Rust Programming Topics: Sample Programs for Presentation

Illustrative Code Examples with Enhanced Syntax Highlighting

Grok

September 16, 2025

1. Introduction
2. Ownership, Borrowing, and Lifetimes
3. Writing, Compiling, and Executing Basic Programs
4. Variables, Data Types, Constants, and Mutability
5. Functions, Parameters, Return Values, and Modules
6. Enums, Pattern Matching, Option/Result Types
7. Traits for Common Behavior
8. File I/O
9. Rusts Role in Modern Software Development
10. Error Handling, Structs, Enums, Pattern Matching
11. Reusable Code with Traits, Generics, Collections, Iterators
12. Strings and String Manipulation
13. Collections and Iterators
14. Structs and Methods
15. Async Programming

Introduction

This presentation provides one illustrative Rust program for each major topic from the course objectives, outcomes, and additional concepts. Each slide includes:

- A brief topic explanation.
- A sample program (concise, with enhanced syntax highlighting).
- Highlights of special features (e.g., Rust's unique safety mechanisms).

Programs are optimized for slide readability and tested on Rust Playground. Wider layout and explicit Rust keyword highlighting prevent overflow and ensure visibility.

Ownership, Borrowing, and Lifetimes

Special Highlight: Ownership ensures memory safety without GC; borrowing prevents data races; lifetimes avoid dangling references.

```rust
fn main() {
    let s = String::from("hello"); // s owns the string
    takes_ownership(s); // ownership moved
    // println!("{}", s); // error: s moved!

    let x = 5;
    makes_copy(x); // x copied (primitives implement Copy)
    println!("{}", x); // ok

    let s2 = String::from("world");
    let len = calculate_length(&s2); // borrow (immutable ref)
    println!("Length of '{}' is {}.", s2, len); // s2 still owned
}

fn takes_ownership(some_string: String) { // owns some_string
    println!("{}", some_string);
} // dropped here

fn makes_copy(some_integer: i32) { // copy
```

Writing, Compiling, and Executing Basic Programs

Special Highlight: Use ‘cargo‘ for building; compiler enforces safety at compile time.

```rust
fn main() {
    println!("Hello, Rust!"); // Basic output
    let sum = add(5, 10); // Call user-defined function
    println!("Sum: {}", sum);
    if sum > 10 { // Control structure
        println!("Greater than 10");
    } else {
        println!("10 or less");
    }
}

fn add(a: i32, b: i32) -> i32 { // Function with params/return
    a + b
}
```

Variables, Data Types, Constants, and Mutability

Special Highlight: Immutability by default promotes safe code; shadowing allows redeclaration.

```
const MAX: i32 = 100; // Constant

fn main() {
    let x: i32 = 5; // Explicit type, immutable
    println!("x: {}", x);
    let mut y = 10; // Mutable
    y = 15;
    println!("y: {}", y);
    let x = "shadowed"; // Shadowing (different type)
    println!("x now: {}", x);
    let arr = [1, 2, 3]; // Array
    let tup = (4, 5.0, 'a'); // Tuple
    println!("Array[0]: {}, Tuple.2: {}", arr[0], tup.2);
}
```

Functions, Parameters, Return Values, and Modules

Special Highlight: Modules organize code; pub for visibility; Rustdoc for documentation.

```rust
pub mod math {
    /// Adds two numbers.
    pub fn add(a: i32, b: i32) -> i32 {
        a + b
    }
}

fn main() {
    use math::add; // Use module
    let result = add(3, 7);
    println!("Result: {}", result);
}
```

Enums, Pattern Matching, Option/Result Types

Special Highlight: Enums with data; pattern matching exhaustively handles cases, preventing errors.

```rust
enum Shape {
    Circle(f64), // Radius
    Rectangle(f64, f64), // Width, height
}

fn area(shape: Shape) -> f64 {
    match shape {
        Shape::Circle(r) => 3.14 * r * r,
        Shape::Rectangle(w, h) => w * h,
    }
}

fn main() {
    let circle = Shape::Circle(5.0);
    println!("Area: {}", area(circle));
    let option: Option<i32> = Some(42);
    if let Some(value) = option {
        println!("Value: {}", value);
    }
```

Traits for Common Behavior

Special Highlight: Traits enable polymorphism without inheritance; default implementations possible.

```
trait Greet {
    fn greet(&self) -> String;
}

struct Person {
    name: String,
}

impl Greet for Person {
    fn greet(&self) -> String {
        format!("Hello, {}!", self.name)
    }
}

fn main() {
    let p = Person { name: String::from("Alice") };
    println!("{}", p.greet());
}
```

File I/O

Special Highlight: Uses Result for error handling; safe file operations.

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

fn main() -> io::Result<()> {
    let mut file = File::create("hello.txt")?;
    file.write_all(b"Hello, Rust!")?;
    let mut contents = String::new();
    let mut file = File::open("hello.txt")?;
    file.read_to_string(&mut contents)?;
    println!("{}", contents);
    Ok(())
}
```

Rusts Role in Modern Software Development

Special Highlight: Memory-safe concurrency; used in web, embedded, systems; no GC for performance.

No specific program; refer to overall examples demonstrating safety and efficiency.

Error Handling, Structs, Enums, Pattern Matching

Special Highlight: Structs with impl for methods; combines with enums for robust types.

```rust
#[derive(Debug)]
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn distance(&self, other: &Point) -> f64 {
        ((self.x - other.x).powi(2) + (self.y - other.y).powi(2)).sqrt()
    }
}

fn main() {
    let p1 = Point { x: 0.0, y: 0.0 };
    let p2 = Point { x: 3.0, y: 4.0 };
    println!("Distance: {}", p1.distance(&p2));
}
```

Reusable Code with Traits, Generics, Collections, Iterators

Special Highlight: Generics for type-agnostic code; iterators for efficient looping.

```rust
fn print_collection<T: std::fmt::Debug>(coll: &[T]) {
    for item in coll.iter() {
        println!("{:?}", item);
    }
}

fn main() {
    let vec = vec![1, 2, 3];
    print_collection(&vec);
    let doubled: Vec<i32> = vec.iter().map(|&x| x * 2).collect();
    println!("{:?}", doubled);
}
```

Strings and String Manipulation

Special Highlight: UTF-8 encoded; slices (str) vs. owned (String).

```rust
fn main() {
    let mut s = String::from("Hello");
    s.push_str(", Rust!");
    println!("{}", s);
    let trimmed = s.trim();
    println!("{}", trimmed);
    if s.contains("Rust") {
        println!("Contains 'Rust'");
    }
    let replaced = s.replace("Rust", "World");
    println!("{}", replaced);
}
```

Collections and Iterators

Special Highlight: Vec is dynamic array; iterators are lazy and composable.

```
fn main() {
    let mut vec = vec![1, 2, 3, 4, 5];
    vec.push(6);
    let sum: i32 = vec.iter().sum();
    println!("Sum: {}", sum);
    let evens: Vec<&i32> = vec.iter().filter(|&&x| x % 2 == 0).collect();
    println!("Evens: {:?}", evens);
}
```

Structs and Methods

Special Highlight: Associated functions (static methods); impl blocks.

```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
    fn new(width: u32, height: u32) -> Self {
        Self { width, height }
    }
}

fn main() {
    let rect = Rectangle::new(10, 20);
    println!("Area: {}", rect.area());
}
```

Async Programming

Special Highlight: async/await for non-blocking I/O; requires runtime like tokio.

```rust
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;
    println!("Listening");
    loop {
        let (mut socket, _) = listener.accept().await?;
        tokio::spawn(async move {
            let mut buf = [0; 1024];
            socket.read(&mut buf).await.unwrap();
            // Handle request
        });
    }
}
```

(Note: Requires ‘tokio = version = "1", features = ["full"] ‘ in Cargo.toml.)

Unsafe Rust

Special Highlight: Unsafe blocks for raw pointers, FFI; bypasses safety checks use cautiously.

```
fn main() {
    let mut num = 5;
    let r1 = &num as *const i32; // Raw pointer
    let r2 = &mut num as *mut i32;
    unsafe {
        println!("r1: {}", *r1);
        *r2 = 10;
        println!("r2: {}", *r2);
    }
}
```

Macros

Special Highlight: Declarative (macro$_{r}ules!$)$orprocedural$; $codegenerationatcompiletime$.

```
macro_rules! say_hello {
    () => {
        println!("Hello!");
    };
}

fn main() {
    say_hello!();
}
```

Threading

Special Highlight: Safe concurrency; Send/Sync traits prevent data races.

```rust
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..5 {
            println!("Spawned thread: {}", i);
        }
    });
    for i in 1..3 {
        println!("Main thread: {}", i);
    }
    handle.join().unwrap();
}
```

Unit Testing

Special Highlight: Built-in testing framework; [test] attribute.

```rust
fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_add() {
        assert_eq!(add(2, 3), 5);
    }
    #[test]
    #[should_panic]
    fn test_panic() {
        panic!("Expected panic");
    }
}
```

(Run with ‘cargo test‘.)