RUST PROGRAMMING

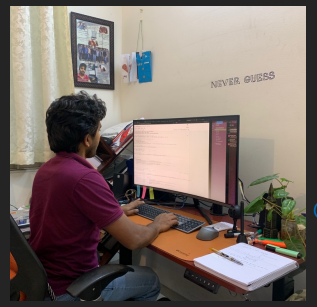
Introduction

Kamal kumar mukiri,
Apt Computing Labs

Kamal Kumar Mukiri

- **Founder of Apt Computing Labs**
- **NIT Rourkela Alumni**
- **Having 17 years s/w industry experience in C, C++, Python and Rust in Linux Environment.**
- **Worked on Network Components, Cyber Security Components, Automotive domain software and Automation.**
- **Worked in Ikanos, Huawei, Savari, Juniper and Harman**

Me



@ ACL, Bangalore

@ NTTF, Bangalore
C/C++ training

@ Great Learning, Mumbai
Python Training,

Outcome of this presentation:

- Course Logistics
- Importance of Rust Programming
  - Issues in C/C++
  - Core Features in Rust
- Installing Rust and Hands on
- Cargo for Project Management
  - Creating/Building/Running
  - Linting and Formatting
  - Writing/Running Test Cases
  - Getting Code Coverage Report
  - Documenting

Course logistics:

All trainees will have access to Daily recorded videos, Presentation slides (can not be distributed), Github path where Assignments and practiced programs gets uploaded daily and RoadMap of entire course.

1. Presentation slides: (Google Drive)

https://drive.google.com/drive/folders/1-2OPLRbJDF8xWtocH8by7mX0eFUreLLk?usp=drive_link

2. Daily recordings:(Google Drive)

https://drive.google.com/drive/folders/16PqZ-eg3VudOIPpSvRNKv3ctRpJRE3Cr?usp=drive_link

3. Programs to practice for trainees and Assignments: (GitHub)

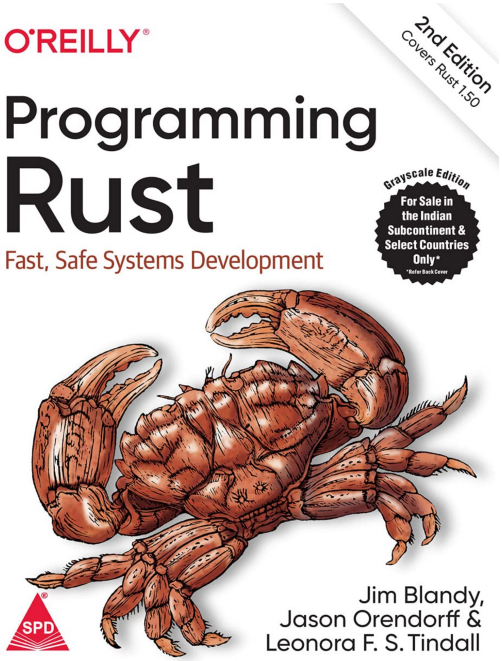
https://github.com/kamallearner123/APTRUSTNOV12

4. Course Roadmap:

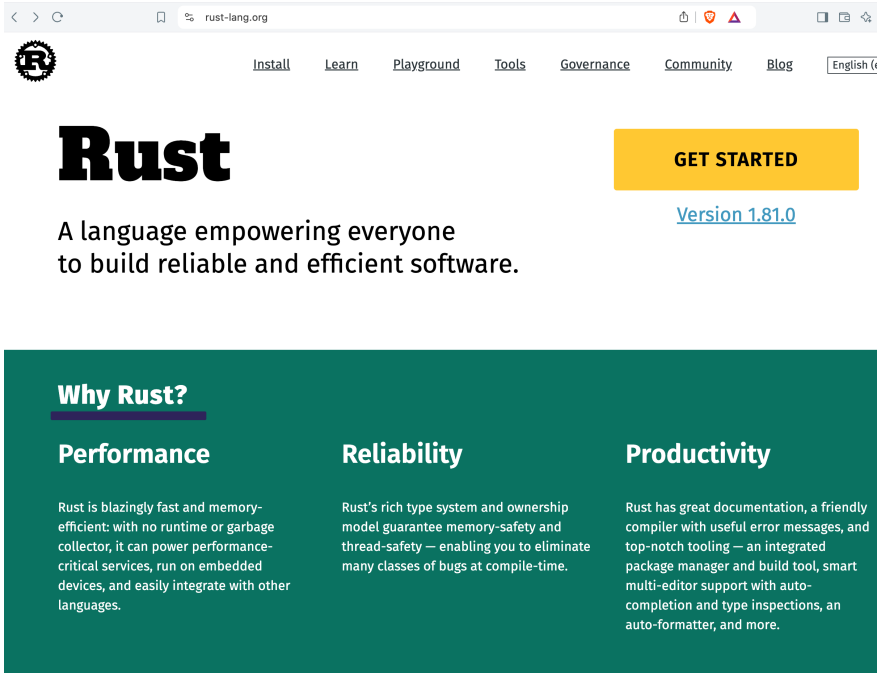
https://roadmap.sh/r/aptrustsess01

Please check access to all the links

Books and References:



Online tutorial and compiler: https://www.rust-lang.org/



Supporting tools:

The below are list of tools will be used as per need base.

1. Debugging : gdb, strace, lsof, top, etc...
2. Memory/Multi-thread Issues : Valgrind
3. Performance : perf
4. Network monitor : wireshark
5. ELF file inspector : readelf, ldd
6. Unit test, coverage, doc : cargo (in-built)

Market demand for Rust

> *"A language that doesn't affect the way you think about programming, is not worth knowing"*
>
> Alan Perlis

Home / Tech / Security

Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

Written by **Catalin Cimpanu,** Contributor

Feb. 11, 2019 at 7:48 a.m. PT

Linux Torvald's comment on Rust Adaptation:

"I was expecting [Rust] updates to be faster, but part of the problem is that old-time kernel developers are used to C and don't know Rust," Torvalds said.

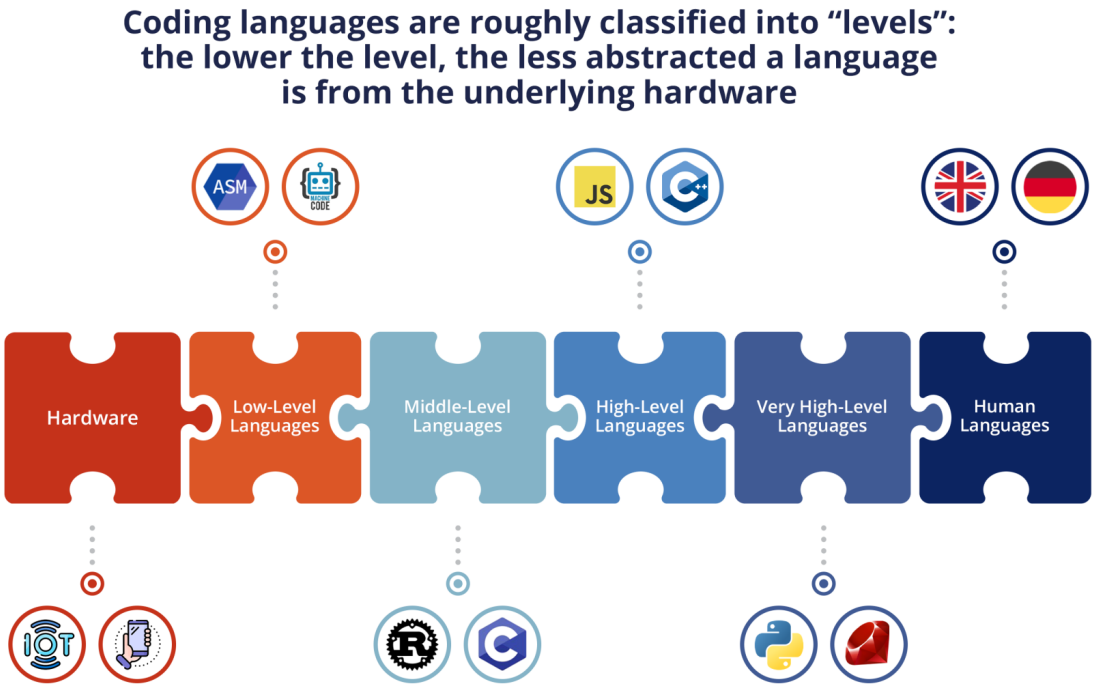
https://arstechnica.com/gadgets/2024/09/rust-in-linux-lead-retires-rather-than-deal-with-more-nontechnical-nonsense/

https://arstechnica.com/gadgets/2024/09/rust-in-linux-lead-retires-rather-than-deal-with-more-nontechnical-nonsense/

Programming in Rust is like a beautiful conversation between a programmer and a highly responsible yet strict compiler.

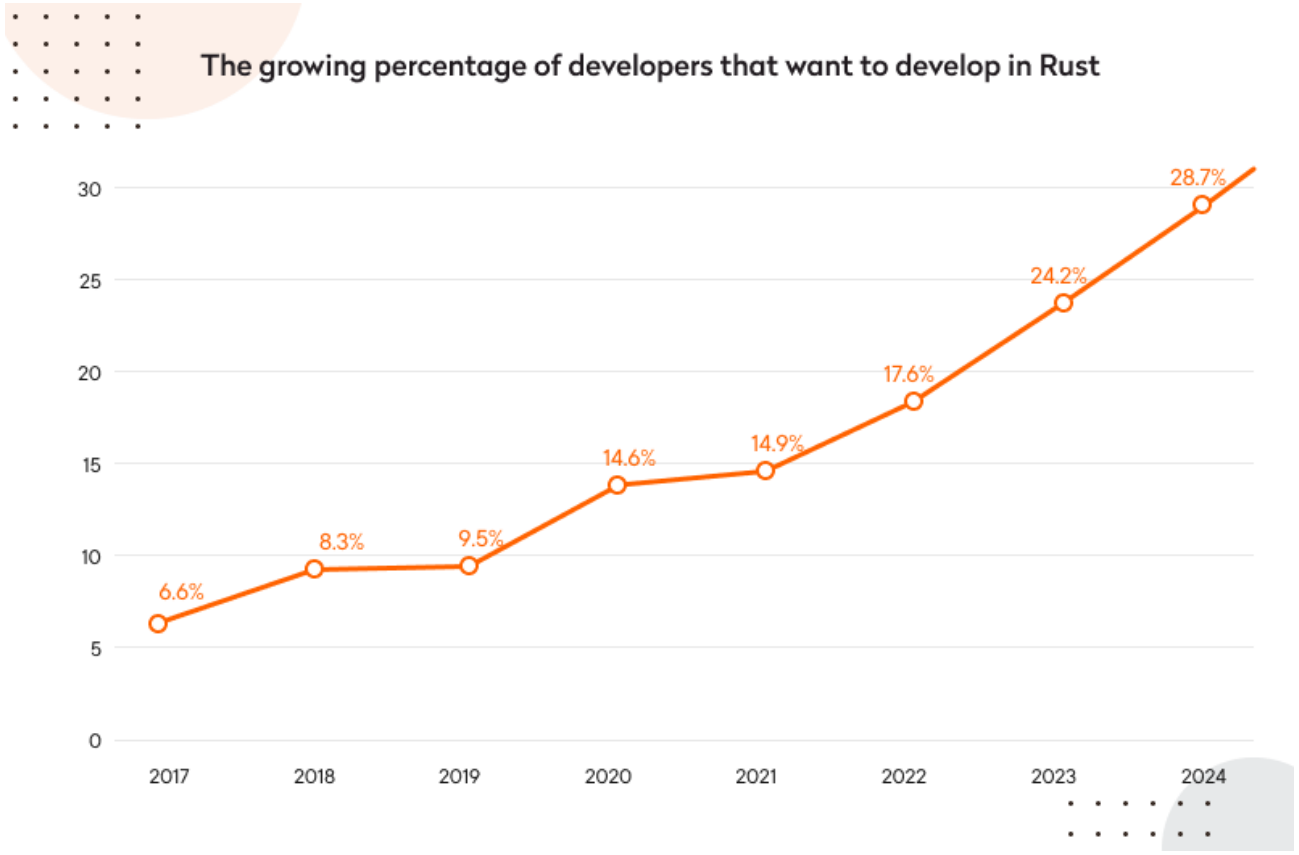
- The communication is clear, with no room for ambiguity.
- Rust's compiler ensures that assumptions, which often lead to unexpected surprises in other languages, are completely avoided.
- It demands precision, guiding developers toward writing safe and efficient code, while still offering flexibility to craft performant software.

What type of language is Rust



Source: https://kruschecompany.com/rust-language-concise-overview/

Growth of Rust Programming



https://yalantis.com/blog/rust-market-overview/

Issues in C and C++


```c
#include <stdio.h>
int main() {
    int x;
    printf("Value of x is: %d\n", x);
    return 0;
}
```


Issues in C and C++


```c
#include <stdio.h>
int main() {
    int x; // Uninitialized variable
    printf("Value of x is: %d\n", x); // Undefined behavior
    return 0;
}
```

Issues in C and C++


```c
#include <stdio.h>
int main( ) {
    int *ptr = NULL;
    *ptr = 100;
    return 0;
}
```


Issues in C and C++


```c
#include <stdio.h>
int main() {
    int *ptr = NULL;
    *ptr = 100; // Segmentation fault
    return 0;
}
```

Issues in C and C++


```c
#include <stdlib.h>
int main() {
    int *ptr = malloc(sizeof(int));
    free(ptr);
    free(ptr);
    return 0;
}
```


Issues in C and C++


```c
#include <stdlib.h>
int main() {
    int *ptr = malloc(sizeof(int));
    free(ptr);
    free(ptr); // Double free
    return 0;
}
```


Issues in C and C++


```c
#include <stdlib.h>
void func_with_leak() {
    int *ptr = malloc(sizeof(int));
}
int main() {
    func_with_leak();
    return 0;
}
```


Issues in C and C++


```c
#include <stdlib.h>
void func_with_leak() {
    int *ptr = malloc(sizeof(int));
    // Memory is allocated but never freed
    // The pointer goes out of scope, losing the reference
}
int main() {
    func_with_leak();
    return 0;
}
```


Issues in C and C++


```c
#include <stdlib.h>
int main() {
    int *ptr = malloc(sizeof(int));
    free(ptr);
    *ptr = 100;
    return 0;
}
```


Issues in C and C++


```c
#include <stdlib.h>
int main() {
    int *ptr = malloc(sizeof(int));
    free(ptr);
    *ptr = 100; // Use-after-free
    return 0;
}
```


Issues in C and C++


```c
#include <stdio.h>
#include <limits.h>
int main() {
    short int x = SHRT_MAX; // Maximum value for a short int
    x = x + 1;
    printf("Value of x is: %d\n", x);
    return 0;
}
```

Issues in C and C++


```c
#include <stdio.h>
#include <limits.h>
int main() {
    short int x = SHRT_MAX; // Maximum value for a short int
    x = x + 1; // Integer overflow
    printf("Value of x is: %d\n", x); // Output will be -32768
    return 0;
}
```

Issues in C and C++


```c
#include <stdio.h>
#include <pthread.h>

// This is the shared resource
long long counter = 0;

void* increment_thread(void* arg) {
    for (int i = 0; i < 10000; i++) {
        counter++;
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // Create two threads
    pthread_create(&thread1, NULL, increment_thread, NULL);
    pthread_create(&thread2, NULL, increment_thread, NULL);

    // Wait for both threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Final counter value: %lld\n", counter);

    return 0;
}
```

Vulnerabilities in C and C++


```cpp
#include <iostream>
#include <cstdlib>

void executeCommand(char *userInput) {
    char command[100];
    snprintf(command, sizeof(command), "ls %s", userInput);
    system(command);
}

int main() {
    char input[50] = "somefile.txt; cat CodeExecute.cpp";
    executeCommand(input);
    return 0;
}
```

CODE INJECT

Vulnerabilities in C and C++


```c
#include <stdio.h>
#include <string.h>
int main() {
    char buffer[10];
    strcpy(buffer, "This string is too long for the buffer");
    return 0;
}
```


Vulnerabilities in C and C++


```c
#include <stdio.h>
#include <string.h>
int main() {
    char buffer[10];
    strcpy(buffer, "This string is too long for the buffer"); // Buffer overflow
    return 0;
}
```

1. Memory Safety Issues

- **Dangling Pointers**
- **Use-After-Free**
- **Buffer Overflows**
- **Manual Memory Management**
- **Null Pointer Dereferencing**

2. Data Races in Multithreaded Programs

- **Race Conditions**: C/C++ programs using threads are prone to data races where two threads access shared data concurrently, with at least one thread modifying the data.
- **Manual Synchronization**: Developers need to use mutexes, locks, and other mechanisms, which are error-prone and can lead to deadlocks or incorrect synchronization.

3. Undefined Behaviour

- **Undefined Behaviour (UB)**: C/C++ has many undefined behaviours, such as integer overflows, dereferencing null pointers, or violating alignment rules. These behaviours are not checked at compile time, potentially leading to serious bugs.
- **Type Casting**: Unsafe type casting in C/C++ (e.g., casting between incompatible types) can lead to undefined behaviour.

4. Lack of Proper Error Handling

- **Error Handling**: In C/C++, error handling is often done through return codes, errno, or exceptions, leading to inconsistent and hard-to-maintain error-handling strategies.
- **Ignored Errors**: It's common for C/C++ developers to ignore or forget to check return values, leading to silent failures.

5. Manual Resource Management

- **RAII (Resource Acquisition Is Initialization)**: While C++ has RAII, developers need to manually implement it to manage resources like file handles and sockets, and there's no equivalent in C.
- **Leaks**: Forgetting to free resources leads to memory and resource leaks.

Any ways... C is cute

Core features provided by Rust

- Ownership and Borrowing
- Lifetimes
- Pattern Matching
- Zero-Cost Abstractions
- Memory Safety without Garbage Collection
- Traits and Trait Objects
- Concurrency without Data Races
- Algebraic Data Types (Enums and Structs)
- Error Handling with Result and Option
- Type Inference

- Macros (Declarative and Procedural)
- Immutable and Mutable Variables
- Crate and Module System
- Closures and Functional Programming Features
- Asynchronous Programming (Async/Await)
- Safe and Unsafe Code Blocks
- Iterator and Iterator Adapters
- Smart Pointers (Box, Rc, Arc)
- Embedded Programming Support
- Cross-Compilation

How does Rust address issues in C++ ?

C/C++ Issues	Rust Features
Memory leaks, dangling pointers	**Ownership and Borrowing** System
Null pointers	**Option<T>** for safe null handling
Buffer overflows	Bounds checking on arrays and slices
Use-after-free	**Ownership, Lifetimes, and Borrowing**
Manual memory management	Automatic memory management via RAII (Resource Acquisition Is Initialization)
Data races	Concurrency with Ownership, **Send**, **Sync**, **Arc**
Arbitrary pointer arithmetic	No pointer arithmetic in safe Rust
Poor error handling	**Result<T, E>** and pattern matching
Cryptic compiler errors	Rust's clear and detailed compiler messages
Threading issues	Safe concurrency primitives (**Mutex**, **Arc**)
Complex build systems	**Cargo** package manager and build system

Cargo: Build and Package System

Rust Core tools:

Rustup: This tool helps manage Rust versions, a crucial feature as Rust is continually evolving to meet the demands of high-performance computing.

Cargo: Rust's dependency management tool relies on TOML configuration files. Cargo is also the main frontend for the compilation process - Simply type **cargo run** or **cargo build** --release. With Cargo Watch, developers can leverage the most innovative iterative workflow.

Rustc: Rust's compiler isn't just a compiler; it's also a static code checker and transpiler. It generates intermediate code passed to LLVM for optimization, a critical aspect of making applications efficient and performant.

Cargo: Tools


```
cargo build         - Compiles the current project.

cargo run           - Compiles and runs the project.

cargo test          - Runs all tests in the project.

cargo fmt           - Formats the project's code according to Rust's style guidelines.

cargo clippy        - Runs the Clippy linter to catch common mistakes and improve code quality.

cargo check         - Quickly checks the code for errors without generating an executable.

cargo doc           - Generates documentation for the project.

cargo install <package> - Installs a Rust binary package.

cargo add <dependency> - Adds a dependency to Cargo.toml (via cargo-edit).
```

Cargo: Tools


```
cargo rm <dependency> - Removes a dependency from Cargo.toml (via cargo-edit).

cargo update          - Updates dependencies in Cargo.lock.

cargo outdated        - Displays outdated dependencies (via cargo-outdated).

cargo audit           - Audits the project's dependencies for known security vulnerabilities (via cargo-
                        audit).

cargo bench           - Runs benchmarks in the project.

cargo watch -x <command> - Watches for changes and runs a command when files are modified (via cargo-
                           watch).
```

Cargo: Tools

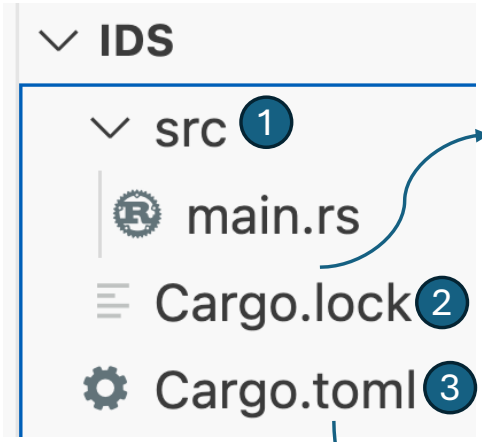

```
cargo tarpaulin     - Runs code coverage analysis (via cargo-tarpaulin).

cargo clean         - Removes the target directory to free up disk space or force a rebuild.
```

Creating a project using cargo

Rust tool chain and echo system

1) Creating a project
2) Examine Cargo.toml
3) Adding additional Binaries
4) Adding additional Libraries

Creating project using "cargo new <name>":



Cargo.lock

```
1  # This file is automatically @generated by Cargo.
2  # It is not intended for manual editing.
3  version = 3
4
5  [[package]]
6  name = "ids"
7  version = "0.1.0"
8
```

Cargo.toml

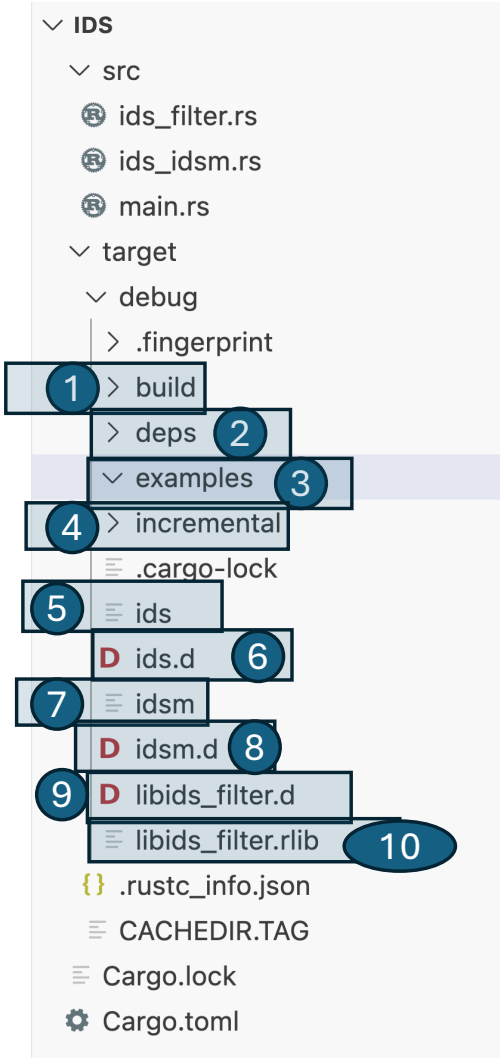
```
1  [package]
2  name = "ids"
3  version = "0.1.0"
4  edition = "2021"
5
6  [dependencies]
7
```

Creating a project (Target is binary)

% cargo new ids

1) src: Source directory (main.rs should be there). All the files should be extended with .rs
2) Cargo.lock freezes the state of dependencies, allowing for consistent builds across different environments and times.
3) Cargo.toml keeps the information about package and its dependencies. It can be edited manually unlike "Cargo.lock"

Building project using "`cargo build`":



Building project

```
% cargo build
```

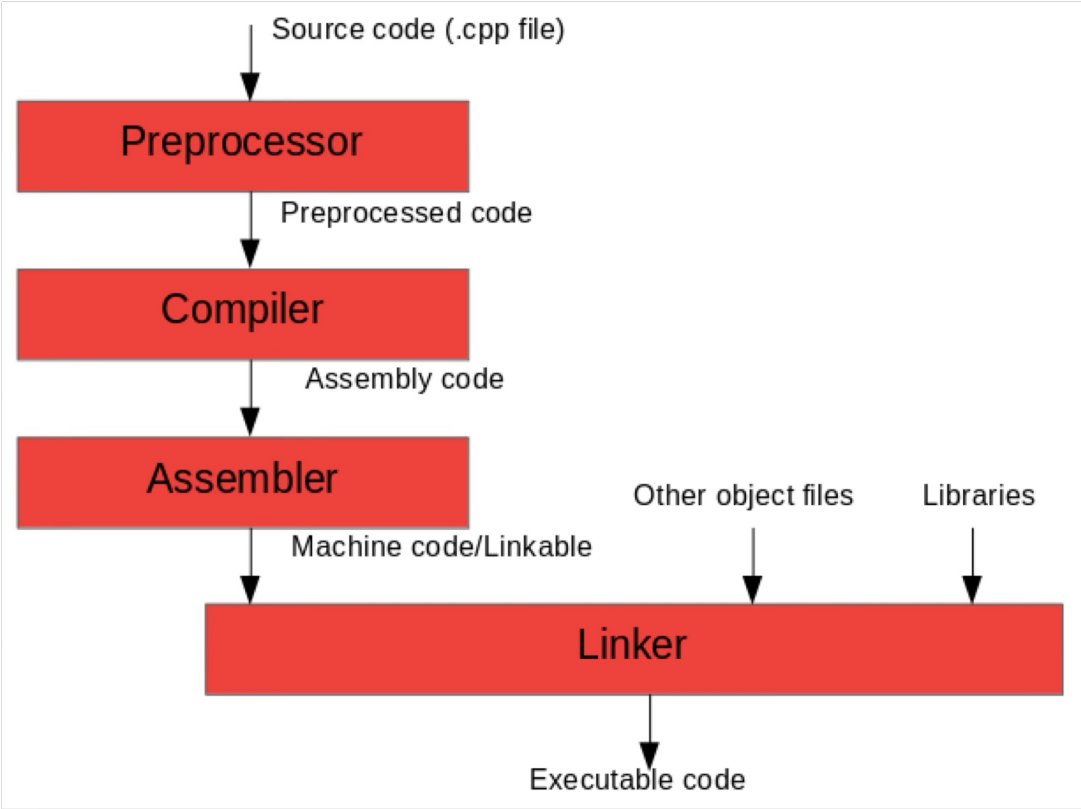
1. Build folder
2. Contains dependencies
3. Sample exmaples
4. Incremental
5. Binary
6. Dynamic library
7. Binary

Creating a project (Target is binary)

```
% cargo build
```

1) “ids”: By default, debug binary gets created

Compilation process of C/C++ program:



Source (.rs)

↓

Lexer → Parser → AST

↓

Macro Expansion → Expanded AST

↓

Name Resolution → HIR

↓

Type Check + Borrow Check

↓

MIR (Mid-level IR) + Optimizations

↓

LLVM IR → Optimizations

↓

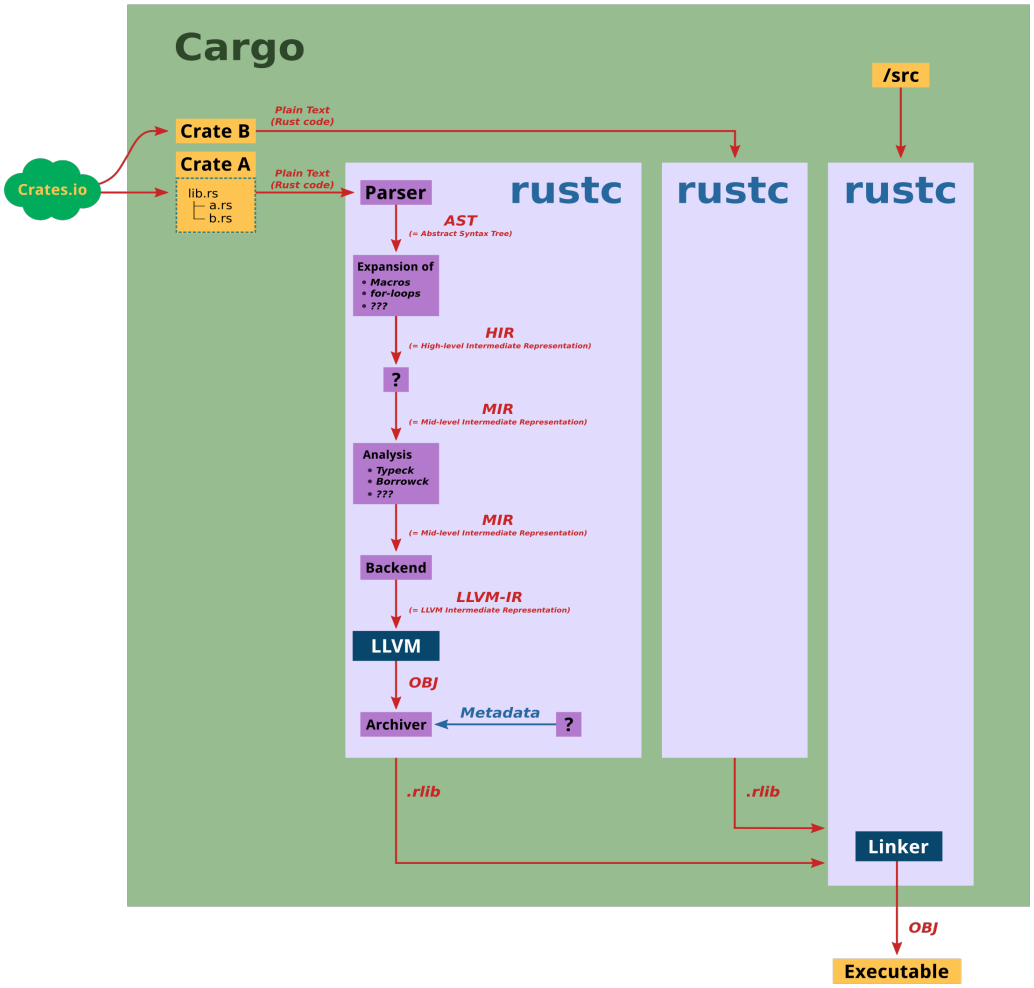
Machine Code

↓

Linker → Executable

Generic Cargo build system:

Source: https://blog.shrirambalaji.com/posts/resolving-rust-symbols/

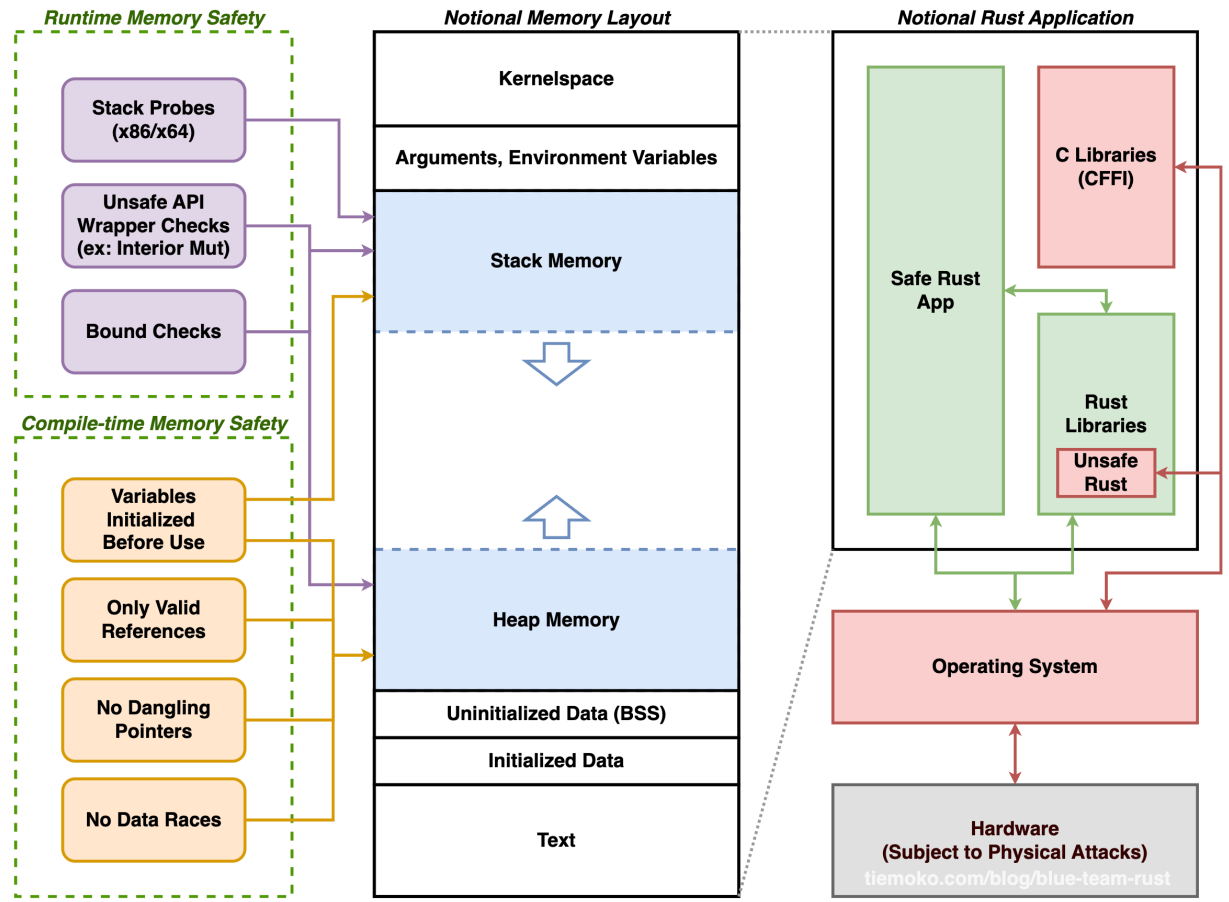


Apt Computing Labs

Rust: LLVM is the **default** and **only** backend in rustc.

GCC: Typically does not use LLVM, as it has its own backend and optimization pipeline.

Process Memory and Features



Source: https://tiemoko.com/blog/blue-team-rust/

Clippy tool

$cargo clippy

A lint tool, does not generate binary/library.

Finds issues:

1) Style
2) Correction
3) Complexity
4) Performance

$cargo clippy

Sample errors:

1) No need of return
2) Useless code
3) Too many args
4) Iterator count

Rust format tool

$cargo fmt


```
3   fn fun() -> u8 {return 0;}
4
5   fn main() {
6   println!("Hello, world!");
7   }
8
```

```
3   fn fun() -> u8 {
4       return 0;
5   }
6
7   fn main() {
8       println!("Hello, world!");
9   }
```

1. Code before running $cargo fmt
2. Code after running command $cargo fmt

Testing code

Testing code: Writing test case


```rust
mod mymod;

fn fun() -> u8 {
    return 0;
}
fn main() {
    println!("Hello, world!");
}

#[cfg(test)]
mod test {
    use super::*;
    #[test]
    fn test_01() {
        assert_eq!(fun(), 0);
    }

    #[test]
    fn test_mymod() {
        assert_eq!(mymod::whole_square(1,2), 9);
    }
}
```

1

2

3

```
kamalmukiri@Kamals[illegible]k-Air fmt_clippy % cargo test
   Compiling fmt_c[illegible]v0.1.0 (/Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Exercise/Kamal/3.IdiomaticCode/fmt_clippy)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.88s
     Running unittests src/main.rs (target/debug/deps/fmt_clippy-ab8821f7788ca832)

running 2 tests
test test::test_01 ... ok
test test::test_mymod ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

1. #[cfg(test)] attribute is used to conditionally include code only when compiling for tests
2. #[test] Marks a function as a test case.
3. $cargo test: Running test cases

$cargo test: Tools

Prerequisites:

$cargo install cargo-tarpaulin

$cargo install cargo-nextest

Running tools:

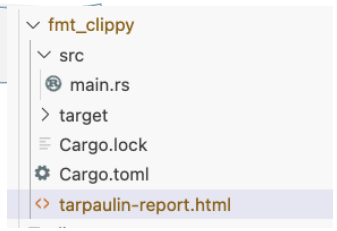
```
$ cargo tarpaulin --out Html    (To generate coverage report)

$ cargo nextest run --test-threads=2 test --message-format=libtest-json > TestReport.json
```

Usage: cargo nextest run [OPTIONS] [FILTERS]... [-- <FILTERS_AND_ARGS>...]

Usage: cargo nextest run [OPTIONS] [FILTERS]... [-- <FILTERS_AND_ARGS>...]

$cargo test: Coverage

Find the report in main directory:



Test report:

/Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Exercise/Kamal/3.IdiomaticCode/fmt_clippy/src — Covered: 3 of 5 (60.00%)

Path	Coverage
main.rs	2 / 4 (50.00%)
mymod.rs	1 / 1 (100.00%)

Back /Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Exercise/Kamal/3.IdiomaticCode/fmt_clippy/src/main.rs — Covered: 2 of 4 (50.00%)

```
1 mod mymod;
2
3 fn fun() -> u8 {
4     return 0;
5 }
6 fn main() {
7     println!("Hello, world!");
8 }
9
10 #[cfg(test)]
11 mod test {
12     use super::*;
13     #[test]
14     fn test_01() {
15         assert_eq!(fun(), 0);
16     }
17
18     #[test]
19     fn test_mymod() {
20         assert_eq!(mymod::whole_square(1,2), 9);
21     }
22 }
23
```

Documenting code

Documenting: Outer documentation (///)


```
...
/// # History                                                    1
/// areaCircle: Function which calculates area of circle and returns
/// - r : Radius in f32                                         2
/// - Returns: Area in f32
/// ## References
/// - Please refer [`whole_square`](mymod::whole_square)        3
fn areaCircle(r:f32) -> f32 {
    return  PI*r.powf(2.0);
}
```

Output

```
kamalmukiri@Kamals-MacBook-Air clippy_fmt % cargo doc --no-deps --open
```

4

Comments can be given with "///" as prefix.

1. #History : Adding headings
2. - r: Adding bullets.
3. [`Refer`](link): Hyper link

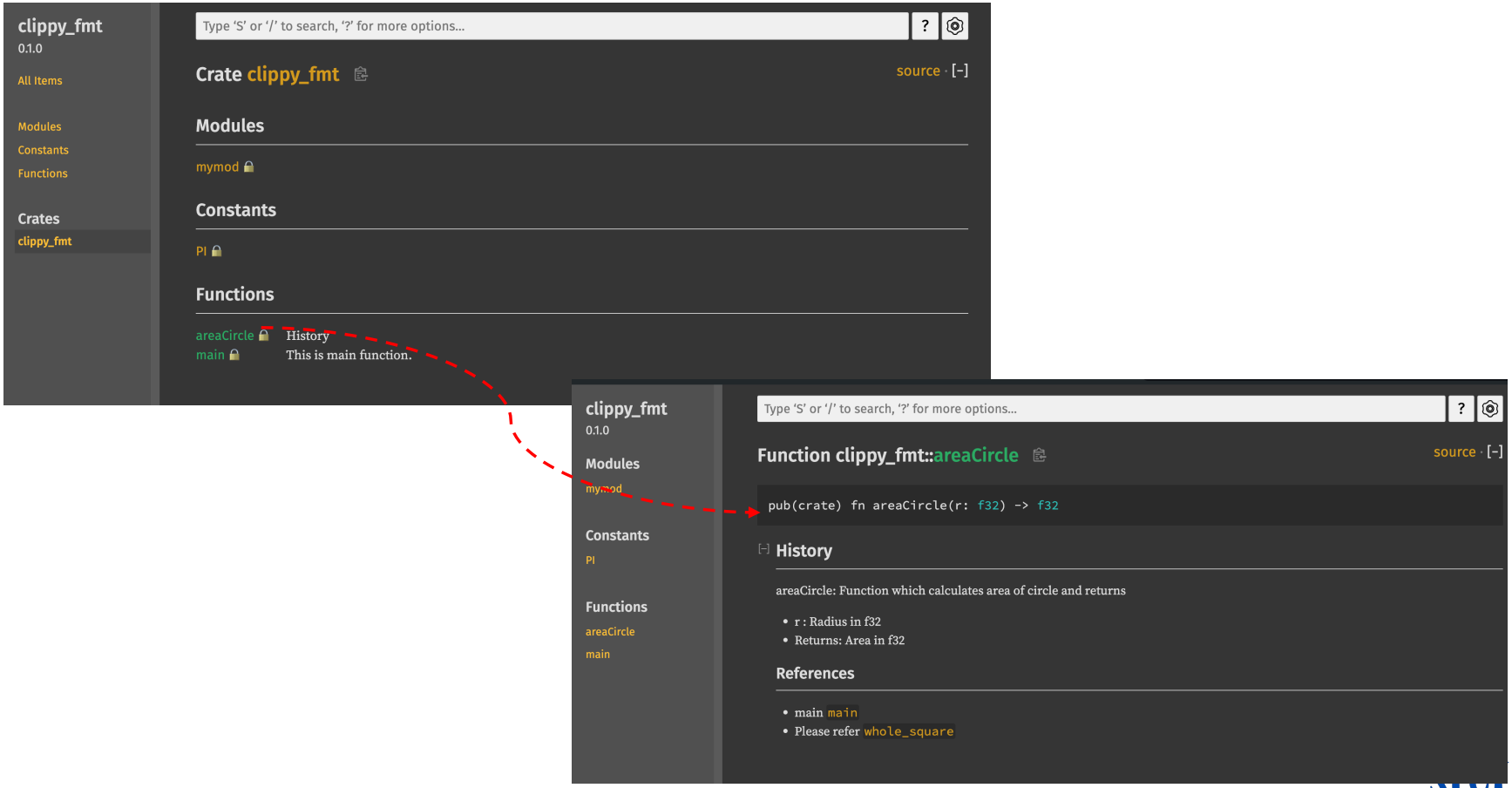
   "whole_square" is link name, "mymod::whole_square" is link.

4. Generating document

   "—no-deps" : No need to generate doc for dependencies

   "—open" : Open the document as soon as the command is executed.

Documenting



Documenting: Inner documentation (//!)


```rust
fn main() {
    //!
    //! At least 3 arguments are required.
    //!
    let radius:f32 = (10.0);
    let my_vec:Vec<i32> = vec![1,2,3,4];

    println!("Radius = {}", radius);
    println!("Area = {}",areaCircle(radius));

}
```

```
kamalmukiri@Kamals-MacBook-Air clippy_fmt % cargo doc --no-deps --open
```

Inner comments can be given with “//!” with prefix.

Documenting:

All mark down features are supported.

More Options: https://doc.rust-lang.org/cargo/commands/cargo-doc.html

Thank you !!!