

JavaScript Exercises

JavaScript Programming

JQuery Exercises

JQuery Programming

Jsp Servlets Programming

Kotlin Programming

Linux Command Line

Linux Exercises

Lua Programming

Machine Learning Programming

Matlab Programming

MySql Programming

Node.Js Programming

Numpy Programming

Pandas Programming
Perl Programming
Php Exercises
Php MySql Programming
PowerShell Programming
Python Exercises
Python Programming
R Programming
React.Js Programming
Ruby Programming
Rust Programming
Scala Programming
Shell Scripting Programming
Swift Programming
TypeScript Programming
Visual Basic Exercises
Visual Basic Programming
Vue.Js Programming
Xml Json Programming