ACES: Rust

Introduction to Rust Programming Language

Jian Tao

jtao@tamu.edu

Fall 2024 HPRC Short Course

11/12/2024

ACES
ACCELERATING COMPUTING
FOR EMERGING SCIENCES

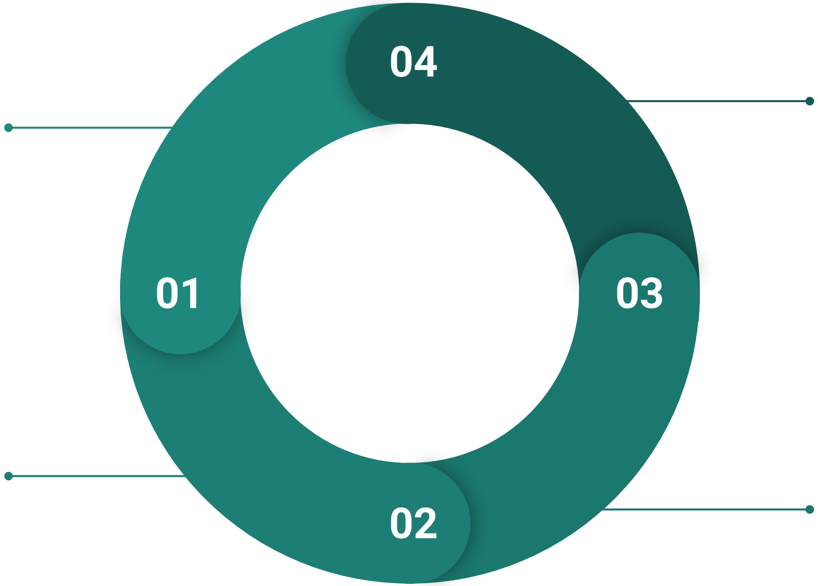
TACC
TEXAS ADVANCED
COMPUTING CENTER

ATM
High Performance
Research Computing
DIVISION OF RESEARCH

UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN

NSF

Introduction to Rust



Part I. Getting Started with ACES (~15 mins)

Part II. Rust - What and Why? (~15 mins)

Part III. Basics of Rust (~60 mins)

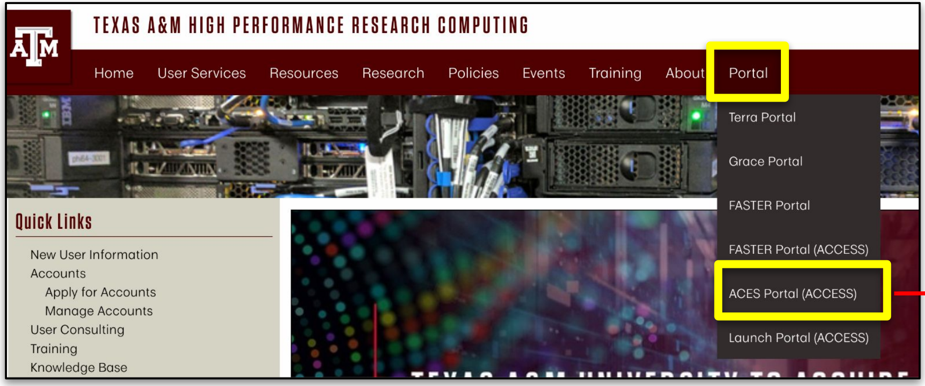
Part IV. Advanced Topics in Rust (~30 mins)

Part I. Get Started with ACES

High Performance Research Computing
ACES
ACCELERATING COMPUTING FOR EMERGING SCIENCES

HPRC Short Course: Introduction to Composable Computing ACES and FASTER

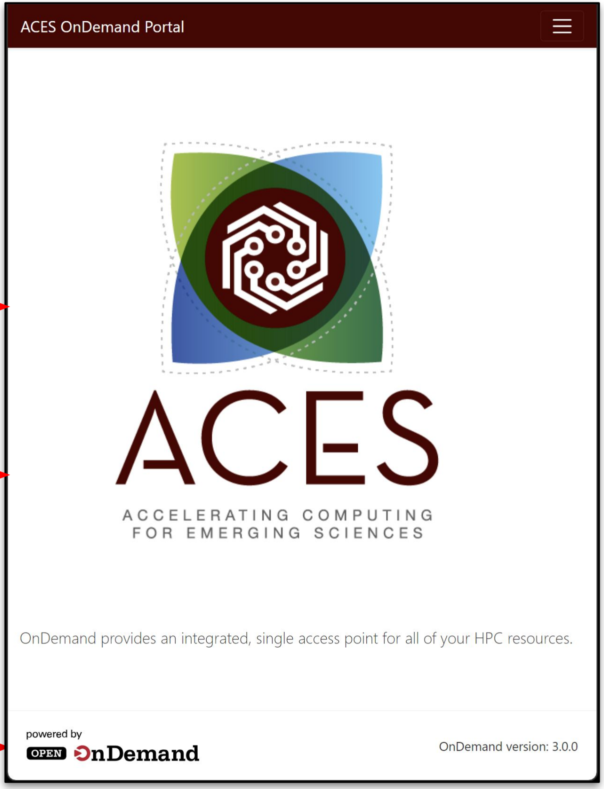
ACES Portal



ACES Portal portal-aces.hprc.tamu.edu
is the web-based user interface for the ACES cluster

HPRC Portal YouTube tutorials

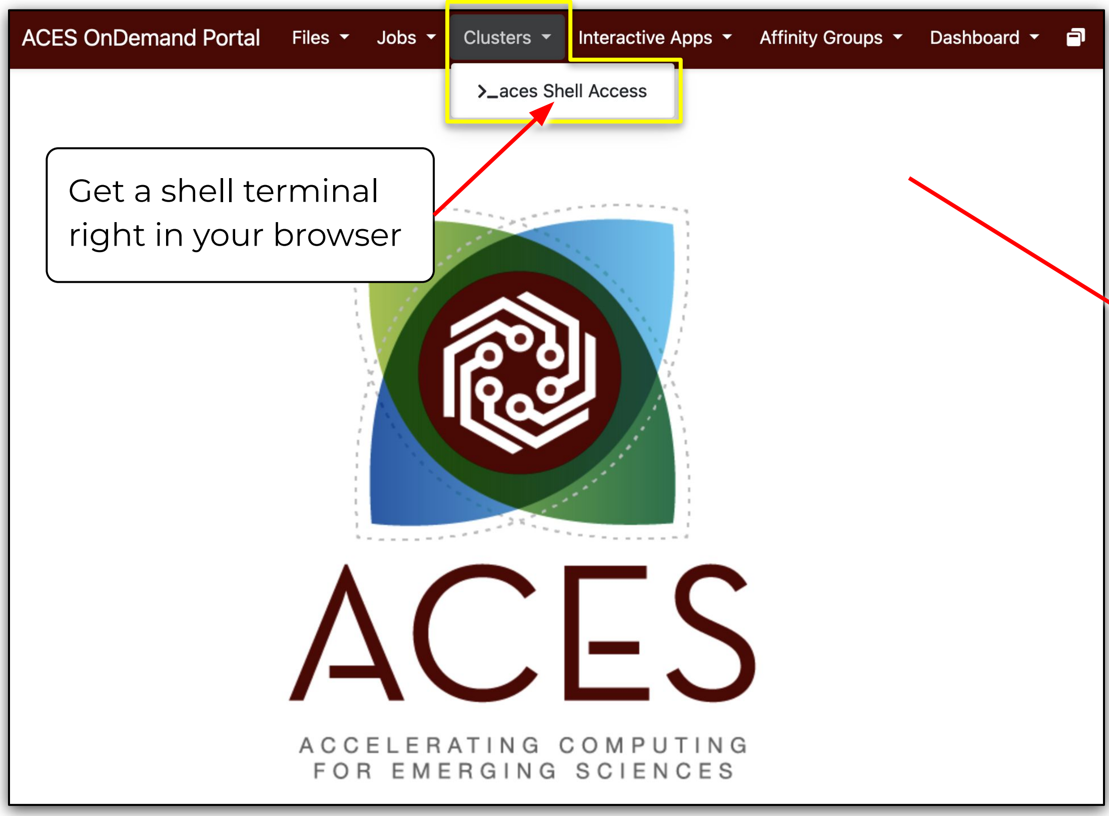
Open OnDemand (OOD) is an advanced web-based graphical interface framework for HPC users



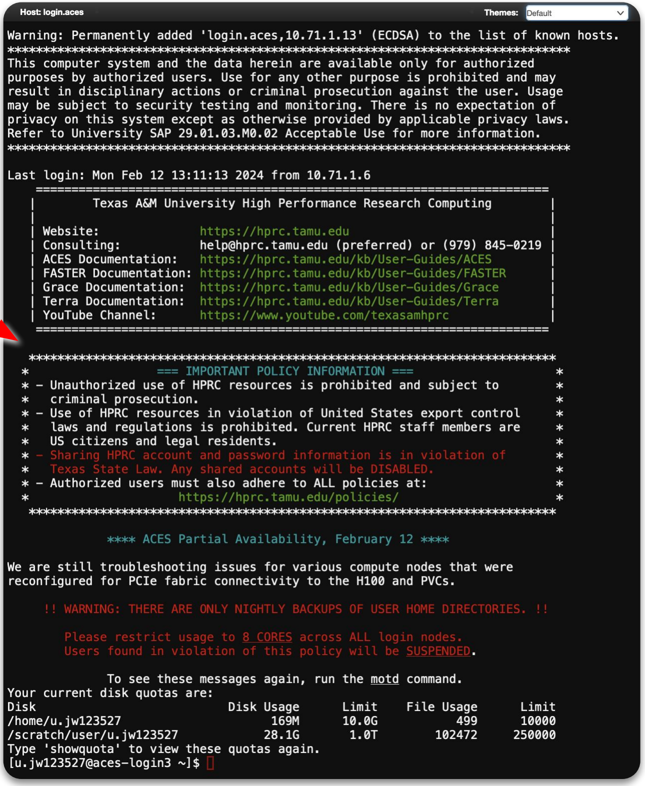
OnDemand provides an integrated, single access point for all of your HPC resources.

OnDemand version: 3.0.0

Shell Access via the Portal



Get a shell terminal right in your browser



```
Warning: Permanently added 'login.aces,10.71.1.13' (ECDSA) to the list of known hosts.
******************************************************************************
This computer system and the data herein are available only for authorized
purposes by authorized users. Use for any other purpose is prohibited and may
result in disciplinary actions or criminal prosecution against the user. Usage
may be subject to security testing and monitoring. There is no expectation of
privacy on this system except as otherwise provided by applicable privacy laws.
Refer to University SAP 29.01.03.M0.02 Acceptable Use for more information.
******************************************************************************

Last login: Mon Feb 12 13:11:13 2024 from 10.71.1.6
    ========================================================================
   |         Texas A&M University High Performance Research Computing        |
   |                                                                         |
   | Website:                https://hprc.tamu.edu                           |
   | Consulting:             help@hprc.tamu.edu (preferred) or (979) 845-0219 |
   | ACES Documentation:     https://hprc.tamu.edu/kb/User-Guides/ACES       |
   | FASTER Documentation:   https://hprc.tamu.edu/kb/User-Guides/FASTER     |
   | Grace Documentation:    https://hprc.tamu.edu/kb/User-Guides/Grace      |
   | Terra Documentation:    https://hprc.tamu.edu/kb/User-Guides/Terra      |
   | YouTube Channel:        https://www.youtube.com/texasamhprc             |
    ========================================================================

   **************************************************************************
  *                  === IMPORTANT POLICY INFORMATION ===                     *
  * - Unauthorized use of HPRC resources is prohibited and subject to        *
  *   criminal prosecution.                                                  *
  * - Use of HPRC resources in violation of United States export control     *
  *   laws and regulations is prohibited. Current HPRC staff members are     *
  *   US citizens and legal residents.                                       *
  * - Sharing HPRC account and password information is in violation of      *
  *   Texas State Law. Any shared accounts will be DISABLED.                 *
  * - Authorized users must also adhere to ALL policies at:                  *
  *                  https://hprc.tamu.edu/policies/                         *
   **************************************************************************

              **** ACES Partial Availability, February 12 ****

We are still troubleshooting issues for various compute nodes that were
reconfigured for PCIe fabric connectivity to the H100 and PVCs.

     !! WARNING: THERE ARE ONLY NIGHTLY BACKUPS OF USER HOME DIRECTORIES. !!

        Please restrict usage to 8 CORES across ALL login nodes.
        Users found in violation of this policy will be SUSPENDED.

              To see these messages again, run the motd command.
Your current disk quotas are:
Disk                           Disk Usage      Limit    File Usage      Limit
/home/u.jw123527                     169M      10.0G           499      10000
/scratch/user/u.jw123527            28.1G       1.0T        102472     250000
Type 'showquota' to view these quotas again.
[u.jw123527@aces-login3 ~]$ 
```

Composable HPC Architectures for AI

Common HPC

- Built on Converged Hardware
- Static Hardware Design
- Fixed GPU/Accelerator
- Fixed Memory
- Storage: SATA and SAS
- Vendor Lock

FUTURE >

< PAST

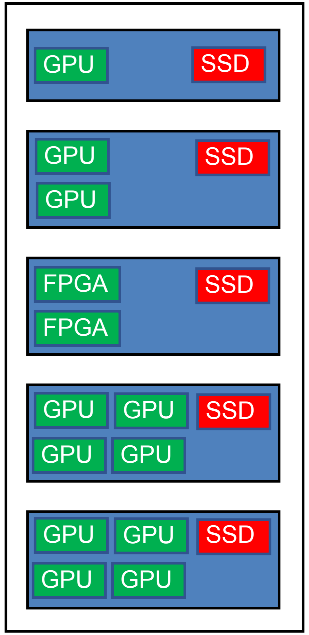
HPC for AI

- Built on Disaggregated Hardware
- Composable Hardware Platform
- Composable GPU/Accelerator
- Composable Memory - Optane
- Modern Storage: NVMe-oF
- Open Platform

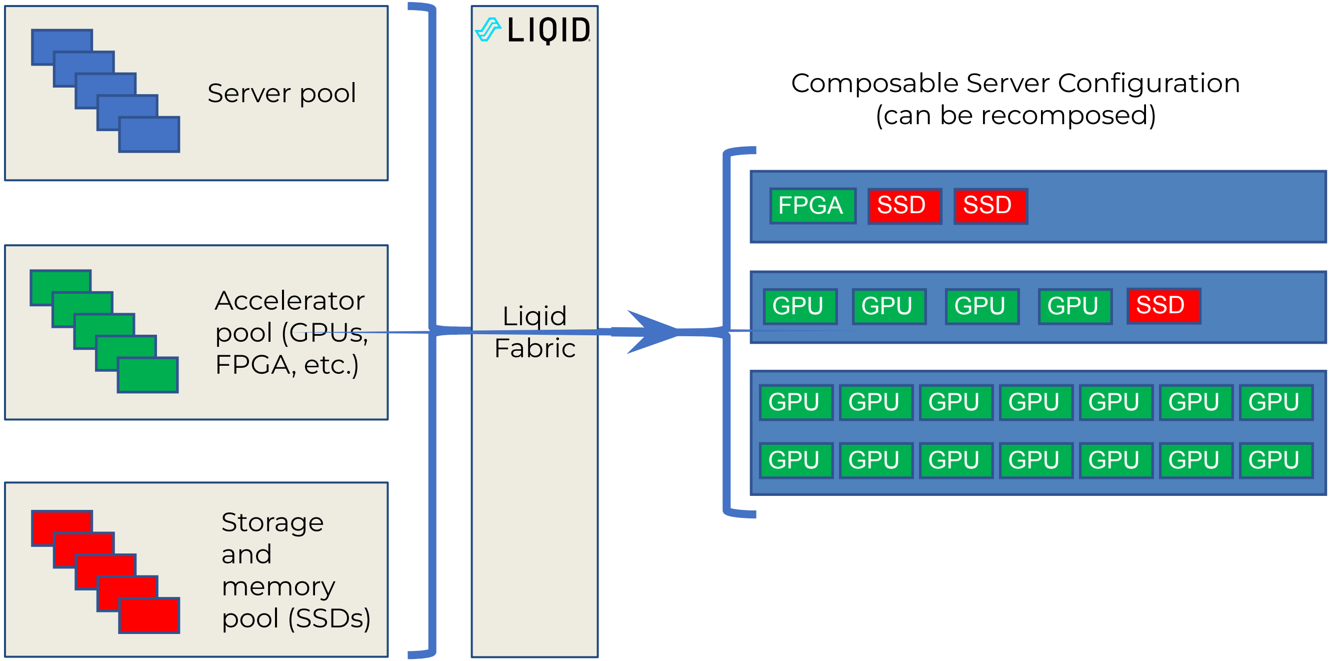
Next Generation HPC/AI Platform Supports Composable Accelerators and Memory

Composability

Traditional Server Configuration



Composable Resources

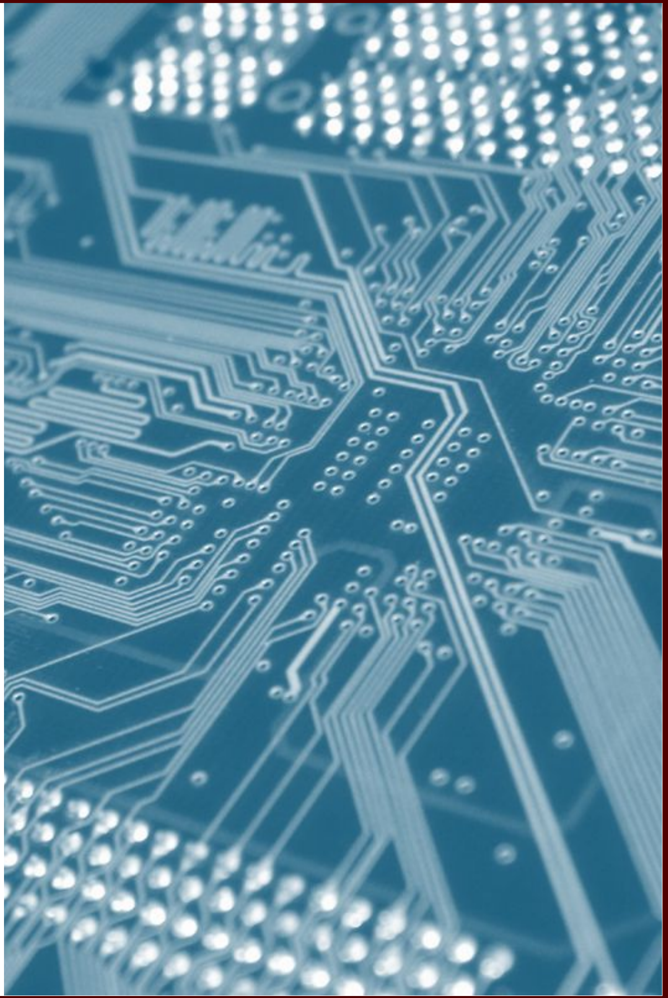

Server pool
Accelerator pool (GPUs, FPGA, etc.)
Storage and memory pool (SSDs)
LIQID
Liqid Fabric
Composable Server Configuration (can be recomposed)
FPGA
SSD
SSD
GPU
GPU
GPU
GPU
SSD
GPU
GPU
GPU
GPU
GPU
GPU
GPU
GPU
GPU
GPU
GPU
GPU
GPU
GPU

https://hprc.tamu.edu/kb/User-Guides/ACES

HPRC's
Composable Clusters

- **FASTER** – First large-scale composable CPU/GPU system
- **ACES** – Composability for mixed-resource workflows

Focusing on ACES today



NSF ACES

Accelerating Computing for Emerging Sciences

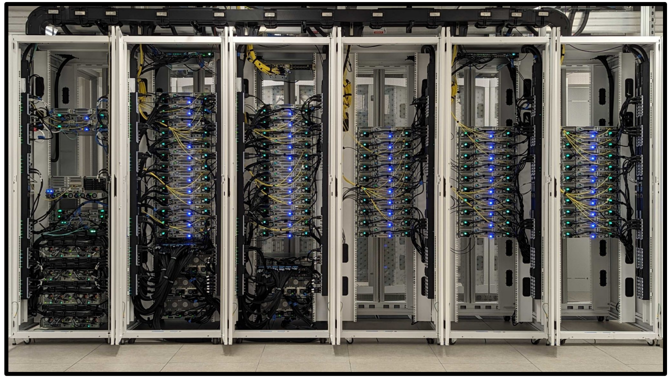
Our Mission:

- NSF ACSS CI test-bed
- Offer an accelerator testbed for numerical simulations and **AI/ML workloads**
- Provide consulting, technical guidance, and training to researchers
- Collaborate on computational and data-enabled research.

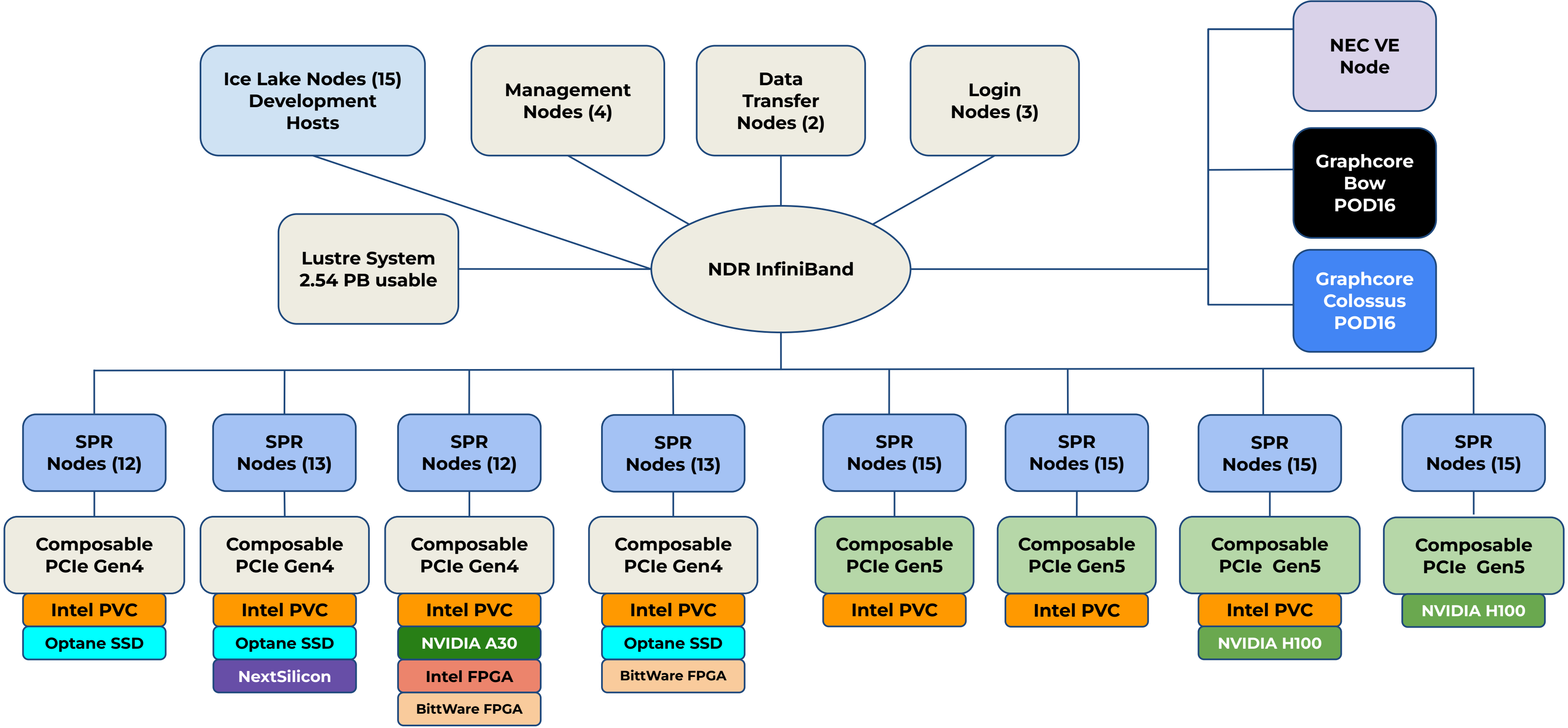
ACES

ACCELERATING COMPUTING
FOR EMERGING SCIENCES

ACES In Action



ACES Configuration



Ice Lake Nodes (15) Development Hosts

Management Nodes (4)

Data Transfer Nodes (2)

Login Nodes (3)

NEC VE Node

Graphcore Bow POD16

Graphcore Colossus POD16

Lustre System 2.54 PB usable

NDR InfiniBand

SPR Nodes (12)
- Composable PCIe Gen4
- Intel PVC
- Optane SSD

SPR Nodes (13)
- Composable PCIe Gen4
- Intel PVC
- Optane SSD
- NextSilicon

SPR Nodes (12)
- Composable PCIe Gen4
- Intel PVC
- NVIDIA A30
- Intel FPGA
- BittWare FPGA

SPR Nodes (13)
- Composable PCIe Gen4
- Intel PVC
- Optane SSD
- BittWare FPGA

SPR Nodes (15)
- Composable PCIe Gen5
- Intel PVC

SPR Nodes (15)
- Composable PCIe Gen5
- Intel PVC

SPR Nodes (15)
- Composable PCIe Gen5
- Intel PVC
- NVIDIA H100

SPR Nodes (15)
- Composable PCIe Gen5
- NVIDIA H100

ACES System Description

Component	Quantity	Description
Sapphire Rapids Nodes: Compute Nodes Data Transfer Nodes Login & Management Nodes	 110 nodes 2 nodes 5 nodes	96 cores per node, dual Intel Xeon 8468 processors 512 GB DDR5 memory 1.6 TB NVMe storage Compute: NVIDIA Mellanox NDR 200 Gbps InfiniBand adapter DTNs & Login & Management nodes: 100 Gbps Ethernet adapter
Ice Lake Login & Management Nodes	2 nodes	64 cores per node, dual Intel Xeon 8352Y processors 512 GB DDR4 memory 1.6 TB NVMe storage NVIDIA Mellanox NDR 200 Gbps InfiniBand adapter
PCIe Gen4 Composable Infrastructure	50 SPR nodes	Dynamically reconfigurable infrastructure that allows up to 20 PCIe cards (GPU, FPGA, etc.) per compute node
PCIe Gen5 Composable Infrastructure	60 SPR nodes	Dynamically reconfigurable infrastructure that allows up to 16 H100s or 14 PVCs per compute node
NVIDIA InfiniBand (IB) Interconnect	110 nodes	Two leaf and two spine switches in a 2:1 fat tree topology
DDN Lustre Storage	2.5 PB usable	HDR IB connected flash and disk storage for Lustre file systems

ACES Accelerators

Component	Quantity	Description
Graphcore IPU	32	16 Colossus GC200 IPUs, 16 Bow IPUs. Each IPU group hosted with a CPU server as a POD16 on a 100 GbE RoCE fabric
FPGAs: Intel PAC D5005 BittWare IA-840F	 2 3	 Accelerator with Intel Stratix 10 GX FPGA and 32 GB DDR4 Accelerator with Agilex AGF027 FPGA and 64 GB of DDR4
NextSilicon Coprocessor	2	Reconfigurable accelerator with an optimizer continuously evaluating application behavior.
NEC Vector Engine	8	Vector computing card (8 cores and HBM2 memory)
Intel Optane SSD	48	18 TB of SSDs addressable as memory w/ MemVerge Memory Machine.
NVIDIA GPUs: H100 A30	 30 4	 For HPC, DL Training, AI Inference For AI Inference and Mainstream Compute
Intel PVC GPUs	120	Intel GPUs for HPC, DL Training, AI Inference

Refer to our Knowledge Base for more:
https://hprc.tamu.edu/kb/User-Guides/ACES/Hardware/

System Software Stack

Function	Component	Version
Cluster Management	xCAT	2.16.4
Primary OS	Red Hat Enterprise Linux	8.9
HPC Scheduler	Slurm	22.05.11
InfiniBand Subnet Manager	UFM	6.12
OFED	Mellanox OFED	23.10-2.1.3
Storage Client	Lustre	2.12.9_ddn38
Software Management	Lmod	8.7
Software Build Framework	EasyBuild	4.9.2
Web Portal Software	Open OnDemand	3.0
Data Movement Software	Globus Connect Server	5.4
Job Reporting Software	Open XDMoD	10.5

Accelerator Access Summary

Component	Access	node or partition
BittWare IA-840F FPGA	Slurm	--partition=bittware
Intel PAC D5005 FPGA	Slurm	--partition=d5005
Intel GPU Max 1100 (PVC)	Slurm	--partition=pvc
Intel Optane SSD	Slurm	--partition=memverge
NextSilicon Coprocessor	Slurm	--partition=nextsilicon
NVIDIA A30 GPUs	Slurm	--partition=gpu
NVIDIA H100 GPUs	Slurm	--partition=gpu
Graphcore Bow IPUs	Interactive	ssh poplar2
Graphcore Colossus IPUs	Interactive	ssh poplar1
NEC Vector Engine	Interactive	ssh dss

Job Scripts on ACES: Slurm

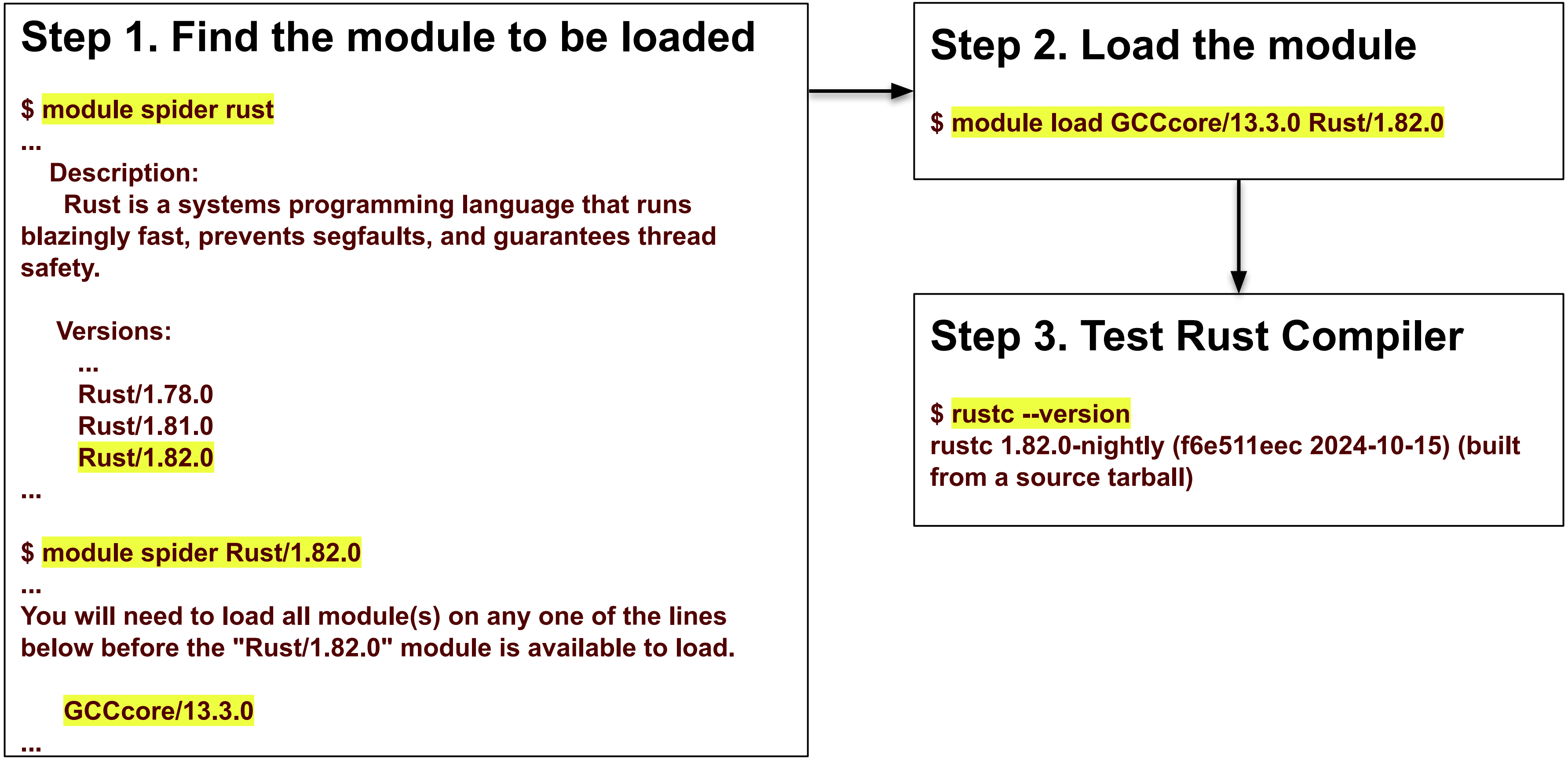
```
#!/bin/bash
#NECESSARY JOB SPECIFICATIONS
#SBATCH --job-name=my_job
#SBATCH --time=2-00:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=96
#SBATCH --mem=488G
#SBATCH --partition=gpu
#SBATCH --gres=gpu:h100:2
#SBATCH --output=stdout.%x.%j
#SBATCH --error=stderr.%x.%j

# load required module(s)
module purge
module load GCC/13.1.0

./my_program.py
```

These parameters describe the resources needed for your program to the job scheduler (Slurm)

Most of the ACES accelerators will be specified with either a partition or gres argument

Script to execute
(In this case, set up environment and launch an executable)

Using Rust Module on ACES



Step 1. Find the module to be loaded

```
$ module spider rust
...
   Description:
     Rust is a systems programming language that runs
blazingly fast, prevents segfaults, and guarantees thread
safety.

    Versions:
       ...
       Rust/1.78.0
       Rust/1.81.0
       Rust/1.82.0
...

$ module spider Rust/1.82.0
...
You will need to load all module(s) on any one of the lines
below before the "Rust/1.82.0" module is available to load.

     GCCcore/13.3.0
...
```

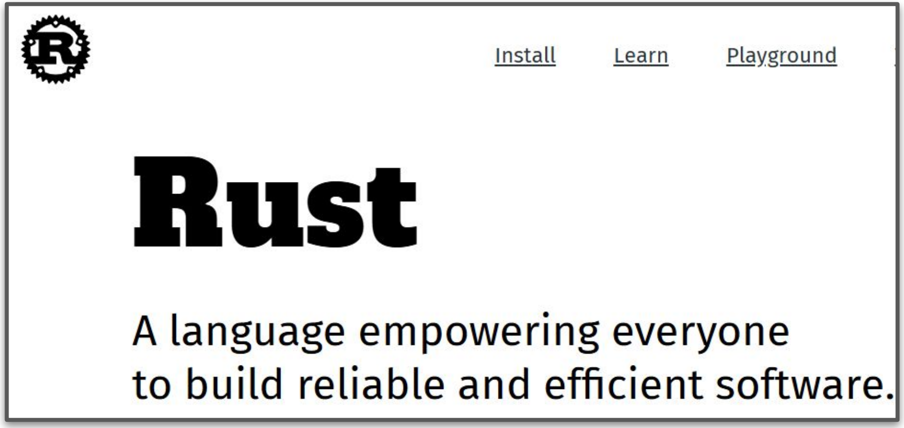
Step 2. Load the module

```
$ module load GCCcore/13.3.0 Rust/1.82.0
```

Step 3. Test Rust Compiler

```
$ rustc --version
rustc 1.82.0-nightly (f6e511eec 2024-10-15) (built
from a source tarball)
```

Part II.
Rust - What and Why?



Rust

A language empowering everyone
to build reliable and efficient software.

Rust Code Example: A First Look

main function
the entrance of a rust program

```rust
fn main() {
    greet();
    let result = add(5, 10);
    println!("The sum of 5 and 10 is: {}", result);
    let word = String::from("hello");
    let length = calculate_length(&word);
    println!("The length of '{}' is {}.", word, length);
}
```

Rust functions
handling various tasks

```rust
fn greet() {
    println!("Hello, World!");
}

fn add(a: i32, b: i32) -> i32 {
    a + b
}
fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Rust is a general-purpose programming language emphasizing **performance**, **type safety**, and **concurrency**.

- Created by **Graydon Hoare** as a personal project while working at **Mozilla Research** in 2006
- Officially sponsored by Mozilla in 2009
- First stable release in May 2015
- Latest stable release v1.82.0 as of Nov 8, 2024
- https://rust-lang.org
- *"A language empowering everyone to build reliable and efficient software."*

Major features of **Rust:**

- **Safe:** memory and thread safety guaranteed at compile time
- **Fast**: designed for high performance
- **General**: supporting different programming patterns
- **Memory control**: Efficient memory management through ownership system without garbage collection overhead
- **Concurrent:** Built-in support for safe concurrent programming with race condition prevention
- **Productive:** Developer-friendly environment with helpful error messages, type inference, and robust tooling

*Mostly importantly, for many of developers, **Rust** is the language of choice for **security-focused development**.*

"Pretty much like C/C++ with some strict rules to help compilation time error checks." --an anonymous Rust user on the first impression of Rust.

DARPA DEFENSE ADVANCED RESEARCH PROJECTS AGENCY ABOUT US /

› Defense Advanced Research Projects Agency › Our Research › Translating All C to Ru

Translating All C to Rust (TRACTOR)

Dr. Dan Wallach

- DARPA TRACTOR Program (2024): Eliminating Memory Safety Vulnerabilities Once and For All - DARPA initiates a new program to automate the translation of the world's highly vulnerable legacy C code to the inherently safer Rust programming language

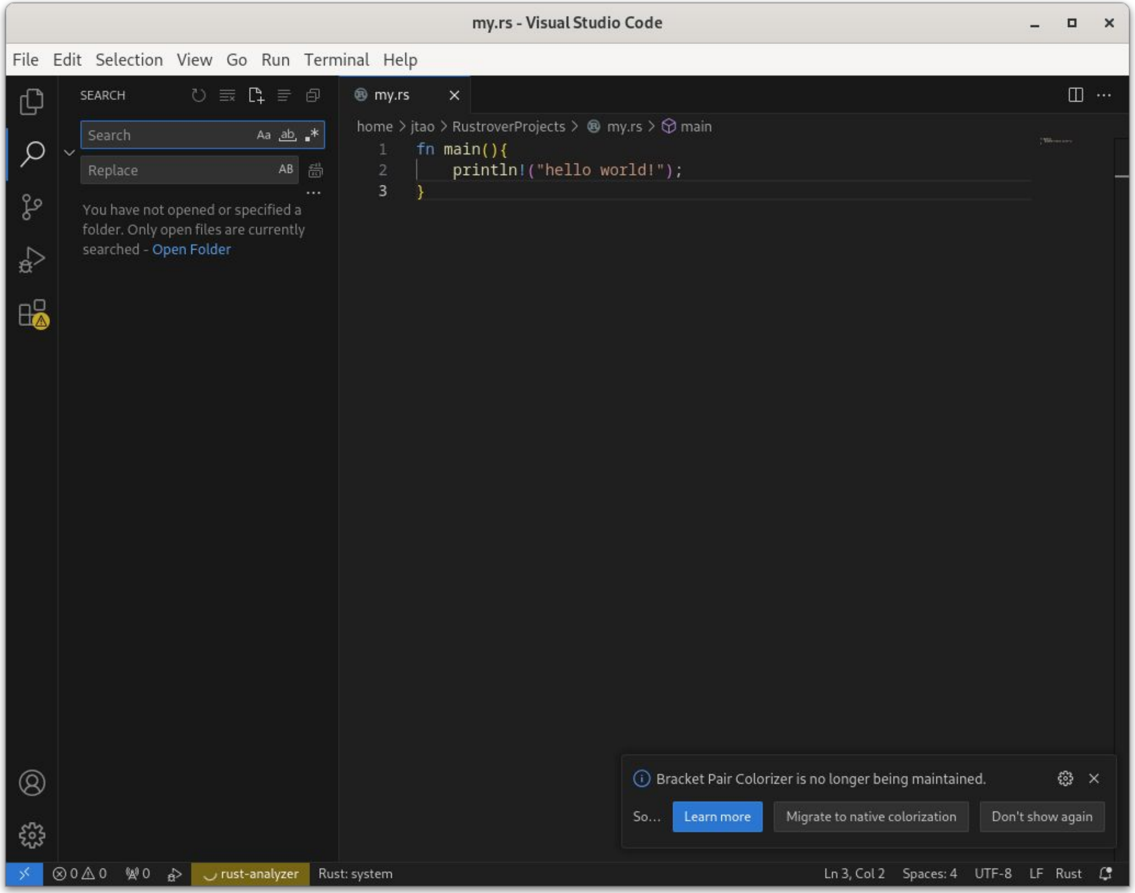
- DHS/CISA Report (2023): The Case for Memory Safe Roadmaps - Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously
- Whitehouse Report (2024): Fact Sheet: ONCD Report Calls for Adoption of Memory Safe Programming Languages and Addressing the Hard Research Problem of Software Measurability

Rust v.s. Python & C/C++

	Rust	Python	C++
Memory Management	No garbage collector	Automatic garbage collection	Manual memory management
Concurrency	Concurrency with ownership rules	Limited by Global Interpreter Lock (GIL)	Manual control over threads
Error Handling	Explicit error handling	Uses exceptions for error handling	Exception-based error handling
Performance	Comparable to C++	Slower due to dynamic typing and interpreted nature.	Highly optimized performance
Standard Library	Minimalistic	Comprehensive standard library	Extensive STL with a wide range of utilities.
Metaprogramming	Trait-based generics	Supports dynamic typing but lacks static template	Template metaprogramming with complex syntax
Type System	Strong, static typing	Dynamic typing at runtime	Strong, static typing
Safety	Memory-safe by design	Less emphasis on safety	Prone to memory safety issues
Learning Curve	Steeper learning curve	Gentler learning curve	Moderate to steep learning curve

VsCode

- VsCode has Rust plugins that supports the development of Rust.



You have not opened or specified a folder. Only open files are currently searched - Open Folder

```rust
fn main(){
    println!("hello world!");
}
```

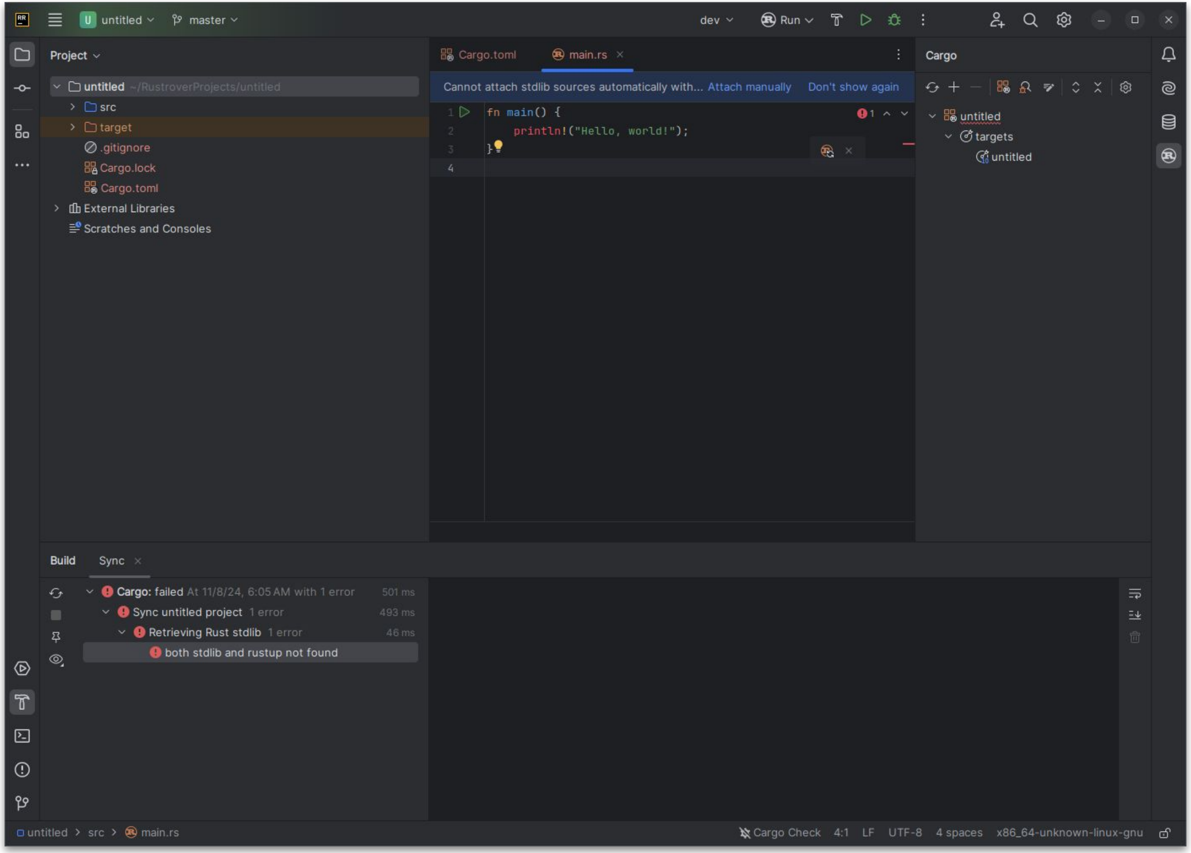
Bracket Pair Colorizer is no longer being maintained.

So...

Visual Studio Code

Rust IDE

- RustRover is an Integrated Development Environment (IDE) for Rust by JetBrains.



Cannot attach stdlib sources automatically with... Attach manually Don't show again

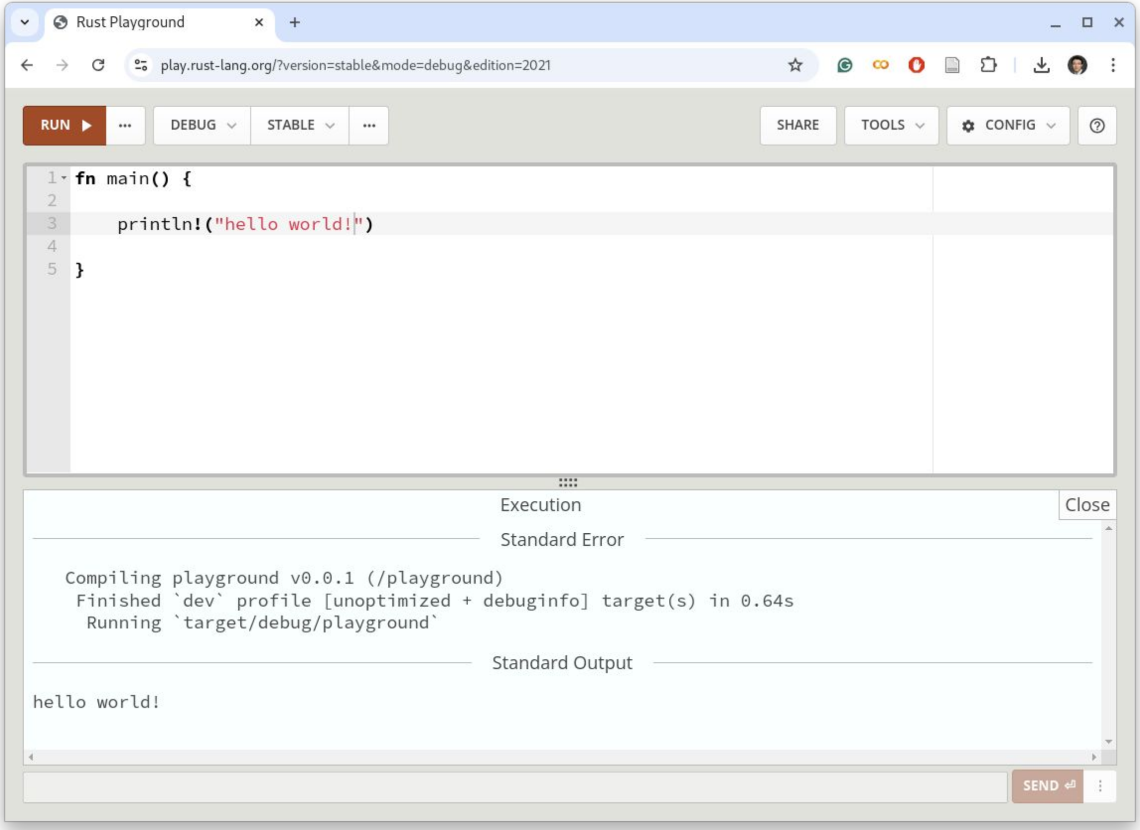
```
fn main() {
    println!("Hello, world!");
}
```

```
Cargo: failed At 11/8/24, 6:05 AM with 1 error       501 ms
  Sync untitled project 1 error                      493 ms
    Retrieving Rust stdlib 1 error                    46 ms
      both stdlib and rustup not found
```

RustRover: Rust IDE by JetBrains

Rust Playground

- An online platform to write, run, and share short Rust programs.
- Ideal for trying out Rust code quickly without needing to install anything.



```rust
fn main() {

    println!("hello world!")

}
```

Standard Error

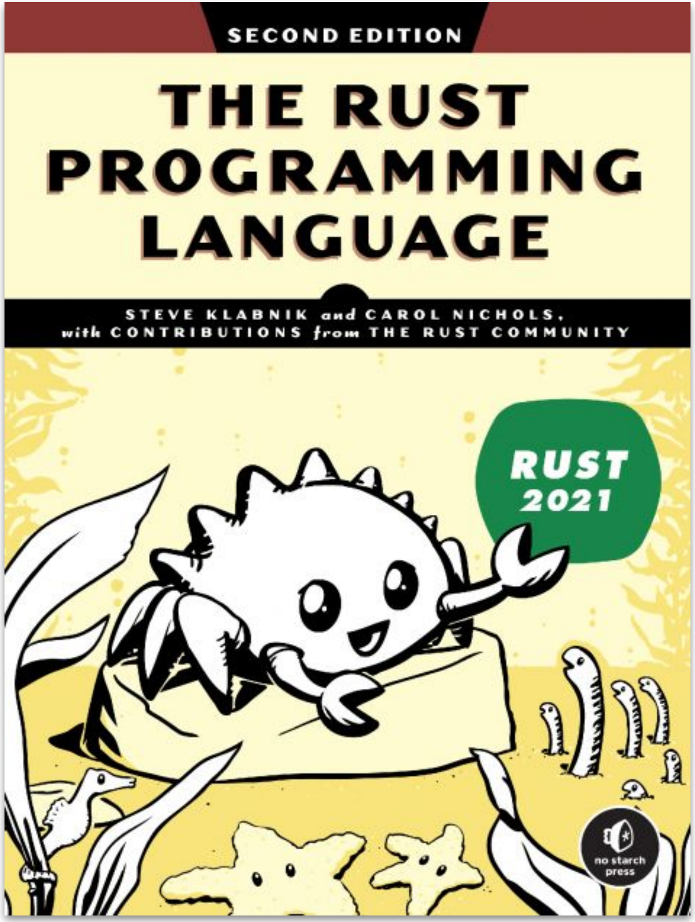
```
   Compiling playground v0.0.1 (/playground)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.64s
     Running `target/debug/playground`
```

Standard Output

```
hello world!
```

[Rust Playground](https://play.rust-lang.org)

Part III. Basics of Rust


SECOND EDITION
THE RUST PROGRAMMING LANGUAGE
STEVE KLABNIK and CAROL NICHOLS, with CONTRIBUTIONS from THE RUST COMMUNITY
RUST 2021
no starch press

Basic Data Types

The basic types of Rust include **int, float**, **bool,** and **char**.

1. **Integer Types**: Used to represent whole numbers.
   - Signed: i8, i16, i32, i64, i128, isize (size depends on the architecture)
   - Unsigned: u8, u16, u32, u64, u128, usize (size depends on the architecture)
2. **Floating-Point Types**: Used for decimal numbers.
   - f32 (32-bit floating point)
   - f64 (64-bit floating point, default)
3. **Boolean Type**: Represents a truth value.
   - bool (can be either true or false)
4. **Character Type**: Represents a single **Unicode** scalar value.
   - char (4 bytes, supports characters like 'a', '∞', etc.)

Compound Data Types

Compound types group multiple values into one type.

- **Enum**: encapsulate multiple values and different types of data within their variants
- **Tuples**: Can store multiple values of different types. Example: (i32, f64, bool) stores an integer, a float, and a boolean.
- **Arrays**: Fixed-size collections of elements of the same type. Example: [i32; 5] is an array of five 32-bit integers.

Compound Data Types - Enum

- **Enums** in Rust allow one to define a type that can take on a limited set of variants, making it easier to model state.

```
enum Direction {
    North,
    South,
    East,
    West,
}
```

Custom Data Types - I

- In Rust, **structures (or structs) are custom data types** that allow you to group together related data under one name.

```
struct Point {
    x: i32,
    y: i32,
}


let p1 = Point { x: 10, y: 20 };

println!("Point x: {}, y: {}", p1.x,
p1.y);
```

Custom Data Types - II

- **struct** in Rust is Similar to class in C++ or Python, but without methods for inheritance. More like struct in C.
- A fundamental part of Rust's type system for modeling more complex data.

```rust
struct Point {
    x: i32,
    y: i32,
}

let mut p2 = Point { x: 5, y: 10 };
p2.x = 15; // Now x is 15
```

Custom Data Types - Method Syntax

- To implement methods for the **Point** struct in Rust, one can implement functions within an **impl** block.

```
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    pub fn move_by(&mut self, dx: i32, dy: i32)
{
        self.x += dx;
        self.y += dy;
    }
}
fn main() {
   let mut p2 = Point { x: 5, y: 10 };
   p2.move_by(3, 4);
   println!("Point x: {}, y: {}", p2.x, p2.y);
}
```

Naming Rules for Variables - I

- Variable names must begin with a **letter or underscore**.

```
let age = 25;
let _temp = 30;
```

- Names can include any combinations of letters, numbers, underscores, and exclamation symbol. Some unicode characters could be used as well.

Naming Rules for Variables - II

- There is **no explicitly defined maximum length** for variable names.

- Rust is **case sensitive**. The variable name **A** is different from the variable name **a**.

- Variable names should be **descriptive**.

- **Avoid leading double underscores.**

Variable - Mutability

All Rust variables are **immutable** by default. Use **mut** to make a variable mutable.

```
let age = 35;
age = 36;
...
error[E0384]: cannot assign twice to immutable variable `age`
 --> src/main.rs:3:5
  |
2 |     let age = 35;
  |         --- first assignment to `age`
3 |     age = 36;
  |     ^^^^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
2 |     let mut age = 35;
  |         +++
...
```

Variable - Mutability

```
let mut age = 35;
age = 36;
```

- Immutability promotes **safer** and more **predictable** code.
- Immutable data can be **safely shared across threads** without needing locks, **improving performance** in concurrent programs.

Variable - Type Inference

- Rust automatically infers types, but you can also specify them explicitly.

```
let mut age: i32 = 35;
age = 40.5;

error[E0308]: mismatched types
 --> src/main.rs:4:11
  |
2 |     let mut age: i32 = 46;
  |                  --- expected due to this type
3 |     println!("{age}");
4 |     age = 234.6;
  |           ^^^^^ expected `i32`, found floating-point number
```

Variable - Shadow

- **Shadowing** allows you to declare a new variable with the same name as a previous variable. The new variable shadows the previous one, making the earlier variable inaccessible.

```rust
fn main() {
    let age = 35;
    println!("age = {}", age);
    let age = age + 1; // Shadows the previous 'age'
    println!("age = {}", age);
    let age = "Shadowed as a string"; // Shadows again with a different type
    println!("age = {}", age);
}
```

Primitive Data Structure

- **Array**: Fixed-size collection of elements of the same type.

```
let arr = [1, 2, 3, 4, 5]; // Array of size 5
```

- **Slice:** Dynamically sized view into a contiguous sequence (e.g., part of an array).

```
let slice = &arr[1..3]; // Slice of the array from index 1 to 2
```

Built-in Data Structure in std:: collections

Category	Data Structure	Description
Sequences	**Vec**	Growable array (dynamic vector).
Sequences	**VecDeque**	Double-ended queue (deque).
Sequences	**LinkedList**	Doubly linked list.
Maps	**HashMap**	Unordered key-value pairs with fast lookup.
Maps	**BTreeMap**	Ordered key-value pairs (sorted by keys).
Sets	**HashSet**	Unordered collection of unique values.
Sets	**BTreeSet**	Ordered collection of unique values (sorted).
Miscellaneous	**BinaryHeap**	Priority queue implemented with a binary heap.

Comments in Rust

Use comments to explain your code.

Single-line comments:

```
// This is a comment

let x = 5; // This is also a comment
```

Multi-line comments:

```
/*
This is a multi-line comment.
It spans multiple lines.
*/
let y = 10; // This is also a comment
```

Semicolons

Semicolon Usage:

- End of Statements
- Suppressing Expression Results

```
let x = 5; // Every statement ends with ;
println!("x = {}", x); // Every statement ends with ;
{
    x + 1 // No semicolon, so this value is returned
}
```

Arithmetic Operators

+	Addition
-	Subtraction/Negative
*	multiplication
/	division
%	mod

Arithmetic Expressions Samples

```
fn main() {
    let sum = 5 + 3;
    let difference = 10 - 4;
    let product = 6 * 7;
    let quotient = 20.0 / 3.0;

    println!("Sum: {}", sum);
    println!("Difference: {}", difference);
    println!("Product: {}", product);
    println!("Quotient: {:.2}", quotient);
}
```

Relational Operators

==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

* Rust allows overloading these operators for custom types by implementing traits from the std::cmp module:

- PartialEq for == and !=
- PartialOrd for <, >, <=, and >=

Boolean and Bitwise Operators

&&	Logical and
||	Logical or
!	Logical not
^	Bitwise XOR (Exclusive OR)
|	Bitwise OR
!	Negate
&	Bitwise And
>>	Right shift
<<	Left shift

NaN and Inf

- **NaN** is not equal to any value, including itself.
  - Operations involving **NaN** generally result in **NaN**.
  - To check if a value is NaN, you can use the .is_nan() method

```
let x = f64::NAN;

println!("{}", x.is_nan());
```

- **Inf** is infinity of type Float64.
  - **Inf** is equal to itself and greater than everything else except **NaN**.
  - **-Inf** is equal to itself and less then everything else except **NaN**.

```
let y = f64::INFINITY;
let z = f64::NEG_INFINITY;

println!("{}",
y.is_infinite());
```

Mathematical Constants

Constant	Description	Example
std::f64::consts::PI	π (Pi)	`let pi = std::f64::consts::PI;`
std::f64::consts::E	Euler's number (e)	`let e = std::f64::consts::E;`
std::f64::consts::SQRT_2	Square root of 2	`let sqrt2 = std::f64::consts::SQRT_2;`
std::f64::consts::FRAC_1_PI	1/π	`let frac_1_pi = std::f64::consts::FRAC_1_PI;`
std::f64::consts::FRAC_2_PI	2/π	`let frac_2_pi = std::f64::consts::FRAC_2_PI;`
std::f64::consts::FRAC_PI_2	π/2	`let frac_pi_2 = std::f64::consts::FRAC_PI_2;`
std::f64::consts::FRAC_PI_3	π/3	`let frac_pi_3 = std::f64::consts::FRAC_PI_3;`
std::f64::consts::FRAC_PI_4	π/4	`let frac_pi_4 = std::f64::consts::FRAC_PI_4;`
std::f64::consts::LN_2	Natural log of 2	`let ln_2 = std::f64::consts::LN_2;`
std::f64::consts::LN_10	Natural log of 10	`let ln_10 = std::f64::consts::LN_10;`
std::f64::consts::LOG2_E	Log base 2 of e	`let log2_e = std::f64::consts::LOG2_E;`
std::f64::consts::LOG10_E	Log base 10 of e	`let log10_e = std::f64::consts::LOG10_E;`

Built-in Math Libraries

Category	Functions
Basic Arithmetic	`+, -, *, /, %`
Powers and Roots	`.powi(n), .powf(f), .sqrt(), .cbrt()`
Exponential and Logarithmic	`.exp(), .exp2(), .ln(), .log10(), .log2(), .ln_1p()`
Trigonometric	`.sin(), .cos(), .tan()`
Inverse Trigonometric	`.asin(), .acos(), .atan(), .atan2(y)`
Hyperbolic Functions	`.sinh(), .cosh(), .tanh()`
Inverse Hyperbolic	`.asinh(), .acosh(), .atanh()`
Rounding Functions	`.ceil(), .floor(), .round(), .trunc(), .fract()`
Miscellaneous Functions	`.abs(), .signum(), .recip(), .hypot(y), .clamp(min, max)`

Expressions & Statements

- Rust programs are built with **expressions** and **statements**.
  - **Expressions** evaluate to a value.
  - **Statements** perform actions but do not return a value.

```rust
fn main() {
    // The literal `5` is an expression
    let x = 5;
    // `x + 2` is an expression
    let y = x + 2;
    // Function call `double(y)` is an
expression
    let z = double(y);
    println!("x: {}, y: {}, z: {}", x, y, z);
}

// `n * 2` is an expression
fn double(n: i32) -> i32 {
    n * 2
}
```

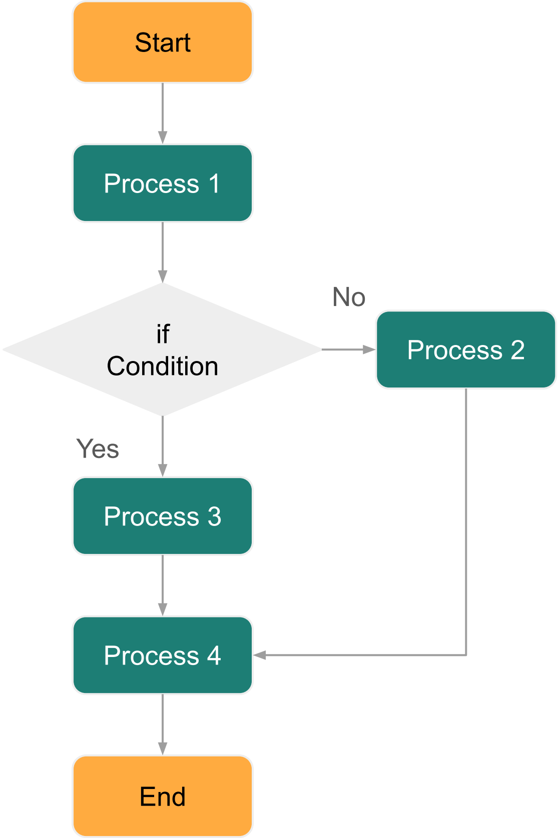
Blocks as Expressions

- A **Block '{...}'** in Rust can act as an expression.
- The last expression inside the block determines the block's value.
- Adding a semicolon turns an expression into a statement, which discards its result.

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

    println!("y = {}", y);
}
```

Controlling Blocks in Rust

- Rust provides several constructs to control the flow of execution:
  - **Conditional statements**: `if`, `else if`, `else`, and `match`.
  - **Looping constructs**: `loop`, `while`, and `for`.
- These blocks can be used to manage data flow, decision-making, and repetition in business processes.



Conditional Statements - if

- Conditional statements allow you to execute code based on certain conditions.
- Rust supports:
  - **if** and **else** for basic conditionals.
  - **match** for pattern matching and more complex control flow.

```
if condition {
    // Code executed if the condition is true
}
```

```
fn main() {
    let number = 5;

    if number > 0 {
        println!("The number is positive");
    }
}
```

Conditional Statements - if-else

- The **if-else** statement lets you define alternative actions when the condition is false.
- **if-else** can be used as an Expression

```
if condition {
    // Code executed if the condition is true
} else {
    // Code executed if the condition is false
}

fn main() {
    let number = -3;
    if number > 0 {
        println!("The number is positive");
    } else {
        println!("The number is not positive");
    }
     let positive = if number >0 { "Yes" } else {
"No" };
    println!("Is the number positive ? {}",
positive);
}
```

Conditional Statements - else if

- The **else if** statement lets you chain multiple conditions using **else if**.

```
if condition1 {
// Code executed if condition1 is true
} else if condition2 {
// Code executed if condition2 is true
} else {
// Code executed if all conditions are false
}
```

```
fn main() {
    let number = 0;

    if number > 0 {
        println!("The number is positive");
    } else if number == 0 {
        println!("The number is zero");
    } else {
        println!("The number is negative");
    }
}
```

Conditional Statements - match

- The **match** statement in Rust allows for more complex branching based on pattern matching.
- It’s similar to a **switch** statement in other languages but more powerful.
- You can match multiple patterns using the pipe (|) symbol.

```
match value {
    pattern1 => action1,
    pattern2 => action2,
// `_` acts as a catch-all pattern
(optional)
    _ => default_action,
}
```

```
fn main() {
    let num = 1;
    match num {
        1 | 2 | 3 => println!("Number
is between one and three"),
        _ => println!("Number is
something else"),
    }
}
```

Conditional Statements - match with enums

- The **match** statement usually used together with **enums.**

```rust
enum Direction {
    North,
    South,
    East,
    West,
}
fn main() {
    let heading = Direction::North;
    match heading {
        Direction::North => println!("Heading North"),
        Direction::South => println!("Heading South"),
        Direction::East  => println!("Heading East"),
        Direction::West  => println!("Heading West"),
    }
}
```

Iterative Logic in Rust

- Loops (loop, while, and for) are used to repeat actions, such as processing multiple items or retrying operations.
- Rust provides three types of loops:
  - **loop**: Infinite loop, must be explicitly broken.
  - **while**: Repeats while a condition is true.
  - **for**: Iterates over a collection or a range.

```
loop {
    // Code that runs
infinitely unless break is
called
}

while condition {
    // Code that runs while the
condition is true
}

for variable in
collection_or_range {
    // Code that runs for each
element in the collection or
range
}
```

Iterative Logic with loop

- The loop statement creates an infinite loop.
- You can **use break to exit** the loop when a condition is met.
- You can return values from a loop using **break with a value**.

```
fn main() {

    let mut counter = 0;

    let result = loop {
        counter += 1;
        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {}", 
result); // Output: The result is 20
}
```

Iterative Logic with while

- The while loop runs as long as a condition is true.
- Like while statements in other languages.

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{}!", number);
        number -= 1;
    }

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

Iterative Logic with for

- The for loop iterates over **a range** or **an iterator**, making it ideal for looping over collections like arrays or vectors.

```
fn main() {
    for i in 1..4 {
        println!("i = {}", i);
    }
}
```

```
fn main() {
    let arr = [10, 20, 30];
    for element in arr.iter()
{
        println!("Element:
{}", element);
    }
}
```

Using .enumerate() in Loops

- The **.enumerate()** method returns both the index and the value of each item in an iterator.

```
fn main() {
    let numbers = [100, 200, 300];

    for (index, value) in
numbers.iter().enumerate() {
        println!("Index: {}, Value:
{}", index, value);
    }
}
```

Comparison of Loop Types

	Description	Use Case
loop	Infinite loop that must be explicitly broken	Use when you need an infinite or manually controlled loop
while	Loops while a condition is true	Use when you don’t know how many iterations are needed
for	Iterates over collections or ranges	Use when iterating over collections or ranges

Definition of Functions - I

- **Functions** are reusable blocks of code that perform specific tasks.
- Defined using the **fn** keyword.
- Every Rust program has at least one function: **main**.
- Functions use **snake_case** for naming (all lowercase, words separated by underscores).
- Rust does not support **optional** or **default arguments** in functions directly, but it supports **Option<T> type.**

```rust
fn main() {
    println!("Hello, world!");
    another_function(42);
}
```

```rust
fn another_function(x: i32) {
    println!("x = {x}");
}
```

Definition of Functions - II

- Functions can take **parameters**, which are variables passed into the function.
- **Parameters** must have a type specified (e.g., i32).
- **Multiple parameters** can be passed, separated by commas.
- Functions can **return values**.
- The **return type is specified** after the arrow (->).
- The last expression in the function block is implicitly returned (no need for return).

```
fn main() {
    println!("Hello, world!");
    let x =
another_function(42);
    println!("returned x =
{x}");
}

fn another_function(x: i32) ->
i32 {
    println!("x = {x}");
    x // equivalent to return
x;
}
```

Anonymous Functions - Closure I

Rust supports anonymous functions or functions without a name through a feature called **closures**.

- **Defined using vertical pipes** (|) to enclose the parameters, followed by the closure body.
- **Capture variables** from the scope in which they are defined.
- **Assigned to variables** and passed around as arguments to other functions.

```
let closure_name = |parameters|
{
    // closure body
};
```

```
fn main() {
    let name =
String::from("Alice");

    let greet = ||
println!("Hello, {}!", name);

    greet();
}
```

Anonymous Functions - Closure II

Rust supports anonymous functions or functions without a name through a feature called **closures**.

- A closure can have **zero or more parameters**.
- Rust can **infer the types** of the parameters and return values of closures, though you can specify them if needed.

```
fn main() {
// A closure without a parameter
    let print_text = ||
println!("Hello from Closure!");

    print_text();
}
```

```
fn main() {
// A closure with a parameter
    let add_one = |x: i32| x + 1;
    let result = add_one(5);
    println!("Result = {}",
result);
}
```

Function of Function - I

- Rust allows defining **higher-order functions** by passing or returning both regular functions and closures.
- This flexibility allows Rust to support functional programming patterns alongside its systems programming capabilities.

```rust
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32,
arg: i32) -> i32 {
    f(f(arg))
}

fn main() {
    let result =
do_twice(add_one, 5); // Pass
`add_one` as an argument
    println!("The result is: {}",
result);
}
```

Function of Function - II

- Function pointers (fn) are used for regular functions, while closures use traits like **`Fn`**, **`FnMut`**, or **`FnOnce`**.

```
// F is any type that implements the Fn trait
fn do_twice<F>(f: F, arg: i32) -> i32
where F: Fn(i32) -> i32,
{
    f(f(arg))
}

fn main() {
// Define a closure that adds 2
    let closure = |x| x + 2;
    let result = do_twice(closure, 5);
    println!("The result is: {}", result);
}
```

Function of Function - II

- One can return closures using trait objects (e.g., **`Box<dyn Fn()>`**) when necessary.

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1) // Return a closure that adds 1
}


fn main() {
    let closure = returns_closure();
    let result = closure(5);
    println!("The result is: {}", result);
// Output: The result is: 6
}
```

Part IV. Advanced Topics of Rust

- Ownership & Borrowing
- Lifetimes
- Error Handling
- Traits & Trait Objects
- Generics
- Standard Libraries

Ownership

- Rust's memory management system is built around two key concepts: **ownership** and **borrowing**.
- Ownership
  - Each value has a single owner
  - Only one owner at a time
  - Automatic cleanup

```
fn main() {
// s owns the String
    let s1 = String::from("hello");
// ownership has been transferred to s1
    let s2 = s1;     // let s2 = s1.clone();
    //println!("{}", s1);

    let x = 5;
// x is copied because integers implement the
Copy trait
    makes_copy(x);
    println!("{}", x);
}

fn makes_copy(some_integer: i32) {
    println!("{}", some_integer);
}
```

Slice Type

- Slices enables referencing a contiguous sequence of elements in a collection rather than the whole collection.

```
fn main() {
    let arr = [10, 20, 30, 40, 50];
    let slice = &arr[1..4];
    println!("Slice: {:?}", slice); //
Output: [20, 30, 40]
}
```

Immutable Borrowing

- Borrowing allows references to a value without transferring ownership.
  - Immutable Borrowing (`&T`)
  - Mutable Borrowing (`&mut T`)

```rust
fn main() {
    let mut num = 42;

    let borrowed_immutable = &num;  // Immutable borrow
    println!("Immutable borrow: {}", borrowed_immutable);

    let borrowed_mutable = &mut num; // Mutable borrow
    *borrowed_mutable += 10;
    println!("Mutable borrow: {}", borrowed_mutable);

    // println!("Immutable borrow again: {}", borrowed_immutable);
}
```

Mutable Borrowing

- Both the original declaration and borrowing need to be mutable.

```
fn main() {
    let mut s = String::from("hello");  // Create a mutable String with
the value "hello"

    change(&mut s);  // Pass a mutable reference of the string `s` to
the `change` function
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");  // Modify the string by appending
", world" to it
}
```

Lifetimes

- A **lifetime** refers to the duration for which a reference is valid.
- Lifetimes are closely tied to **scopes**. A reference must not outlive its scope.
- In Rust, lifetimes ensure that references do not outlive the data they point.
- Lifetimes are crucial for **memory safety**.

```rust
fn main() {
    let r;
    {
        let x = 5;
        r = &x;  // Error: `x`
does not live long enough
    }
    println!("r: {}", r);
}
```

Lifetimes - Annotations

- Syntax: Lifetimes are denoted using apostrophes (`'a, 'b,` etc.).
- Sometimes, different parameters may have different lifetimes.

```
fn longest1<'a>(x: &'a str, y: &'a
str) -> &'a str {
    if x.len() > y.len() { x } else
{ y }
}

fn longest2<'a, 'b>(x: &'a str, y:
&'b str) -> &'a str {
    if x.len() > y.len() { x } else
{ y }

}
```

Lifetimes - Elision Rules

- Rust can **infer lifetimes** in certain cases to reduce verbosity.
- Elision Rules:
    - Each input reference gets its own lifetime parameter.
    - If there is exactly one input lifetime, it is assigned to all output lifetimes.

```
fn main() {
    let r;
    {
        let x = 5;
        r = &x;  // Error: `x`
does not live long enough
    }
    println!("r: {}", r);
}
```

Lifetimes - Structs

When structs hold references, their lifetimes must be annotated.

```rust
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael.");
    let first_sentence = novel.split('.').next().expect("Could not find a '.'");
    let excerpt = ImportantExcerpt { part: first_sentence };
    println!("{}", excerpt.part);
}
```

Error Handling - Result & Panic!

- Recoverable Errors: Handled using the **`Result<T, E>`** enum.
- Unrecoverable Errors: Handled using the **`panic!`** macro where continuing execution would be unsafe or nonsensical.

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

```
fn main() {
    panic!("crash and burn");
}
```

Error Handling - Result Example

- The Result enum is part of Rust's standard library and is widely used across many Rust programs for handling errors in a type-safe manner.

```rust
fn divide(dividend: f64, divisor: f64) -> Result<f64,
String> {
    if divisor == 0.0 {
        Err(String::from("Division by zero"))
    } else {
        Ok(dividend / divisor)
    }
}

fn main() {
    match divide(10.0, 2.0) {
        Ok(result) => println!("Result: {}", result),
        Err(error) => println!("Error: {}", error),
    }

    match divide(10.0, 0.0) {
        Ok(result) => println!("Result: {}", result),
        Err(error) => println!("Error: {}", error),
    }
}
```

Traits in Rust

- Traits define **shared behavior** across types.
- Similar to **interfaces** in other languages.
- Traits allow you to write **generic** and **reusable** code.

```
struct NewsArticle {
    headline: String,
    author: String,
    content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {}", self.headline, self.author)
    }
}

struct Tweet {
    username: String,
    content: String,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Implementation Traits

- **Traits** are defined using the **trait** keyword.
- Define methods without implementation inside the trait.
- Use **impl** to provide implementations of the trait methods.
- Traits can contain **multiple** methods.

```
// Define the Greet trait
trait Greet {
    fn say_hello(&self);
}

// Define the Person struct
struct Person {
    name: String,
}

// Implement the Greet trait for Person
impl Greet for Person {
    fn say_hello(&self) {
        println!("Hello, my name is {}", self.name);
    }
}

fn main() {
    let john = Person { name: String::from("John")
};
    john.say_hello();
}
```

Trait Objects

- **`dyn`** keyword followed by the trait name is used to create a trait object.
- Trait objects must be used behind some kind of pointer, such as **`&dyn Trait`**, **`Box<dyn Trait>`**, or **`Rc<dyn Trait>`**.

```
fn notify(item: &dyn Summary) {
    println!("Breaking news! {}",
item.summarize());
}
```

Here, **`&dyn Summary`** is a reference to a trait object. Rust will perform dynamic dispatch to call the correct implementation of the **`summarize`** function at runtime.

Generics in Rust

Generics in Rust allow one to write flexible, reusable code that works with different types while maintaining type safety. Rust's generics are similar to C++ templates.

- **Generic Functions**: Functions that can accept **parameters of any type**.
- **Generic Structs**: Structs that can store **data of any type**.
- **Generic Enums**: Enums that can hold **values of different types**.
- **Trait Bounds**: Constraints on generics to ensure they **implement specific traits**.

Generics Struct

A **struct** Point that can hold coordinates of any type.

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let int_point = Point { x: 5, y: 10 };
    let float_point = Point { x: 1.0, y: 4.0
};

    println!("int_point: ({}, {})",
int_point.x, int_point.y);
    println!("float_point: ({}, {})",
float_point.x, float_point.y);
}
```

Generics Enum

The **Result enum** is a common example of generics in Rust. It can handle success (**`Ok(T)`**) or failure (**`Err(E)`**).

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

fn divide(a: i32, b: i32) -> Result<i32, &'static str>
{
    if b == 0 {
        Err("Division by zero")
    } else {
        Ok(a / b)
    }
}

fn main() {
    match divide(10, 2) {
        Ok(result) => println!("Result is {}", result),
        Err(err) => println!("Error: {}", err),
    }
}
```

Rust Standard Library

The Rust Standard Library (std) provides a comprehensive set of modules and utilities that serve as the foundation for Rust programs.

- Collection of **core utilities**, **data structures**, and **functions**.
- Provides tools for **systems programming**, **concurrency**, **networking**, and **more**.
- Designed to be **fast**, **safe**, and **efficient**.
- A **foundation** for building complex Rust applications.

`std::io`: Input/output operations.
`std::fs`: File system operations.
`std::thread`: Concurrency with threads.
`std::time`: Time management.
`std::collections`: Data structures.
`std::env`: Environment handling.
`std::sync`: Synchronization utilities.
`...`

Basic Input and Output with std::io

- Handles reading from and writing to standard input, output, and files.
- Provides **`stdin()`**, **`stdout()`**, and **`stderr()`** for console interaction.

```rust
use std::io::{self, Write};

fn main() {
    print!("Enter your name: ");
    io::stdout().flush().unwrap();

    let mut name = String::new();
    io::stdin().read_line(&mut name).unwrap();
    println!("Hello, {}!", name.trim());
}
```

File Operations with std::fs

- Create, read, write, and manage files and directories.
- Common functions: `File::create, read_to_string, write.`

```
use std::fs;

fn main() {
    let content = "Hello, Rust!";

    fs::write("hello.txt",
content).expect("Unable to write file");

    let read_content =
fs::read_to_string("hello.txt").expect("Unable
to read file");

    println!("{}", read_content);
}
```

Concurrency with std::thread

- Enables multithreading with lightweight threads.
- Functions: **`spawn`** to create new threads, **`join`** to wait for completion.

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..5 {
            println!("Thread says: {}", i);
        }
    });
    for i in 1..5 {
        println!("Main says: {}", i);
    }
    handle.join().unwrap();
}
```

Managing Time with std::time

- Functions for working with system time and durations.
- Useful for delays, performance measurement, etc.

```rust
use std::thread;
use std::time::Duration;

fn main() {
    println!("Sleeping for 2 seconds...");
    thread::sleep(Duration::from_secs(2));
    println!("Done!");
}
```

Data Structures in std::collections

- Includes **`Vec`**, **`HashMap`**, **`HashSet`**, **`LinkedList`**, and **`more`**.
- Used to store and manage data effectively.

```rust
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();
    scores.insert("Alice", 10);
    scores.insert("Bob", 20);

    for (player, score) in &scores {
        println!("{}: {}", player, score);
    }
}
```

Environment Management with std::env

- Functions to retrieve environment variables, command-line arguments.
- Common methods: **`args, var, set_var`**.

```rust
use std::env;

fn main() {
    let args: Vec<String> =
env::args().collect();
    println!("Command-line arguments: {:?}",
args);

    env::set_var("MY_VAR", "Hello,
Environment!");
    println!("MY_VAR: {}",
env::var("MY_VAR").unwrap());
}
```

Synchronization with std::sync

- Tools for managing data safely across threads.
- Includes **`Mutex`**, **`Arc`** (atomic reference count), **`RwLock`**.

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

Using Error Handling in the Standard Library

- The standard library provides Result and Option for handling errors.
- **`Result<T, E>`** is used for error handling, while **`Option<T>`** is for optional values.

```rust
fn divide(a: f64, b: f64) -> Result<f64, &'static
str> {
    if b == 0.0 {
        Err("Cannot divide by zero")
    } else {
        Ok(a / b)
    }
}

fn main() {
    match divide(10.0, 2.0) {
        Ok(result) => println!("Result: {}", result),
        Err(error) => println!("Error: {}", error),
    }
}
```

Online Resources

- The Rust Programming Language
- Programming Rust
- Rust for Rustaceans
- GitHub - rust-lang/rustlings: :crab: Small exercises to get you used to reading and writing Rust code!
- Welcome to Comprehensive Rust
- Rust on Exercism
- Let's Get Rusty - YouTube
- Rust Events

Acknowledgments

- The slides are created based on the materials from Rust official website.
- Support from Texas A&M Institute of Data Science (TAMIDS), and Texas A&M High Performance Research Computing (HPRC).
- Support from NSF OAC Award #2019129 - MRI: Acquisition of FASTER - Fostering Accelerated Sciences Transformation Education and Research
- Support from NSF OAC Award #2112356 - Category II: ACES - Accelerating Computing for Emerging Sciences

HPRC Survey

https://u.tamu.edu/hprc_shortcourse_survey

HPRC Survey

HPRC Survey