

Rust Programming (2023)

Level : Beginner

Author : WinHtut

Subject: Computer Science

Org : National Institute of Science and Technology

WinHtut (NCCIOS)

Rust Programming သည် performance နဲ့ safety အတွက် design ဆွဲထားတာ ဖြစ်တဲ့အတွက် များသော အားဖြင့် system programming ဟု ဖော်ပြ ကြပါတယ်။ Rust programming ရဲ့ အချို့သော အရာတွေဖြစ်တဲ့ for loop ကဲ့သို့ looping ပတ်တာတွေ statements တွေအဆုံးမှာ semicolon နဲ့ အဆုံး သတ်တာတွေ နဲ့ curly braces ကို အသုံးပြုပြီး block တွေကို သတ်မှတ်ထားတာတွေက c programming နဲ့ အတော်ကို ဆင်တူတာ တွေ့ရပါတယ်။

Rust သည် statically typed language ဖြစ်ပြီး သူ့ရဲ့ variables တွေဟာ သူ့ရဲ့ type တွေကို မချိန်းနိုင်ပါဘူး ဥပမာ number ကနေ string သို့ ချိန်းတာမျိုးပါ။ အကယ်၍ စာဖတ်သူဟာ c/c++ , java စတဲ့ programming languages တွေနဲ့ ထိတွေ့ပြီးသား ဖြစ်ရင်တော့ statically typed language တွေကို ရင်းနှီးပြီးသား ဖြစ်နေမှာပါ။

သင်အနေနဲ့ rust မှာ variable တစ်ခုကို အသုံးပြုဖို့ သူ့ရဲ့ type ကို မကြေငြာ ပေးလည်း အဆင်ပြေပါတယ် အဘယ်ကြောင့် ဆိုသော် rust compiler က ထို ကိစ္စကို ကိုင်တွယ် ပေးသွားမှာ ဖြစ်ပြီး သူ့ရဲ့ type ကိုတော့ ပြန်ချိန်းခြင်း ခွင့်ပြု ပေးမှာ မဟုတ်ပါဘူး။ အကယ်၍ သင်က dynmaically typed language တွေဖြစ်တဲ့ Perl , JavaScript , Python တို့နဲ့ ထိတွေ့ ပြီးသား ဆိုရင်တော့ အသစ်ပုံစံ ဖြစ်နေမှာပါ။ ထို့ပြင် သတိပြုရန် လိုအပ်သေးသည်မှာ rust သည် object-oriented(OO) language မဟုတ်သည့် အတွက် clesses တို့ inheritance တို့ သူ့မှာ ရှိနေမှာ မဟုတ်ပါဘူး သို့သော် struct (structure) ပါဝင်ပြီး ရှုပ်ထွေးနဲ့ type တွေကို ဖန်တီး အသုံးပြု နိုင်ပါတယ်။ ထို့ပြင် ထို structures တွေထဲမှာ method တွေကို အသုံးပြုနိုင်သလို data တွေကိုလည်း mutate လုပ်နိုင်ပါတယ် ထို့ကြောင့် object ဟု ခေါ်ကြသော်လည်း ပုံမှန် အားဖြင့်တော့ object မဟုတ်ပါဘူး။

Rust မှာ အချို့သော functional language တွေ ဖြစ်တဲ့ Haskell မှ ideas တွေလည်းပါဝင် နေပါသေးတယ်။ (variables အကြောင်း နဲ့ functions

သင်ခန်းစာများကြမှ အသေးစိတ် ဆက်လက် ဆွေးနွေးပါမည်) ဥပမာ functions တွေဟာ first-class value တွေကဲ့သို့ arguments တွေ အဖြစ် ဖြတ်နိုင်တာမျိုးတွေပါ နဲ့ algebraic data types (ADTs) အကြောင်းများပါ။

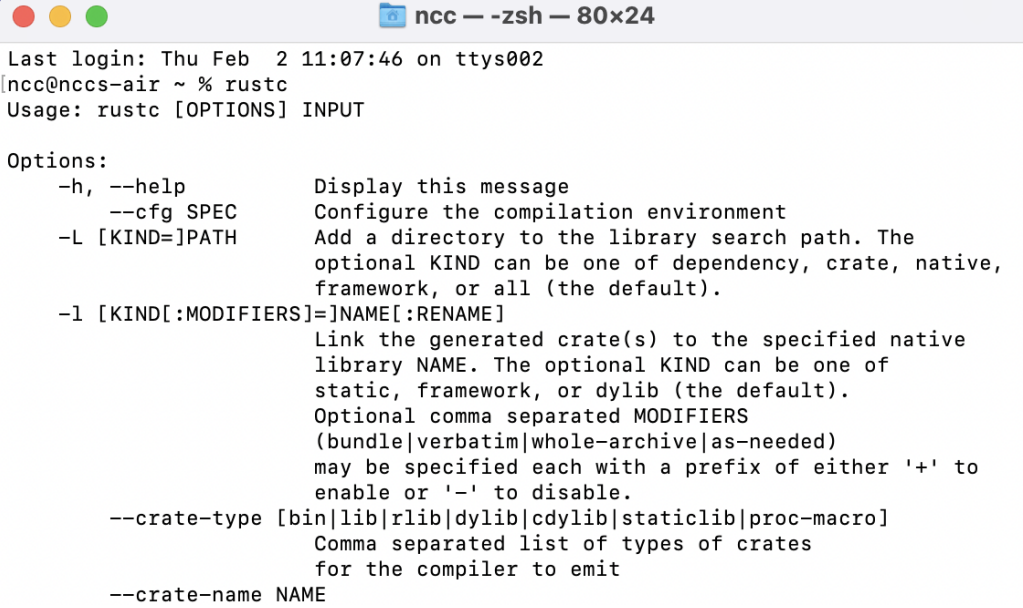
Rust Programming အား စတင် အသုံးပြုဖို့ ဆိုလျှင် မိမိ တို့ computer ထဲမှာ rust programming အား install လုပ်ပေးထားရန် လိုအပ်ပါသည်။ Rust အား install လုပ်ရန် Linux , MacOS သို့မဟုတ် အခြား သော Unix-like OS များအတွက် ဆိုလျှင် အောက်ပါ command အား အသုံးပြုပြီး ပြုလုပ်နိုင်ပါသည်။

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

အကယ်၍ windows os အတွက် အသုံး ပြုလိုလျှင်တော့ အောက်ပါ link သို့ သွားပြီး step by step ပြုလုပ်နိုင်ပါသည်။

<https://forge.rust-lang.org/infra/other-installation-methods.html>

မိမိတို့ computer ထဲတွင် rust အား install ပြုလုပ်လို့ ပြီးပါက အောက်ပါ command ဖြင့် installation အောင်မြင်ခြင်း မအောင်မြင်ခြင်းကို စမ်းသပ် နိုင်သည်။

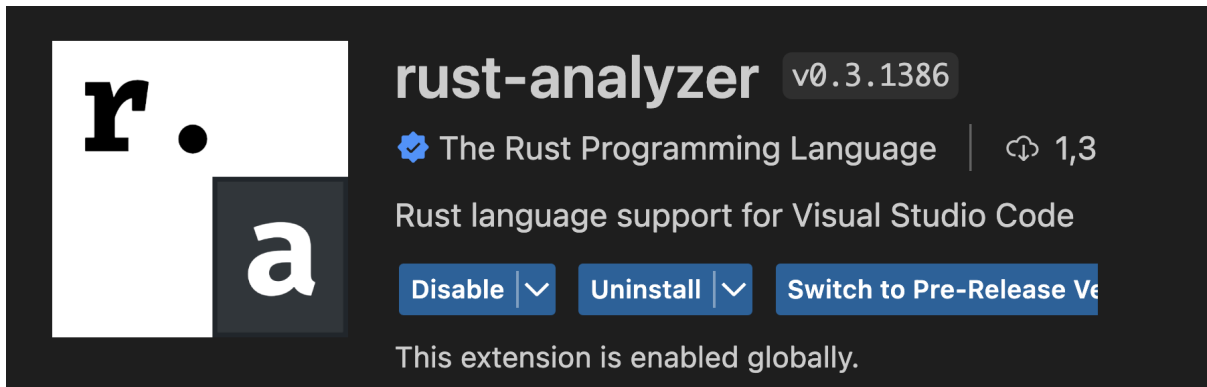


```
ncc -- -zsh -- 80x24
Last login: Thu Feb  2 11:07:46 on ttys002
[ncc@nccs-air ~ % rustc
Usage: rustc [OPTIONS] INPUT

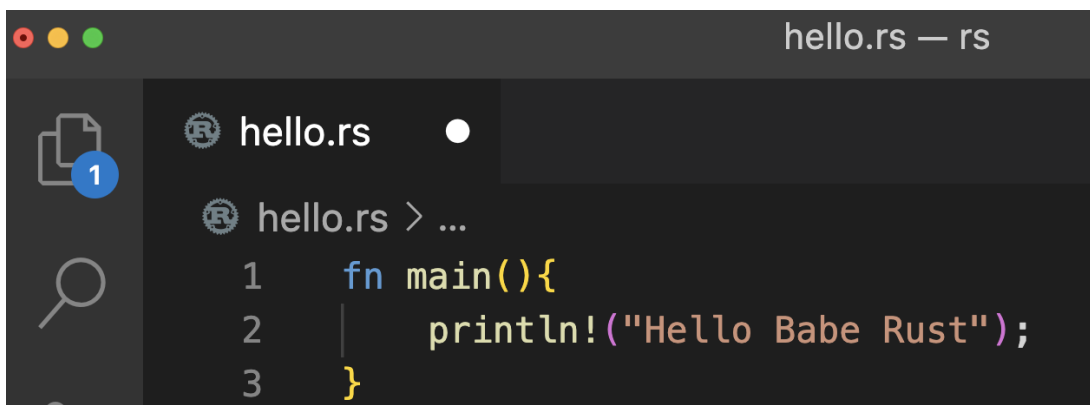
Options:
  -h, --help                Display this message
  --cfg SPEC                Configure the compilation environment
  -L [KIND=]PATH            Add a directory to the library search path. The
                             optional KIND can be one of dependency, crate, native,
                             framework, or all (the default).
  -l [KIND[:MODIFIERS]]=NAME[:RENAME]
                             Link the generated crate(s) to the specified native
                             library NAME. The optional KIND can be one of
                             static, framework, or dylib (the default).
                             Optional comma separated MODIFIERS
                             (bundle|verbatim|whole-archive|as-needed)
                             may be specified each with a prefix of either '+' to
                             enable or '-' to disable.
  --crate-type [bin|lib|rlib|dylib|cdylib|staticlib|proc-macro]
                             Comma separated list of types of crates
                             for the compiler to emit
  --crate-name NAME
```

အထက်ပါ အတိုင်း rustc ဆိုသည့် command အား အသုံးပြုပြီး rust နဲ့ ပတ်သက်သည့် အချက်အလက်များ ကျလာပါက မိမိတို့ computer တွင် rust အား စတင် အသုံးပြု နိုင်ပြီ ဖြစ်သည်။ Visual Studio Code ကိုလည်းအသုံးပြု နိုင်သလို အခြား သော မိမိတို့နှစ်သက်ရာ IDEs များကိုလည်း အသုံးပြုနိုင်ပါတယ်။ အကယ်၍ visual studio code ကို အသုံးပြုမည် ဆိုလျှင်တော့ rust-analyzer ကို အသုံးပြုနိုင်ပါတယ်။ rust-analyzer ကို သုံးခြင်း အားဖြင့် code အလိုလျှောက် ဖြည့်ပေးတာတွေ (code completion) ကြော်ငြာထားတဲ့ နေရာတွေကို ပြန်သွား ပေးတာတွေ (go to definition) နဲ့ code များကို အမျိုးအစား အလိုက် colors ခွဲခြား ပေးတာတွေပါ ပြုလုပ် ပေးပါတယ်။

ထို့ကြောင့် rust-analyzer အား အသုံးပြုရန် visual studio code ထဲမှ extension ထဲတွင်း ရှာပြီး install ပြုလုပ်နိုင်ပါတယ်။



အထက်ပါ extension အား install ပြုလုပ် ပြီးသည်နှင့် rust ကို စတင် ရေးသား နိုင်ပါပြီ။ မိမိတို့ အဆင်ပြေသည့် နေရာတွင် (command line ဖြင့်သွားတတ်သော) hello.rs ဆိုသည့် file တစ်ခုကို တည်ဆောက်ပြီး rust code များကို ရေးသား နိုင်ပါတယ်။ rust programming တွင် rust source code files များသည် .rs နှင့် ဆုံးသည်ကို beginner များ အနေဖြင့် သတိပြုရန် လိုအပ်ပါသည်။



- Line 1 တွင် main ဆိုသည့် function တစ်ခုကို ကြေငြာ ထားပြီး သူ့ရှေ့တွင် keyword အနေဖြင့် fn ကို သုံးထားပါသည်။ ထို့ကြောင့် rust တွင် function များကို ကြေငြာ သော အခါ function name ရှေ့တွင် fn ကို အသုံးပြုရမည်။
- Line 2 တွင် စာသား အချို့ကို println (print line) ကို အသုံးပြုထားပါသည်။ println ကို STDOUT (standard out) အဖြစ်သုံးခြင်း ဖြစ်သည်။ ; semicolon ကတော့ ထို statement ပြီးဆုံးပြီ ဖြစ်ကြောင်း ဖော်ပြ ပေးခြင်း ဖြစ်ပါတယ်။
- Line 3 Function ရဲ့ body ပိုင်းကို ကန့်သတ်ထားရန် { } curly braces များကို သုံးပါတယ်။

C programming အတိုင်းကဲ့သို့ပင် rust သည်လည်း main function ကနေပဲ

အလုပ် စလုပ်ပါတယ် ထို function အတွက် လိုအပ်တဲ့ arguments တွေကို function name ဘေးမှ () parentheses ထဲတွင် ဖော်ပြပါတယ်။ ယခု သင်ခန်းစမှာတော့ main function သည် မည်သည့် argument မှ မယူသည့် အတွက် () ထဲတွင် blank ဖြစ်နေခြင်း ဖြစ်ပါတယ်။ println သည် function ဟု ထင်ရသော်လည်း function မဟုတ်ပါဘူး သူသည် macro ဖြစ်ပါတယ်။ (macro အကြောင်းကို တော့ macro သင်ခန်းစမှာ အသေးစိတ် ဆွေးနွေး ပေးပါမည်)

အထက်ပါ program အား run ရန် ဦးစွာ compile လုပ် ပေးရမည် ဖြစ်သည် compile လုပ်ရန် Unix-like os များတွင် rustc hello.rs ကို သုံးပြီး windows ဆိုလျှင်တော့ rustc.exe ကိုသုံးပါတယ် အောက်ပါ ပုံကတော့ mac os ပေါ်မှာ အသုံးပြု ထားတာ ဖြစ်ပါတယ်။ rustc သုံးပြီး compile လုပ်ပြီးသော အခါတွင် hello ဆိုသည့် file ကို ရရှိမှာ ဖြစ်ပါတယ်။ ထို့ကြောင့် ./hello ဟု run ပြီး output ကို ကြည့်နိုင်ပါတယ်။ forward slash ရှေ့မှ . (dot) ယခု လက်ရှိ ရှိနေသည့် current directory ကို ဖော်ပြ ပေးတာ ဖြစ်ပါတယ်။ စာရေးသူသည် mac os ကို အသုံးပြုထားခြင်း ဖြစ်သည့် အတွက် . (dot) နောက်တွင် forward slash ကိုသုံးခြင်းဖြစ်ပါတယ်။

အကယ်၍ windows os မှာ အသုံးပြုမည် ဆိုလျှင်တော့ . (dot) နောက်တွင် backward slash ကို အသုံးပြုမှာ ဖြစ်ပြီး .exe ပါ ထည့်ပေးရမှာ ဖြစ်ပါတယ်။ example .\hello.exe

The screenshot shows a terminal window with a dark background. At the top, there are four tabs: 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is selected and underlined. Below the tabs, there are three lines of text, each preceded by a colored circle (blue, blue, and grey respectively):

```
● ncc@nccs-air rs % rustc hello.rs
● ncc@nccs-air rs % ./hello
  Hello Babe Rust
○ ncc@nccs-air rs %
```

File command ကိုသုံးပြီးတော့လည်း မည်သို့ file ဖြစ်ကြောင်းကို သိရှိနိုင်ပါသေးတယ်။

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
● ncc@nccs-air rs % rustc hello.rs
● ncc@nccs-air rs % ./hello
  Hello Babe Rust
● ncc@nccs-air rs % file hello
  hello: Mach-0 64-bit executable arm64
○ ncc@nccs-air rs % █

```

Rust Project Directory

စာရေးသူ အနေဖြင့် hello ဆိုသည့် directory တစ်ခု တည်ဆောက်ပါမည်
ထို့ကြောင့် current directory ထဲမှ hello ဆိုသည့် binary file အားဖျက်ပါမည်။

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
● ncc@nccs-air rs % rustc hello.rs
● ncc@nccs-air rs % ./hello
  Hello Babe Rust
● ncc@nccs-air rs % file hello
  hello: Mach-0 64-bit executable arm64
○ ncc@nccs-air rs % █

```

ထို့နောက် src directory ကိုပါ ထပ်မံ ဖန်တီးမှာ ဖြစ်ပါတယ်။

```

● ncc@nccs-air rs % mkdir -p hello/src
● ncc@nccs-air rs % ls
  hello          hello.rs
● ncc@nccs-air rs % mv hello.rs hello/src
● ncc@nccs-air rs % ls
  hello
● ncc@nccs-air rs % cd hello
● ncc@nccs-air hello % ls
  src
● ncc@nccs-air hello % rustc src/hello.rs

```

mkdir -p hello/src command အားသုံးပြီး hello directory အောက်တွင် src
ဆိုသည့် directory တစ်ခု ထပ်ဆောက်ထားပါသည်။ ထို့နောက် hello.rs file ကို ထို့ src
directory အောက်သို့ ရွှေ့လိုသည့် အတွက် mv hello.rs hello/src ဆိုသည့် command ကို
ရေးလိုက်ပါသည်။

```
● ncc@nccs-air hello % tree
```

```

├─ hello
└─ src
    └─ hello.rs

```

2 directories, 2 files

စာရေးသူ အနေဖြင့် အထက်ပါ ပုံတွင် tree command ကို သုံးပြီး file directory ကို ဖော်ပြ ပေးထားပါသည်။ tree command ကိုအသုံးပြုလိုလျှင် tree ကို install လုပ် ပေးရန် လိုအပ်ပါ သေးသည်။

Using Cargo

```

● ncc@nccs-air hello % cd ..
● ncc@nccs-air rs % ls
  hello
● ncc@nccs-air rs % rm -rf hello
● ncc@nccs-air rs % ls
○ ncc@nccs-air rs % █

```

Cargo ကို သုံးပြီး hello ဆိုသည့် project file ကို တည်ဆောက်မှာ ဖြစ်တဲ့ အတွက် မူရင်း ရှိနေသော် hello file ကို ဖျက်ပါမည်။ rm -rf hello ဆိုသည့် command ကို သုံးထားပါသည်။ ပထမ ရေးထားသည့် cd command သည် directory change ရန် ဖြစ်ပြီး နောက်မှ .. သည် parent directory ကို ဆိုလိုခြင်း ဖြစ်သည်။ rm command သည် file တစ်ခု သို့မဟုတ် empty directory တစ်ခုကို ဖျက် မည်ဟု ဆိုလိုခြင်း ဖြစ်ပြီး r (recursive) directory ထဲမှ ရှိသမျှ files အားလုံးကို ဖျက်မည် ဟု ဆိုလိုခြင်း ဖြစ်ပြီး f (force) တက်လာမျှ error အားလုံးကို force လုပ်ပြီး ကျော်သွားမည် ဟု ဆိုလိုခြင်း ဖြစ်သည်။

- ncc@nccs-air rs % cargo new hello
Created binary (application) `hello` package
- ncc@nccs-air rs % cd hello
- ncc@nccs-air hello % tree

```

├─ Cargo.toml
└─ src
    └─ main.rs

```

2 directories, 2 files

Cargo new hello command သည် hello ဆိုသည့် rust project တစ်ခုကို တည်ဆောက် ပေးမှာ ဖြစ်ပါတယ်။ cargo သည် rust package manager ဖြစ်ပါတယ်။ cargo command သည် rust project တစ်ခု တည်ဆောက်ရန် မရှိ မဖြစ်လိုအပ်သည့် အရာ တစ်ခုတော့ မဟုတ်ပါ။ သို့သော် Rust community ထဲမှာ ရှိတဲ့သူ အားလုံး နီးပါ သုံးကြပါတယ်။ project အတွက် directory တွေကို မိမိ ကိုယ်တိုင်လည်း ဖန်တီးလို့ ရပါတယ်။ new သည် cargo package အသစ် တစ်ခုကို ဖန်တီး ပေးခြင်း ဖြစ်ပါတယ်။

Cargo ကို သုံးခြင်းဖြင့် src directory ထဲတွင် main.rs ဆိုသည့် rust file တစ်ခုကို ဖန်တီး ပေးထားပါတယ် ။

- ncc@nccs-air hello % cat src/main.rs
fn main() {
 println!("Hello, world!");
}
- ncc@nccs-air hello %

cat သည် concatenate ကိုဆိုလိုခြင်း ဖြစ်ပြီး သူ့နောက်မှ ရှိတဲ့ file ထဲမှ အချက်လက်များကို ဖော်ပြ ပေးပါတယ်။ rustc ကိုသုံးပြီးတော့ပဲ rust program ကို compile လုပ်လို့ ရတာ မဟုတ်ပါဘူး အကယ်၍ မြန်မြန် ဆန် run လိုလျှင် အောက်ပါ အတိုင်းလည်း run လို့ရနိုင်ပါတယ်။

- ncc@nccs-air hello % cargo run
Compiling hello v0.1.0 (/Users/ncc/myCode/rs/hello)
Finished dev [unoptimized + debuginfo] target(s) in 1.10s
Running `target/debug/hello`
Hello, world!
- ncc@nccs-air hello %

ပထမ လိုင်းသုံးခုသည် cargo မှ မည်သည့် အရာများ လုပ်လိုက်ကြောင်းကို ဆိုလိုခြင်း ဖြစ်ပြီး နောက်ဆုံး လိုင်းကတော့ program ရဲ့ output ဖြစ်ပါတယ်။ အကယ်၍ Compiling , Finished , Running စသည့် သုံးကြောင်းအား မပေါ်လိုပါက အောက်ပါ

အတိုင်းလည်း run လို့ရပါသေးသည်။

```
● ncc@nccs-air hello % cargo run --quiet
Hello, world!
○ ncc@nccs-air hello %
```

cargo နဲ့ ပတ်သက်ပြီး အသေးစိတ် သိရှိလိုပါက အောက်ပါ အတိုင်း လည်း သိရှိနိုင်ပါ သေးသည်။

ncc@nccs-air ~ % cargo

Rust's package manager

Usage: cargo [+toolchain] [OPTIONS] [COMMAND]

Options:

- V, --version Print version info and exit
- list List installed commands
- explain <CODE> Run `rustc --explain CODE`
- v, --verbose... Use verbose output (-vv very verbose/build.rs output)
- q, --quiet Do not print cargo log messages
- color <WHEN> Coloring: auto, always, never
- frozen Require Cargo.lock and cache are up to date
- locked Require Cargo.lock is up to date
- offline Run without accessing the network
- config <KEY=VALUE> Override a configuration value
- Z <FLAG> Unstable (nightly-only) flags to Cargo, see 'cargo -Z help' for details
- h, --help Print help information

Some common cargo commands are (see all commands with --list):

- build, b Compile the current package
- check, c Analyze the current package and report errors, but don't build object files
- clean Remove the target directory
- doc, d Build this package's and its dependencies' documentation
- new Create a new cargo package
- init Create a new cargo package in an existing directory

add Add dependencies to a manifest file
remove Remove dependencies from a manifest file
run, r Run a binary or example of the local package
test, t Run the tests
bench Run the benchmarks
update Update dependencies listed in Cargo.lock
search Search registry for crates
publish Package and upload this package to the registry
install Install a Rust binary. Default location is \$HOME/.cargo/bin
uninstall Uninstall a Rust binary

See 'cargo help <command>' for more information on a specific command.

ncc@nccs-air ~ %

```

● ncc@nccs-air hello % ls
  Cargo.lock      Cargo.toml      src              target
● ncc@nccs-air hello % ./target/debug/hello
  Hello, world!
○ ncc@nccs-air hello % █
  
```

hello directory ကို ကြည့်လိုက်ပါ အောက်ပါ အတိုင်း အသေးစိတ် ကို မြင်ရပါမည်။ ထို target/debug directory ထဲတွင် hello ဆိုသည့် executable file ရှိနေပါသည်။

ထို့နောက် ./target/debug/hello ဆိုပြီး run ကြည့်ပါက Hello, World! ဆိုသည့် output ထွက်နေသေးသည်ကို တွေ့ရပါမည်။ ထို့ကြောင့် cargo သည် program ရေးသူတို့ အတိအကျ ညွှန်းစရာမ လိုပဲ managed လုပ် ပေးနိုင်ပါတယ်။ ထို အကြောင်းအရာများ ပိုမို ရှင်းလင်းရန် Cargo.toml file အား ကြည့်ရန် လိုအပ်ပါသည်။ ထို toml file သည် project အတွက် configuration file ဖြစ်ပါတယ်။ toml file သည် package တစ်ခုခြင်းစီ အတွက် ဖြစ်ပြီး manifest လို့လည်း ခေါ်ပါတယ်။ TOML သည် Tom's obvious Minimal Language ကို ဆိုလိုခြင်း ဖြစ်ပြီး ထို file ထဲတွင် package များကို compile လုပ်ရန် metadata များ ပါဝင်ပါတယ်။

```

ncc@nccs-air hello % cat Cargo.toml
[package]
name = "hello"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]

```

ပထမဆုံး ဖြစ်သည့် name သည် Cargo နဲ့ project ဖန်တီးတုန်းက ထည့်ပေးလိုက်သော နံမည် ဖြစ်ပြီး ထို name သည် executable file ရဲ့ name လည်း ဖြစ်ပါတယ်။

ဒုတိယ တစ်ခုကတော့ program ရဲ့ version ဖြစ်ပါတယ်။

တတိယ တစ်ခုကတော့ rust ရဲ့ edition ဖြစ်ပါတယ်။

Running Integrations Tests

Test code လုပ်ရန် ပထမ ဦးစွာ tests ဆိုသည့် ဆိုသည့် တစ်ခု တည်ဆောက်ပါမည် ။ ထို tests directory ထဲတွင်လည်း cli.rs ဆိုသည့် rust file တစ်ခု တည်ဆောက်ထားပါမည်။

```

ncc@nccs-air hello % tree -L 2
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
├── target
│   ├── CACHEDIR.TAG
│   └── debug
└── tests
    └── cli.rs

5 directories, 5 files

```

ထို့နောက် cli.rs file ထဲတွင် အောက်ပါ code များကို ရေးသား ပါမည်။

```

hello.rs cli.rs U
hello > tests > cli.rs > ...
1  #[test]
2
3  fn works(){
4      |    assert!(ture);
5  }
6

```

`#[test]` သည် attribute တစ်ခု ဖြစ်ပြီး Rust အား ယခု function အား testing လုပ်ချိန်တွင် run ရန် ပြောခြင်း ဖြစ်သည်။ `assert!` သည် macro ဖြစ်ပြီး Boolean expression true ကိုထည့်ထားပါသည်။

ထို့နောက် file အား save ပြီး `cargo test` ဟု run ကြည့်ပါ။

```

ncc@nccs-air hello % cargo test
Compiling hello v0.1.0 (/Users/ncc/myCode/rs/hello)
Finished test [unoptimized + debuginfo] target(s) in 0.76s
Running unittests src/main.rs (target/debug/deps/hello-15694aad092babc2)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Running tests/cli.rs (target/debug/deps/cli-3564f46a4e249c97)

running 1 test
test works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

```

ထို့နောက် `assert!` ထဲမှ boolean expression အား false ဟု ပြောင်းကြည့်ပါ test works Fail ကြောင်းကို မြင်တွေ့ရမည်။

```

ncc@nccs-air hello % cargo test
Compiling hello v0.1.0 (/Users/ncc/myCode/rs/hello)
Finished test [unoptimized + debuginfo] target(s) in 0.67s
Running unittests src/main.rs (target/debug/deps/hello-15694aad092bab2)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Running tests/cli.rs (target/debug/deps/cli-3564f46a4e249c97)

running 1 test
test works ... FAILED

failures:

---- works stdout ----
thread 'works' panicked at 'assertion failed: false', tests/cli.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
  works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

error: test failed, to rerun pass `--test cli`

```

တကယ်တော့ assert ကိုသုံးပြီး true or false input များထည့်ခြင်းဟာ အသုံးမဝင်ပါဘူး။ ထို့ကြောင့် ထို function အား ထုတ်ပြီး command တစ်ခု အား execute လုပ်ရန် std::process::Command ကို အသုံးပြုပါမည်။

```

1 use std::process::Command;
2 #[test]
3
4 fn runs(){
5     let mut cmd = Command::new("ls");
6     let res = cmd.output();
7     assert!(res.is_ok());
8 }

```

- Line 1 သည် new command တစ်ခု ဖန်တီးရန် အတွက် std::process::Command ဆိုသည့် structure ကို import လုပ်ခြင်း ဖြစ်ပါတယ်။
- Line 2 ကတော့ ls ဆိုသည့် command ကို run ရန် ဖန်တီးခြင်းဖြစ်ပြီး mut ကတော့ variable ကို mutable လုပ်ရန် ဖြစ်ပါတယ်။
- Line 3 ကတော့ command ကို run ပြီး result တွင် သိမ်းထားရန် ဖြစ်သည်။
- Line 4 ကတော့ result value ကို Ok Value ဖြစ်သလား ဆိုတာစစ်ဆေးခြင်း ဖြစ်ပါတယ်။
ပုံမှန် အားဖြင့် rust variables များသည် immutable ဖြစ်ပြီး သူတို့ရဲ့ value

တွေကို ချိန်းလို့ မရပါဘူး။

အထက်ပါ အတိုင်း cargo test ဟု run ကြည့်မည် ဆိုလျှင် ok ဆိုသည့် value ပြန်ရကြောင်းကို တွေ့ရမည်။ ထို့ကြောင့် command နေရာတွင် hello ဆိုသည့် စာသား အား ထည့်ပြီး run ကြည့်ပါက FAILED ဖြစ်ကြောင်း တွေ့ရမည်။

```
1 use std::process::Command;
2 #[test]
3
4 fn runs(){
5     let mut cmd = Command::new("hello");
6     let res = cmd.output();
7     assert!(res.is_ok());
8 }
```

```
failures:
  runs

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s

error: test failed, to rerun pass `--test cli`
```

Touch to Rust

ပထမဆုံး အနေဖြင့် input ဆိုသည့် package တစ်ခု တည်ဆောက်ပါမည်။ ထို့နောက် run ကြည့်ပါမည်။ cargo new input ဆိုသည့် command ကို အသုံးပြုပါသည်။

```
● ncc@nccs-air rs % cargo new input
    Created binary (application) `input` package
● ncc@nccs-air rs % ls
hello          hello.rs      input
```

စာရေးသူ အနေဖြင့် rs ဆိုသည့် folder ထဲတွင် ယခင်က ရေးခဲ့သော hello ဆိုသည့် package နှင့် ယခု input ဆိုသည့် package နှစ်ခု ဖန်တီးထားပါသည်။ input folder ထဲမှ src ထဲမှ main.rs ထဲတွင် အောက်ပါ code များကို ရေးသား ပါမည်။

```

main.rs U
input > src > main.rs > ...
1  use std::env;
2
3  fn main(){
4      let name = env::args().skip(1).next();
5
6      match name{
7          Some(n) => println!("Hi Rust Who r u!{}",n),
8          None => panic!("Didn't get anythings?")
9      }
10
11 }

```

ထို့နောက် input ဆိုသည့် folder ထဲသို့ ဝင်ပြီး project ကို run ကြည့်ပါမယ်။

```

ncc@nccs-air input % cargo run winhtut
Finished dev [unoptimized + debuginfo] target(s) in 0.00s
Running `target/debug/input winhtut`
Hi Rust Who r u!winhtut
ncc@nccs-air input % cargo run --quiet winhtut
Hi Rust Who r u!winhtut
ncc@nccs-air input %

```

- Line 1 တွင် env ဆိုသည့် module ကို import လုပ်ထားပြီး ထို module သည် crates ဆိုသည့် std ထဲမှ ဖြစ်သည်။ std သည် rust ရဲ့ standard library ဖြစ်ပါသည်။
- Line 3 မှာ တော့ပုံမှန် အတိုင်း main function ကို ကြေငြာ ထားပါသည်။
- Line 4 မှာတော့ args() ဆိုတဲ့ function ကို env module ထဲမှ ခေါ်ထားပြီး သူက return အနေနဲ့ program ထဲသို့ ထည့်ပေးလိုက်တဲ့ arguments တွေကို အစဉ်လိုက် ပြန်ပေးပါတယ်။ ထို arguments များထဲတွင် ပထမအဆုံးအနေဖြင့် ပါဝင်လာမှာက program name ဖြစ်တဲ့ အတွက် ထို argument ကို ကျော်သွားရန် skip ကို သုံးထားပါတယ်။ skip ထဲတွင်မှ မိမိတို့ ကျော်မည့် argument ပမာဏကို ထည့်ပေးရပါတယ် စာရေးသူ အနေဖြင့် program name တစ်ခုတည်းကိုသာ ကျော်လိုသောကြောင့် 1 ဟု ရေးထားခြင်း ဖြစ်ပါတယ်။ အကယ်၍ skip(2) ဟု ပြောင်းလိုက်ပါက စာရေးသူတို့ ထည့်ပေးလိုက်သော winhtut ဆိုသည့် name ကို သိမှာ မဟုတ်တော့ပါဘူး။

```

ncc@nccs-air input % cargo run --quiet winhtut
thread 'main' panicked at 'Didn't get anythings?', src/main.rs:8:17
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
ncc@nccs-air input %

```

- skip (2) ဟု ပြောင်းထားပြီး အောက်ပါ အတိုင်း argument အပို တစ်ခု ထပ်ထည့်ကြည့်ပါက program မှ ထို argument ကို ပြန်ပေးမှာဖြစ်ပါတယ်။

```
● ncc@nccs-air input % cargo run --quiet winhtut ncc
Hi Rust Who r u!ncc
○ ncc@nccs-air input %
```

- Line 4 မှာပင် next() ကို ထပ်ခေါ်ထားပြီး ယင်း next သည် နောက်မှ ထပ်ထည့်လိုက်သော arguments များကို ရယူပြီး enum type ဖြင့် ပြန်ပေးမှာ ဖြစ်ပါတယ်။ (enum ကို နောက်ပိုင်းမှာ အသေးစိတ် ရှင်း လင်း သွားပါမည်) next မှ enum type ဖြင့် ပြန်ပေးသည်ကို Option လို့လည်း ခေါ်ပါတယ်။ ထို return ပြန်ပေးသော အရာသည် value တစ်ခုလည်း ဖြစ်နိုင်သလို None value လည်း ဖြစ်နိုင်ပါတယ်။
- Line 6 မှာတော့ match expression ကို သုံးထားပြီး ထို match expression သည် name variable ထဲတွင် တစ်ခုခု ရှိနေသလားဆိုတာကို Some(n) ဖြင့် စစ်ဆေးပါတယ်။ အကယ်၍ Some(n) ထဲတွင် တစ်ခုခု ရှိနေပါက println ဆိုတာကို ဆက်ခေါ်ပါတယ်။ ထိုသို့ ပြန်ခေါ်ရာတွင် string variable n ကိုပါ ပြန် ဖော်ပြ ပေးပါတယ်။ println သည် function မဟုတ်ပဲ macro ဖြစ်ကြောင်းကို ရှေ့သင်ခန်းစာတွင် ရှင်း ပြခဲ့ပြီး ဖြစ်သလို macro များသည် နောက်က ! ဖြင့် လိုက်ကြပါတယ်။
- Line 8 မှာတော့ return သည် enum ဖြစ်နေခဲ့မည် ဆိုလျှင် panic ဆိုသည့် macro ကို ပြန်ခေါ်ပြီး program ကို error message ထုတ်ပြပြီး ရပ် ပေးပါတယ်။
- println! macro ထဲတွင် placeholders “{ }” များကိုလည်း ထားနိုင်ပြီး ထို placeholders များတွင် မိမိတို့ စိတ်တိုင်းကျ position number များ ပေးပြီးလည်း သတ်မှတ်နိုင်သည်။ ထို braces များနေရာတွင် ဖော်ပြ သည့် strings များကို format strings ဟု ခေါ်ပြီး { } ထဲတွင် ဖော်ပြ သည့် အရာများကိုတော့ format specifiers ဟု ခေါ်ပါတယ်။ primitives များ ကို ဖော်ပြ ရာတွင် “{ }” များကို သုံးနိုင်ပြီး အခြားသော types များ အတွက်ဆိုလျှင်တော့ “{:?}” ကို သုံးပါတယ်။

Primitive types

Rust programming မှာ အောက်ပါ built-in primitive types များ ပါဝင်ပါတယ်။

- bool : booleans type များအတွက် သုံးပြီး true or false တစ်ခုခု အတွက်သုံးပါတယ်။
let c = true ; // ပုံမှန် အတိုင်း ကြေငြာခြင်း ဖြစ်ပါတယ်။

let y: bool= false // explicit type ပုံစံဖြင့် ကြေငြာခြင်း ဖြစ်ပြီး y ကို boolean type ဖြင့်သာ အသုံးပြုမည်ဟု ဆိုလိုခြင်း ဖြစ်ပါတယ်။

- char : စကားလုံး တစ်လုံးခြင်းစီ အတွက် အသုံးပြုပါတယ် ဥပမာ n ။

let c = 'z'; // c ကို ပုံမှန်အတိုင်း variable တစ်လုံး အနေဖြင့်သာ ကြေငြာခြင်း ဖြစ်ပြီး မည်သည့် type အသုံးပြုမည်ဟု ကြေငြာထားခြင်း မရှိပါ။

let z: char = 'Z'; // z ကို explicit type ပုံစံဖြင့် ကြေငြာထားခြင်း ဖြစ်ပြီး character type သာ သုံးမည် ဟု ကြေငြာ ထားခြင်း ဖြစ်ပါတယ်။

- Integer types : interger types များကိုတော့ bit အနဲအများပေါ် မူတည်ပြီး များစွာ သတ်မှတ် ပေးထားပါတယ်။ Rust မှာတော့ 128 bits ထိ သတ်မှတ် ပေးထားပါတယ်။

Signed	Unsigned
i8	u8
i16	u16
i32	u32
i64	u32
i128	u128

- isize : သူကတော့ pointer-sized signed integer type ဖြစ်ပြီး 32-bit CPU များပေါ်တွင် i32 နဲ့ ညီမျှ ပြီး 64-bit CPU များတွင်တော့ i64 နဲ့ ညီမျှပါတယ်။
- usize : usize သည်လည်း pointer-sized ဖြစ်ပြီး unsigned integer type အတွက် ဖြစ်ပါတယ် 32-bit CPU များပေါ်တွင် i32 နဲ့ ညီမျှ ပြီး 64-bit CPU များတွင်တော့ i64 နဲ့ ညီမျှပါတယ်။
- f32 : floating point ကိုမှ 32-bit နဲ့ ဖြစ်ပြီး IEEE 754 floating point standard ကို implements လုပ်ထားပါတယ်။
- f64 : 64-bit floating point ဖြစ်ပါတယ်။

အကယ်၍ let x = 1.1 ; ဟု ကြေငြာမည် ဆိုလျှင်တော့ modern cpu များတွင် default အနေဖြင့် 64 bit precision ကိုသာ အသုံးပြုသွားမှာ ဖြစ်ပါတယ်။

let x = 2.0; // f64

let y: f32 = 3.3; //f32

- [T ; N] : fixed-size array များကို ကြေငြာ ရာတွင် သုံးပြီး T သည် element type များ အတွက် ဖြစ်ပြီး N သည် non-negative ဖြစ်ပါတယ်။

Example: let _: [u8,3] = [1,2,3,]; u8 သည် element များရဲ့ type ဖြစ်ပြီး

နောက်က 3 ကတော့ သူ့ရဲ့ size ဖြစ်ပါတယ်။

- `str : &str` ပုံစံဖြင့် အဓိက အသုံးပြုပါတယ်။

Example: `_: [&str, 3] = ["1", "2", "3"];`

- `fn(i32) -> i32` : function တစ်ခုကိုကြေငြာခြင်း ဖြစ်ပြီး ထို function သည် `i32` ကို ယူ မည်ဖြစ်ပြီး return အနေဖြင့်လည်း `i32` ကို ပြန်မည်ဟု ဆိုလိုခြင်း ဖြစ်သည်။

Compound Types

Compound types တွေဆိုတာ values အများကြီးကို type တစ်ခုတည်း အနေဖြင့် အသုံးပြုခြင်း ဖြစ်ပါတယ်။ Rust programming မှာ primitive compound type နှစ်ခုရှိပြီး tuples and arrays တို့ ဖြစ်ပါတယ်။

- Tuple Type

Tuple ဆိုတာက မတူညီတဲ့ data အများကြီးကို compound type တစ်ခုတည်း အဖြစ် စုဆည်း အသုံးပြုခြင်း ဖြစ်ပါတယ်။ Tuples တွေရဲ့ size ဟာ အသေဖြစ်ပြီး တစ်ခါ ကြေငြာ ပြီးတာနဲ့ ထို size ထပ် ကြီးလာလို့ မရသလို ပြန်လျော့လို့လည်း မရပါဘူး။ tuple types တွေကို ဖန်တီး ရာတွင် comma-separated ပုံစံများဖြင့် () “parentheses” ကြေငြာပါတယ်။ အကယ်၍ explicit ပုံစံမျိုးဖြင့် ကြေငြာလိုလျှင်တော့ အောက်ပါ အတိုင်း ကြေငြာပါတယ်။

`let tup: (i32 , f64 , u8 , f32) = (100 , 1.1 , 1 , 2.2);`

```
1  fn main(){
2      let tup = ( 100 , 2.2 , "ncc_rust");
3      let (x , y ,z) = tup;
4
5      println!("The value of y is: {}",z);
6  }
```

အထက်ပါ program line 2 တွင် မတူညီတဲ့ data သုံးခုကို ကြေငြာထားပြီး tup variable တစ်ခုတည်းသို့ bind လုပ်ထားပါတယ်။ ထို့နောက် line 3 တွင် tup မှ x , y ,z အဖြစ် value 3 ခုကို ပြန်ခွဲထုတ်ပါတယ် ထိုသို့ ပြုလုပ်ခြင်းကို destructuring ဟု ခေါ်ပါတယ်။ အောက်ပါ program တွင် rust tuple မှ indices များဖြင့် ပြန်ထုတ် ပုံကို ဖော်ပြ ထားပါတယ်။

```

1  fn main(){
2
3      let x:(i32 , f64 , u8 , &str) = (10 , 1.1 , 100 , "rust_string");
4      let zero = x.0;
5      let one = x.1;
6      let two = x.2;
7      let three = x.3;
8
9      println!("{}", zero , one , two , three);
10 }

```

အထက်ပါ ပုံအတိုင်း ရေးမည် ဆိုလျှင်တော့ data များနှင့် data type များကို သတိထားရမှာ ဖြစ်ပါတယ်။ အကယ်၍ data type မှာ 5 ခု ကြေငြာထားပြီး data 4 ခု သာ ရေးထားမည် ဆိုလျှင်တော့ အောက်ပါ ပုံအတိုင်း error တက်မှာ ဖြစ်ပါတယ်။

```

--> 4-usingdot.rs:3:42
3 |     let x:(i32 , f64 , u8 , &str , u8) = (10 , 1.1 , 100 , "rust_string");
    |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected a tuple with 5 elements, found one with 4 elements

```

Array Type

Tuple ကို အသုံးမပြုလိုပဲ values များစွာကို သိမ်းလိုပါက array ကို အသုံးပြု နိုင်ပါသေးတယ်။ array နဲ့ tuple ရဲ့ မတူညီသော အချက်သည် array ထဲတွင် သိမ်းသော data များသည် c and c++ programming မှာကဲ့သို့ same type များ ဖြစ်ရပါမည်။ rust array များသည် fixed length ဖြစ်ပြီး vector များကဲ့သို့ ထပ်တိုးတာတွေ လျော့တာတွေ လုပ်လို့ မရပါဘူး။ rust vector များ အကြောင်းကိုတော့ နောက်ပိုင်းမှာ အသေးစိတ် ဆွေးနွေး ပေးပါမည်။

```

1  fn main(){
2
3      let _a = [1,2,3,4,5];
4
5      let _arrstr=["name1","name2","name3","name4"];
6
7      // let c = [2,2,2,2,2];
8      let _c = [2;5];
9
10     // default value with data type
11     let _my_number: [i32;5]=[3;5];
12
13     //accessing data from array;
14     let _first = _c[0];
15     let _sec = _c[1];
16
17     println!("{:?}",_a);
18     println!("{:?}",_arrstr);
19     println!("{:?}",_my_number);
20
21 }

```

Rust တွင် array များအား ဖန်တီးရန် အခြေခံ အားဖြင့် နည်းလမ်းသုံးမျိုးကို အသုံးပြု နိုင်ပါတယ်။ line 3 တွင် ကြေငြာထားသည့် array အား data type မပါပဲ ဖန်တီးသည့် နည်းလမ်း၊ line 8 တွင်ကြေငြာထားသည့် default value နည်းလမ်း နှင့် line 11 တွင် ကြေငြာထားသည့် default value များကို data type များဖြင့် ကြေငြာသည့် နည်းလမ်း ဖြစ်ပါတယ်။

Rust array မှ data များကို access လုပ်ရာတွင် အခြားသော programming များမှ နည်းလမ်း အတိုင်း array indexes များကို သုံးပြီး access လုပ်နိုင်ပါတယ်။ line number 14 and 15 တွင် ဖော်ပြထားပါတယ်။

Output

```

[1, 2, 3, 4, 5]
["name1", "name2", "name3", "name4"]
[3, 3, 3, 3, 3]

```

Mutable Array in Rust

Rust programming မှာ array သည် immutable ဖြစ်ပါတယ် ဆိုလိုချင်တာ က တစ်ခေါ်ဖန်တီး ပြီးသည့်နှင့် သူထဲမှ data များကို ပြန်ပြောင်းလို့ မရနိုင်တော့ပါဘူး ထို့ကြောင့် ပြန်ပြောင်းလိုသော အခါများတွင် mut ဆို keyword ကို သုံးပြီး mutable အဖြစ် ရေးနိုင်သလို ပြန် မပြောင်းလိုသော အခါများတွင်တော့ ပုံမှန်အတိုင်း let ကိုသုံးပြီးသာ ကြေငြာနိုင်ပါတယ်။

```
1 fn main(){
2   let _num: [i32; 5]=[1,2,3,4,5];
3
4   _num[0]=100;
5 }
```

အကယ်၍ အထက်ပါ အတိုင်း ရေးခဲ့မည် ဆိုလျှင်တော့ line 4 တွင် error တက်မှာ ဖြစ်ပါတယ်။ အဘယ်ကြောင့်ဆိုသော် immutable ဖြင့် ရေးထားပြီး data များ ထပ်မံ update လုပ်သောကြောင့် ဖြစ်သည်။ array data များအား update လုပ်လိုသော အခါတွင် အောက်တွင် ဖော်ပြထားသည့် line 4 မှ အတိုင်း ရေးနိုင်ပါတယ်။

```
1 fn main(){
2   let _num: [i32; 5]=[1,2,3,4,5];
3
4   let mut _num2: [i32;3]=[10,20,30];
5
6   _num2[0] = 100;
7
8   println!("{}",_num2[0]);
9 }
```

Accessing array data using loop

Array ထဲတွင် ရှိသော data များအား အခြား programming များ အတိုင်းပင် loop ကို သုံးပြီး ထုတ်နိုင်ပါတယ်။

```

1  fn main(){
2
3      let _data = ["email","pass","amount"];
4
5      for i in 0..3{
6          println!("index:{}->data {}",i,_data[i]);
7      }
8  }

```

Looping အကြောင်းကိုတော့ control structure lesson တွင် အသေးစိတ် ဖော်ပြပေးပါမည်။ ယခု သင်ခန်းစမှာတော့ looping သည် 0 မှ 2 ထိ သုံးကြိမ် အလုပ်လုပ်သွားပါသည်။ i တန်ဖိုးသည် 0 , 1 , 2 စသည်ဖြင့် အစဉ်လိုက် ပေါ်မှာ ဖြစ်ပါတယ်။

Slice in Rust

Slice သည် data များ စုဝေးထားသည့် ထဲမှ မိမိတို့ လိုချင်သော အပိုင်းများအား အစဉ်လိုက် သီးသန့် ခွဲထုတ်ပြီး ယူခြင်း ဖြစ်ပါတယ်။ ဥပမာ rust programmingတွင် arrays , vectors and strings တို့ ဖြစ်ပါတယ်။ slice သည် reference ကိုသာ store လုပ်ခြင်းဖြစ်ပြီး actual value တော့ မဟုတ်ပါဘူး ထိုအကြောင်းကို pointer သင်ခန်းစတွင် အသေးစိတ် ဆွေးနွေးပါမည်။

```

1  fn main(){
2
3
4      let _values = [1,2,3,4,5];
5
6      let slice = &_values[1..3];
7
8      println!("array's values {:?}",_values);
9      println!("slice's values {:?}",slice);
10 }

```

အထက်ပါ သင်ခန်းစတွင် _values ဆိုသည့် array ထဲမှ &_values ဟုရေးပြီး 1 နောက်မှ value ဖြစ်သည့် 2 မှ စတင်ကာ 3 ထိ ယူမည်ဟု ဆိုလိုခြင်း ဖြစ်ပါသည်။

```

• ncc@192 rs % rustc 8-slicerust.rs
• ncc@192 rs % ./8-slicerust
array's values :[1, 2, 3, 4, 5]
slice's values :[2, 3]

```

- Values များကို အစမှ စတင်ပြီး မိမိ တို့ လိုအပ်သော နေရာထိ ရယူ လိုပါက အောက်ပါ အတိုင်း ရေးနိုင်ပါတယ်။

```
let slice = &_values[..3];
```

- အကယ်၍ မိမိတို့ စလိုသော နေရာမှ စတင်ပြီးအ ဆုံးထိ ရယူ လိုပါက အောက်ပါ အတိုင်း ရေးနိုင်ပါတယ်။

```
let slice = &_values[2..];
```

- ထို့ပြင် အစမှ အဆုံးထိ ရယူလိုပါကလည်း &_values[..]; ဟု ရေးပြီး ရယူ နိုင်ပါတယ်။

```
1 fn main(){
2
3
4     let mut values = [1,2,3,4,5];
5     println!("array's values {:?}", values);
6
7     let slice = &mut values[2..];
8
9     slice[2]=100;
10    println!("slice's values {:?}", slice);
11 }
```

အထက်ပါ program မှာတော့ mutable array ထဲမှ slice ပြန်လုပ်ထားခြင်း ဖြစ်ပါတယ်။ ထိုသို့ slice လုပ်ရာတွင် mutable ပုံစံဖြင့် ပြန်လည် ရယူလိုပါက slice လုပ်ရာတွင် & နောက်၌ mut ဆိုသည့် keyword ကို ထည့်ပေးရပါမည်။ line 7 တွင်ရေးထားသော slice variable သည်လည်း mutable ဖြစ်လာသည့် အတွက် line 9 တွင် value အသစ်အား ထပ်မံ assign လုပ်လို့ရခြင်း ဖြစ်ပါတယ်။

```
● ncc@192 rs % rustc 8-slicerust.rs
● ncc@192 rs % ./8-slicerust
array's values : [1, 2, 3, 4, 5]
slice's values : [3, 4, 100]
```

Declaring variables and immutability


```

1  fn main(){
2
3      let target = "ncc";
4      let mut testMut = "are u ok";
5      println!("{}",target,testMut);
6
7      testMut = "Adding New Data";
8      target = "test for target";
9
10     println!("{}",target , testMut);
11 }

```

ပထမဆုံး အနေဖြင့် ဖော်ပြပါ အတိုင်း program တစ်ပုဒ်ကို စတင် ရေးသား ပါမယ်။ rust programming တွင် variable များကို let keyword ကိုသုံးပြီး ကြေငြာပါတယ်။ ထို့ပြင် variables များအား mutable or immutable အဖြစ်လည်း ကြေငြာ နိုင်ပါသေးတယ်။ အကယ်၍ variables များအား ပုံမှန် အတိုင်း mut keyword ကို မသုံးပဲ ကြေငြာမည် ဆိုလျှင် variables များသည် immutable ဖြစ်နေမှာပါ။ စာရေးသူ အနေဖြင့် ဖော်ပြပါ program အား compile လုပ်ကြည့်ပါက အောက်ပါ error များကို ရရှိမည် ဖြစ်သည်။

```

ncc@nccs-air rs % rustc variable.rs
error[E0384]: cannot assign twice to immutable variable `target`
--> variable.rs:9:5
4 |     let target = "ncc";
   |     -----
   |     |
   |     first assignment to `target`
   |     help: consider making this binding mutable: `mut target`
9 |     target = "test for target";
   |     ~~~~~~ cannot assign twice to immutable variable
error: aborting due to previous error

For more information about this error, try `rustc --explain E0384`.

```

အဘယ်ကြောင့် ထိုကဲ့သို့ error တက်ရသနည်း ဆိုလျှင် target သည် let ဖြင့်သာ ရှိရှိ ကြေငြာထားပြီး immutable variable ဖြစ်နေသည် ထို့ကြောင့် နောက် ထပ် value အသစ်များ assign အလုပ် မခံပါ။ အကယ်၍ ထိုသို့ ထပ် assign လုပ်မည်ဆိုလျှင် error တက်မှာ ဖြစ်ပါတယ်။

Structure in Rust

Rust structure သည်လည်း c programming မှာကဲ့သို့ပင် user-defined types ပုံစံဖြင့် မတူညီတဲ့ data များကို store လုပ်ရန် အသုံးပြုပါတယ်။ ဥပမာ user

တစ်ယောက်ရဲ့ အချက်အလက်များကို သိမ်းဆည်းမည်ဆိုပါစို့ name , email , password , phone , address စသည့် အချက်အလက်များကို structure ထဲတွင် ထည့်ထားပြီး ထို structure အား users မြောက်များစွာ အတွက် အသုံးပြု နိုင်ပါတယ်။

```

1  fn main(){
2
3      struct User{
4          name: String,
5          email: String,
6          password: String,
7          age : u8,
8          address: String,
9      }
10
11     let u1 = User{
12         name: String::from("NCC"),
13         email: String::from("example@gmail.com"),
14         password:String::from("123ncc"),
15         age:20,
16         address:String::from("PyinOoLwin")
17     };
18
19     println!(" User name = {}",u1.name);
20 }
```

အထက်ပါ program မှာတော့ User ဆိုတဲ့ structure တစ်ခုကိုတည်ဆောက် ထားပြီး ထို structre ထဲတွင် data အချို့ကို ထည့်ထားပါတယ်။ data တစ်ခုပြီးလျှင် တစ်ခု နောက်မှ comma separated , ပုံစံဖြင့် ရေးရပါတယ်။ ရှေ့တွင် variable name ကို ရေးပြီး နောက်တွင် သူ့အတွက် type ကို ရေးပေးရပါတယ်။

Line 11 မှာတော့ User structure form ကိုသုံးပြီး u1 ဆိုသည့် variable ကို တည်ဆောက် လိုက်ပါတယ်။ ထို့နောက် line 12 မှာ တော့ name ထဲသို့ “NCC” ဆိုသည့် value ကို စထည့်ပေးပါတယ်။ Line 19 မှာတော့ ထည့်ပေးလိုက်သည့် data အား dot (.) ခံပြီး ပြန် လည် ထုတ်ပေးခြင်း ဖြစ်ပါတယ်။

Functions

Functions တွေဆိုတာ မိမိတို့ လုပ်ဆောင်လိုတဲ့ အချက်အလက်များကို စုဆည်း ရေးသားထားခြင်း ဖြစ်ပါတယ်။ အောက်ပါ program တွင် ဖော်ပြ ပေးထားပါတယ်။

```

1  fn adding(a: u32 , b: u32) -> u32{
2      return a+b;
3  }
4
5  fn main(){
6      let a: u32 = 10;
7      let b: u32 = 20;
8      let result: u32= adding(a,b);
9      println!("Result {}",result);
10 }

```

အထက်ပါ program တွင် adding ဆိုသည့် function အသစ်တစ်ခု တည်ဆောက်ထားပြီး ထို function ထဲတွင် parameter နှစ်ခု ဖြတ်ပါသည် ပထမ တစ်ခုသည်လည်း u32 type ဖြစ်ပြီး ဒုတိယ တစ်ခုသည်လည်း u32 ဖြစ်ပါသည်။ ထိုနည်းတူ return type သည်လည်း u32 ဖြစ်ပါသည် ။ အကယ်၍ function သည် မည်သည့် အရာမျှ return မပြန်လိုသော အခါတွင် ထို -> u32 အား ကျော်သွားနိုင်ပါသည်။ ထို့ပြင် line 6 and 7 ၌ ရေးသားထားသော u32 ဆိုသည့် type များကိုလည်း မိမိတို့ မရေးသားလိုပါက ချန်ထားနိုင်ပါတယ် သို့သော် ထိုသို့ types များ ရေးသားထားခြင်းဖြင့် readable ပိုဖြစ်စေသလို type များအားလက်ခံရာ၌ မှားယွင်း ခြင်းများကိုလည်း ကာကွယ် နိုင်ပါတယ်။

ENUM

Enum (enumeration) ဆိုတာ data type များကို user များ စိတ်ကြိုက် အသုံးပြု နိုင်ဖို့ ဖန်တီးရာတွင် သုံးပါတယ်။ enum ဆိုတဲ့ keyword ကိုသုံးကာ enum ကိုကြေငြာပြီး သူ့နောက်တွင်မှ name ကို ရေးပါတယ် ထို့နောက် {} brace တစ်စုံကို ရေးပါတယ်။ ထို {} brace ထဲတွင်မှ မိမိတို့ ဖန်တီးချင်တဲ့ types များကို ရေးသားပါတယ် ထို types များအား တစ်နည်းအားဖြင့် variants ဟုလည်း ခေါ်ပါတယ်။ ထို variants များအား ကြေငြာရာတွင် data များ တစ်ခါတည်း ထည့်ပြီး သို့မဟုတ် မထည့်ပဲလည်း ကြေငြာ နိုင်ပါတယ်။ data များဟု ဆိုရာတွင် primitive type , structs , tuple structs များ အပြင် enum ကိုတောင် ပြန်ထည့်နိုင်ပါတယ်။

```

1  enum Bank {
2      Register,
3      Login,
4      Loan(i64),
5      Suspend(String),
6      InterestRate {loan_rate :i64 , calcu_amount:i64},
7      CurrentAmount { money: i64},
8  }
9

```

အထက်ပါ ပုံတွင်တော့ Bank ဆိုသည့် enum တစ်ခုကို ကြေငြာ ထားခြင်း ဖြစ်ပြီး အသုံးပြုလိုသည့် data များကိုလည်း ကြေငြာ ထားပါတယ်။ အထက်ပါ program တွင်မူ line 6 and 7 တွင် structure နှစ်ခုကိုပါ ထည့်ထားပါတယ်။ ထို့ကြောင့် InterestRate and CurrentAmount ကို ခေါ်ရာ၌ သက်ဆိုင်ရာ structure ရဲ့ data များပါ ထည့်ပေးရမှာ ဖြစ်ပါတယ်။

```

10 fn bank_process(option: Bank) {
11     match option {
12         Bank::Register => println!("This is Register Sector"),
13         Bank::Login => println!("This is Login Sector"),
14         Bank::Loan(amount :i64 ) => println!("Your Loan Amount '{}'.", amount),
15         Bank::Suspend(status :String) => println!("Your account was '{}\'.", status),
16         Bank::InterestRate { loan_rate :i64 , calcu_amount :i64 } => {
17             println!("Your InterestRate loan_rate={}, calcu_amount={}.", loan_rate, calcu_amount);
18         },
19
20         Bank::CurrentAmount {money :i64 } =>{
21             println!("Your current amount = {}. ",money);
22         },
23     }
24 }

```

အထက်ပါ ပုံတွင်တော့ bank_process ဆိုသည့် function တစ်ခုကို ကြေငြာ ထားပြီး ထို function ထဲတွင် option ဆိုသည့် parameter တစ်ခု ဖြတ်ပါတယ်။ ထို option သည် Bank type ဖြစ်မှာ ဖြစ်သည့် အတွက် : colon နောက်တွင် Bank ဟု ရေးထားခြင်း ဖြစ်ပါတယ်။ ထို့နောက် match statement ကို သုံးပြီး option ကို စစ်ဆေးပါတယ်။ ဝင်လာသည့် option ပေါ်မူတည်ပြီး output များကို ပြန်ထုတ် ပေးသွားမှာ ဖြစ်ပါတယ်။

=> ကိုတော့ math တွင် အသုံးပြုပုံမှာ pattern and code ကြားတွင် အသုံးပြု ပါတယ်။ Bank::Register သည် pattern ဖြစ်ပြီး => သို့နောက်မှတော့ အလုပ်လုပ်မည့် code များကိုရေးပါတယ်။ math မှာတော့ expression တစ်ခုနှင့်တစ်ခုကြား comma နှင့် ခွဲခြားပါတယ်။

```

26 ▶ fn main() {
27
28     let bank_register = Bank::Register;
29     let bank_login = Bank::Login;
30     let bank_loan = Bank::Loan(10000);
31     let bank_suspend = Bank::Suspend("Suspend".to_owned());
32     let bank_interest_rate = Bank::InterestRate { loan_rate: 100 ,calcu_amount: 200 };
33     let bank_current_amount = Bank::CurrentAmount {money:300000000};
34
35     bank_process( option: bank_register);
36     bank_process( option: bank_login);
37     bank_process( option: bank_loan);
38     bank_process( option: bank_suspend);
39     bank_process( option: bank_interest_rate);
40     bank_process( option: bank_current_amount);
41 }

```

အထက်ပါ ပုံမှာတော့ main function ထဲမှ လုပ်ဆောင်ချက်များကို ဖော်ပြထားပါတယ်။ Enum ထဲမှ data များကို access လုပ်ရာတွင် :: notation ဖြင့် access လုပ်ပါတယ်။ line number 28 မှာတော့ Bank enum ထဲမှ Register ကို လှမ်း access လုပ်ထားပြီး bank_register ဆိုသည့် variable ထဲသို့ ထည့်ပါတယ်။

Line 31 မှာတော့ "Suspend".to_owned() ကို သုံးထားပါတယ် သူက rust ရဲ့ ownership သင်ခန်းစာမှ ဖြစ်ပြီး ငှားထားတဲ့ data မှ တဆင့် ကိုယ်ပိုင် data တစ်ခု ဖန်တီးခြင်း ဖြစ်ပါတယ်။ Rust ownership သင်ခန်းစာမှာ အသေးစိတ် ထပ်မံ ရှင်းလင်းပြသွားပါမည်။

Basic Control Flow

ယခုသင်ခန်းစာမှာတော့ basic control flow ဖြစ်တဲ့ if , else , looping များအကြောင်း ကို အကျဉ်းချုပ် ဆွေးနွေး သွားမှာ ဖြစ်ပါတယ်။

```

1 ▶ fn main(){
2     let data :i32 = 10;
3     let unknow_data :i32 = 5;
4
5     if data < unknow_data{
6         println!("data was small {data}");
7     }else {
8         println!("data was big {data}");
9     }
10
11 }

```

အထက်ပါ program မှာတော့ variable နှစ်ခု ကြေငြာထားပြီး ထို variable နှစ်ခုကို line 5 မှာ နှိုင်းယှဉ်တာ ဖြစ်ပါတယ်။

```

1 ▶ fn main(){
2     let data:bool = true;
3     if data{
4         println!("Checking True of False : True");
5     }else {
6         println!("Checking True or False : False");
7     }
8
9 }
```

အထက်ပါ program မှာတော့ boolean type ဖြစ်တဲ့ true or false ကို စစ်ဆေးခြင်း ဖြစ်ပါတယ်။ အကယ်၍ data သည် true ဖြစ်နေလျှင် line 4 ကို အလုပ်လုပ်မှာ ဖြစ်ပြီး false ဖြစ်နေလျှင်တော့ line 6 ကို အလုပ်လုပ်မှာ ဖြစ်ပါတယ်။ အကယ်၍ data ထဲတွင် boolean type မဟုတ်ပဲ အခြားသော type များ ထည့်ထားပါက program error တက်မှာ ဖြစ်ပါတယ်။

Infinite loop and compare two strings

```

1  /*
2   Lesson 3 Infinite loop and compare two strings
3   WinHtut(NCCIOS)
4  */
5   use std::io;
6  ► fn main(){
7      loop {
8          let mut user_input :String = String::new();
9          eprintln!("Enter data:");
10         match io::stdin().read_line( buf: &mut user_input) {
11             Ok(n :usize) => {
12                 println!("{n} bytes read");
13             }
14             Err(error :Error) => println!("error : {error}"),
15         }
16         if user_input.trim() == "success" {
17             println!("You are success this lesson");
18             break;
19         } else {
20             println!("You are fail this lesson");
21         }
22     }
23 }

```

Lesson 3 ပုံမှာတော့ looping ထဲမှ infinite loop ဖြစ်သည့် loop ကို သုံးထားပါတယ်။ rust programming တွင် infinite loop ကို အသုံးပြုရန် loop ဆိုသည့် keyword ကို သုံးနိုင်ပြီး line 7 တွင် ဖော်ပြထားပါတယ်။

Rust programming တွင် string type နှစ်မျိုးရှိပြီး ပထမ တစ်ခုသည် String ဖြစ်ပြီး ဒုတိယ တစ်ခုသည် &str (reference string) ဖြစ်ပါတယ်။