Why Safe Programming Matters and Why Rust?

Deepu K Sasidharan

@deepu105 | deepu.tech

okta

Deepu K Sasidharan

JHipster co-lead developer

Creator of KDash

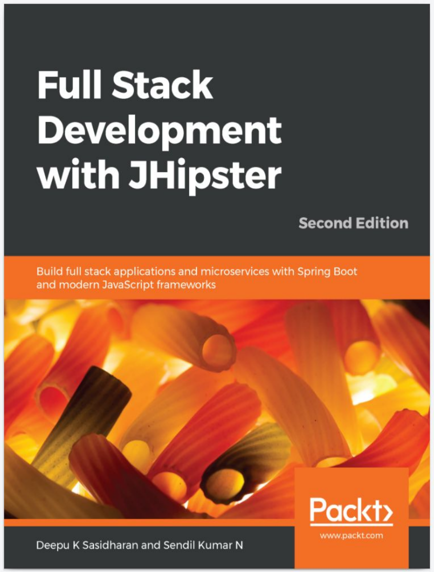
Developer Advocate @ Okta

OSS aficionado, author, speaker, polyglot dev

@deepu105

deepu.tech

DEV deepu105



What is safe programming?
More precisely,
What is a safe programming language?

Safe programming

Programming Safety = Memory safety + Type safety + Thread safety

Memory safety

- Predictable behaviour (No undefined behaviours)
- No unauthorized/invalid pointer access
- No free after use errors
- No double free errors
- No buffer overflows
- Null safety
    - Not applicable to all languages but still an issue in many
    - The worst invention in programming - as per its inventor

https://deepu.tech/memory-management-in-programming/

Type safety

- Correctness of data type is ensured
- No need to check at runtime
- Memory safety is required for type safety

Thread safety

- No race conditions
- No memory corruption
- Fearless concurrency

Why does it matter?

CVE galore from memory safety issues

- About 70% of all CVEs at Microsoft are memory safety issues
- Two-thirds of Linux kernel vulnerabilities come from memory safety issues
- An Apple study found that 60-70% of vulnerabilities in iOS and macOS are memory safety vulnerabilities
- Google estimated that 90% of Android vulnerabilities are memory safety issues
- 70% of all Chrome security bugs are memory safety issues
- An analysis of 0-days that were discovered being exploited in the wild found that more than 80% of the exploited vulnerabilities were memory safety issues
- Some of the most popular security issues of all time are memory safety issues
  - Slammer worm, WannaCry, Trident exploit, HeartBleed, Stagefright, Ghost

Security issues from thread safety

- Information loss caused by a thread overwriting information from another
  - Pointer corruption that allows privilege escalation or remote execution
- Integrity loss due to information from multiple threads being interlaced
  - The best-known attack of this type is called a TOCTOU (time of check to time of use) attack caused by race conditions

Security issues from type safety

- Low level exploits are possible in languages that are not type safe.
- Type safety is important for memory safety as type safety issues can lead to memory safety issues

Why Rust?

Rust = High level general purpose language

- Multi-paradigm, ideal for functional, imperative and even OOP
- Modern tooling
- Ideal for systems programming, embedded, web servers and more
- Memory safe
- Concurrent
- No garbage collection
- Performance focused
- Most loved language in Stack Overflow survey for 6 years in a row

“Rust throws around some buzz words in its docs, but they are not just marketing buzz, they actually mean it with full sincerity and they actually matter a lot”

Safety guarantee

- Memory safe
- Null safe
- Type safe
- Thread safe

Rust is safe by default and you can write unsafe code only within `unsafe` code blocks

Memory safety

- Memory safety ensured at compile time using the ownership mechanism
- Borrow checker built into the compiler
- Unsafe mode for manual memory management and memory unsafe code
- There is no concept of null at the language level. Instead, Rust provides Option monad

Ownership and borrowing

- No garbage collection or any runtime memory management
- Memory is managed using lifetimes of variables using a borrow checker at compile time
- No pause times, no runtime overhead
- Efficient and very low memory usage
- Reference counting available when needed

Type safety

"Most modern strictly typed languages guarantees this"

- No need for reflection
- Memory safety assures type safety as well
- Strict compile time type checks
- Dynamic typing is possible with `dyn` and `Any` but compiler is smart enough to ensure type safety for those

Thread safety

"Fearless concurrency"

- Threads, coroutines and asynchronous concurrency
- Mutex and ARC for shared data concurrency
- Channels for message passing concurrency
- Data race is not possible in Rust
- No need for thread synchronization
- Memory and Type safety ensures thread safety

Zero cost abstractions

“What you don’t use, you don’t pay for. And further: What you do use, you couldn’t hand code any better.”
– Bjarne Stroustrup

- Your programming style or paradigm does not affect performance
- Number of abstractions does not affect performance as the compiler always translates your code to the best possible machine code
- You could not get more performance from hand written optimizations
- In-built abstractions are often more performant than hand written code
- Compiler produces identical assembly code for almost all variations

Zero cost abstractions

```java
// Average  10.059 ns/op
public long factorialForLoop(long number) {
    long result = 1;
    for (; number > 0; number--) {
        result *= number;
    }
    return result;
}

// Average  20.689  ns/op
public long factorialRecursive(long number) {
    return number == 1 ? 1 : number * factorialRecursive(number - 1);
}

// Average  23.457 ns/op
public long factorialStream(long number) {
    return LongStream.rangeClosed(1, number)
            .reduce(1, (n1, n2) -> n1 * n2);
}

/*
# Run complete. Total time: 00:02:30 (JDK 11)

Benchmark                    Mode  Cnt   Score    Error  Units
MyBenchMark.forLoop          avgt    3  10.059 ±  1.229  ns/op
MyBenchMark.recursive        avgt    3  20.689 ±  4.465  ns/op
MyBenchMark.stream           avgt    3  23.457 ± 32.424  ns/op
*/
```

```rust
// Average  8.5858 ns/op
fn factorial_loop(mut num: usize) -> usize {
    let mut result = 1;
    while num > 0 {
        result *= num;
        num = num - 1;
    }
    return result;
}

// Average  8.6150 ns/op
fn factorial_recursion(num: usize) -> usize {
    return match num {
        0 => 1,
        _ => num * factorial_recursion(num - 1),
    };
}

// Average 6.6387 ns/op
fn factorial_iterator(num: usize) -> usize {
    (1..num).fold(1, |n1, n2| n1 * n2)
}

/*
Benchmark              time:   [min       avg       max      ]
factorial_loop         time:   [8.4579 ns 8.5732 ns 8.7105 ns]
factorial_recursion    time:   [8.4394 ns 8.5074 ns 8.5829 ns]
factorial_iterator     time:   [6.4240 ns 6.4742 ns 6.5338 ns]
*/
```

Immutable by default

- Variable are immutable by default, including references
- Mutations needs to explicitly declared at all stages using the `mut` keyword, like var declaration and method signatures
- Variables can be passed by value or reference, mutable or immutable

Pattern matching

- First class support
- Can be used for control flow in `if`, `switch`, `while`, `for` statements
- Can be used for error handling, optionals and so on
- Can be used for value assignments and for code blocks

Advanced generics, traits and types

- Advanced generics
  - Generics in types, structs, enums and functions
  - No performance impact due to zero cost abstractions
  - Generics with lifetime annotations
- Traits for shared behaviour
  - Default implementation for traits
  - Placeholders for traits, operator overloading
  - Trait bounds for Generics
  - Multiple and compound trait bounds
- Type aliasing and great type inference

Macros

- Meta programming
- Great for non generic reusable code
- Custom behaviours
- Declarative macros and Procedural macros

Tooling and compiler

- Hands down, one of the best compilers out there
- One of the best tooling you can find in terms of features and developer experience
    - Cargo is one stop shop for Rust tooling, build, compilation, formatting, linting, and so on
- One of the best documentation, which is shipped with the tooling

Community and ecosystem

- A very diverse, welcoming and vibrant community
    - Community formed from other languages hence bringing in best of many
- Rapidly maturing ecosystem
    - Growing number of libraries and use cases
    - Has a forum which is used more than stack overflow for Rust
- Great backward compatibility
- Big names like Google, Apple, Microsoft, Amazon and Facebook are already behind rust and investing it.
- It’s on path to become the second supported language in Linux development.
- Use case has already extended to embedded, web assembly, kubernetes, web development, game development and even client side
    - It’s only a matter of time until you can do any use case in Rust

Does that mean there is no downsides?

The downsides

- Complexity
- Steep learning curve
- Young and maturing
- Many ways to do the same thing (kind of like JS)

Rust can be the ideal general purpose language

High level vs Low level language

High level language

- Human oriented
- Easier to read
- Portable
- Need to be compiled to machine code
- Not as efficient as a low level language
- Provides features like memory management, abstractions and so on

Low level language

- Machine oriented
- Harder to read
- Hardware specific
- Can be understood by machines
- Fast and efficient
- No fancy features

Performance, Memory and power

From the research paper [“Energy Efficiency across Programming Languages”]

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

High level language compromise

- Safety
- Speed
- Abstractions

Pick two

High level language compromise

- Safety
- Speed
- Abstractions

With Rust we can get all three. Hence Rust is a high level language with performance and memory efficiency closest to a low level language. The only tradeoff you will make with Rust is the learning curve.

“Rust, not Firefox, is Mozilla’s greatest industry contribution”

– TechRepublic

Thank You

Deepu K Sasidharan

@deepu105 | deepu.tech

https://deepu.tech/tags#rust

okta