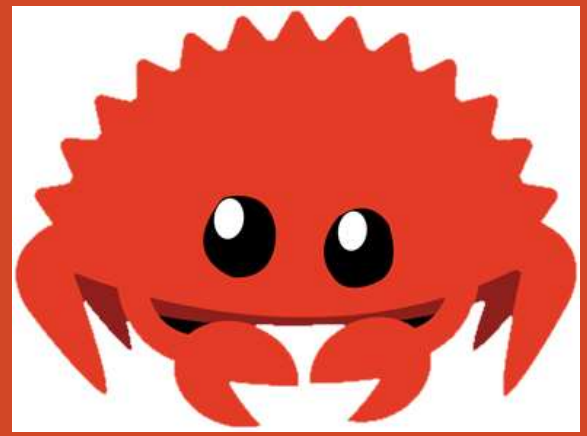
Rust, a C/C++ Replacement?

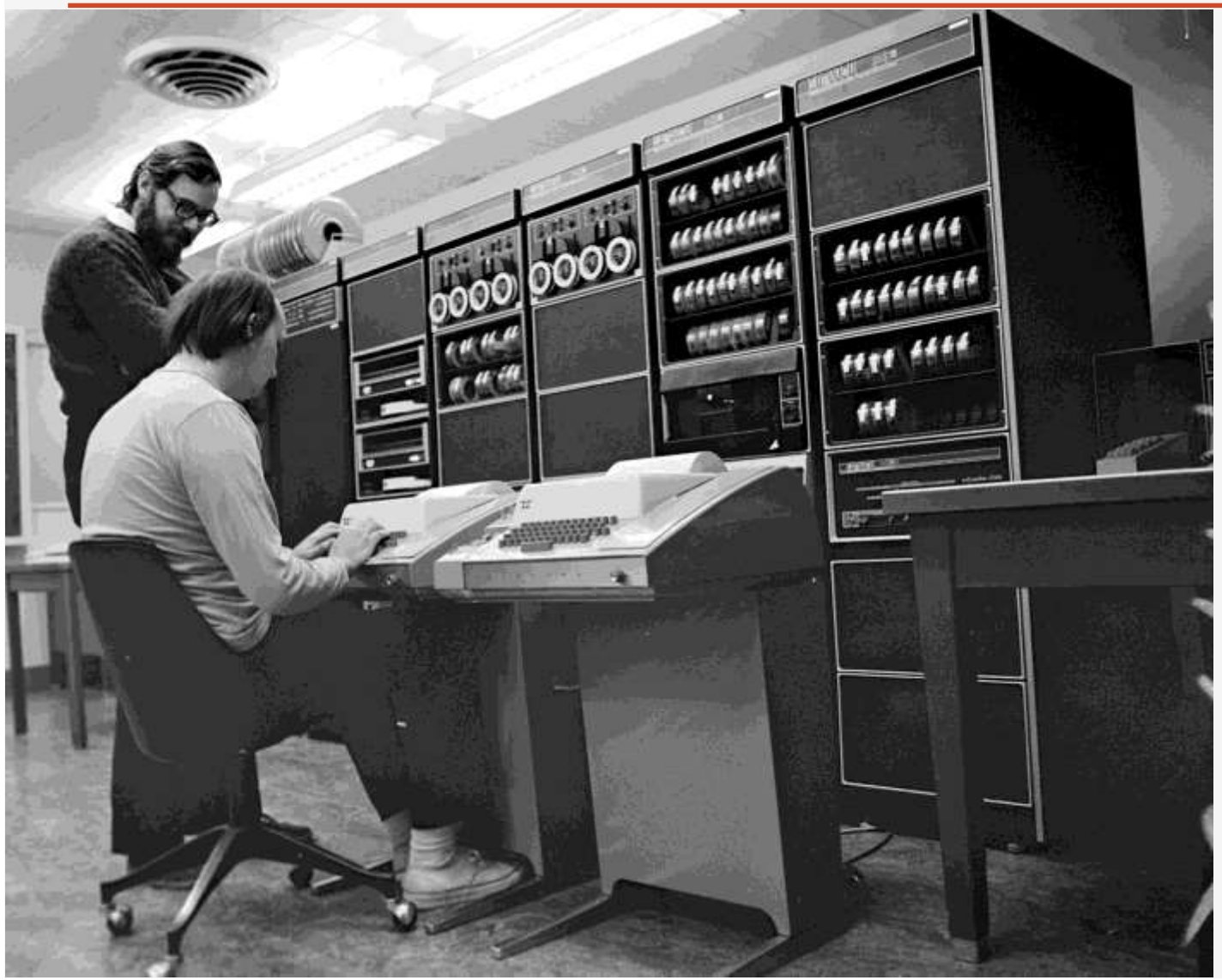


James R. Small, Principal Architect

Michigan!/usr/group

mug.org – A Free and Open Source Michigan Community

Dennis Ritchie and Ken Thompson – Inventors of C and UNIX



- C evolved from the B Programming Language (1971)
- First graphics terminals (1972)
- First Intel x86 CPU (8080 – 1974)
- Migration of Internet to TCP/IP (1983)
- Unicode (modern language/ symbol representation – 1987)
- First widespread security incident (Morris worm – 1988)

Image Source: Wikipedia

Bjarne Stroustrup – Inventor of C++

- C with classes (later renamed C++) evolved from C (1979)
- Stroustrup wanted the Simula programming language class features and C's efficiency
- A key decision was to build on C (maintain compatibility)
  - A big part of C++'s success
  - C essentially portable assembler with near ubiquitous platform support
  - Unfortunately also major weakness for C++, shackled with C's weaknesses

Image Source: Wikipedia

What This Talk Is Not

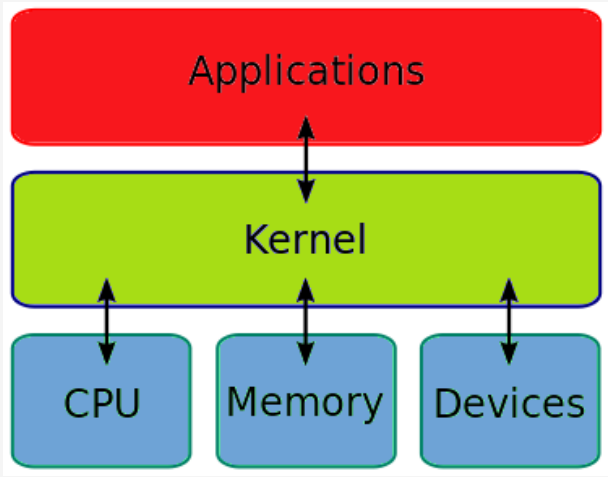
- Not about disparaging C
  - To me, Dennis Ritchie is an inspiration
  - He took what he had and made it better
- Not about denigrating C++
  - Dr. Stroustrup combined some of the best languages at the time
  - Much of want he wanted to do wasn’t possible at the time

Using The Right Tool…

- When you need a Systems Programming Language – what is the best choice?
- No language is perfect – which tradeoffs you want to make?
- I hope to convince you that Rust is a strong contender for your time
- Using Rust will make you a better programmer – even if you primarily use another language!

Image Source: x

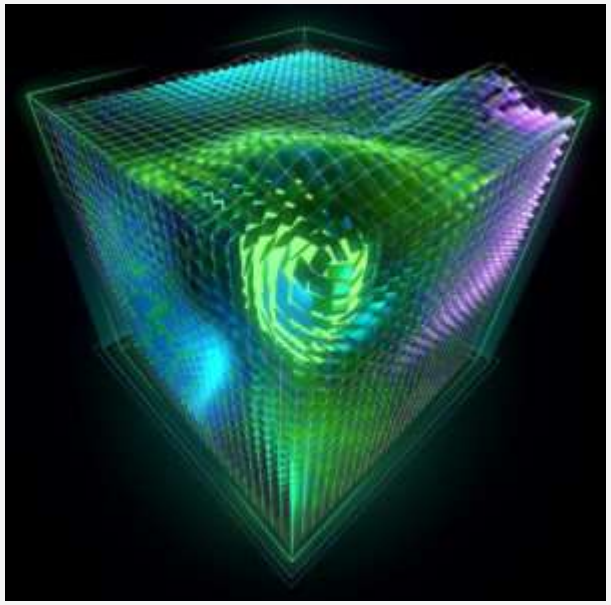
Why C?



- Proven – Used everywhere for nearly 50 years
- Performant – Fast, Efficient, Minimal runtime
- Portable – often referred to as “portable assembler”
- Ubiquitous – compilers for virtually any platform
- Simplicity – relatively small language
- Existing Code Base – easiest interface option
- Stable Specification - standardized, little change

Image Sources: https://www.online-sciences.com/programming/c-programming-language-features-advantages-and-disadvantages/, Wikipedia, https://www.udemy.com/course/embedded-c-programming-for-embedded-systems/

Why C++?



- Proven – Extensive use for over 40 years
- Performant, Portable, Ubiquitous – C-like
- Existing Code Base – easiest interface option
- Better Abstractions – classes and OOP
- Generics – data structures/algorithms (STL)
- Functional features – template metaprogramming, lambdas
- Zero-Overhead Principle – don't pay for what you don't use

Image Sources: Wikipedia, https://devblogs.nvidia.com/separate-compilation-linking-cuda-device-code/

What's missing from C/C++ versus Modern Programming Languages?

- Ecosystem Problems:
  - No standard build system
  - No standard dependency management system
  - No standard packaging format or system
  - No standard/centralized package registry
  - No reference compiler implementation
    - Standard language definition with various implementations – MSVC, GCC, Clang, …

Image Sources: The Blazing Center, Inhabitat

Issues with C/C++ versus Modern Programming Languages?

- Language Problems:
  - Undefined Behavior everywhere – creating robust and secure programs extremely difficult
  - C has no native concept of a String
  - C++ has a String, but backwards compatible with C
    - Essentially an array of 8-bit integers designed for ASCII
    - Weakly typed, endless source of security issues
    - Doesn't natively support Unicode
    - Null terminated (i.e., uses in-band ASCII character)*

Image Sources: Quick Country 96.5, Fiore Group Training

Issues with C/C++ versus Modern Programming Languages?

- Language Problems (continued):
  - Built-in arrays problematic
    - Decay to pointer when passed
    - Don’t know their own size
    - Don’t really support multi-dimensioning
  - C++ Templates also problematic
    - Didn’t include early type checking in their design (per Stroustrup this wasn’t possible at the time)
  - Macros are a separate system which don’t understand the language

Image Sources: Quick Country 96.5, Fiore Group Training

Why not C/C++?

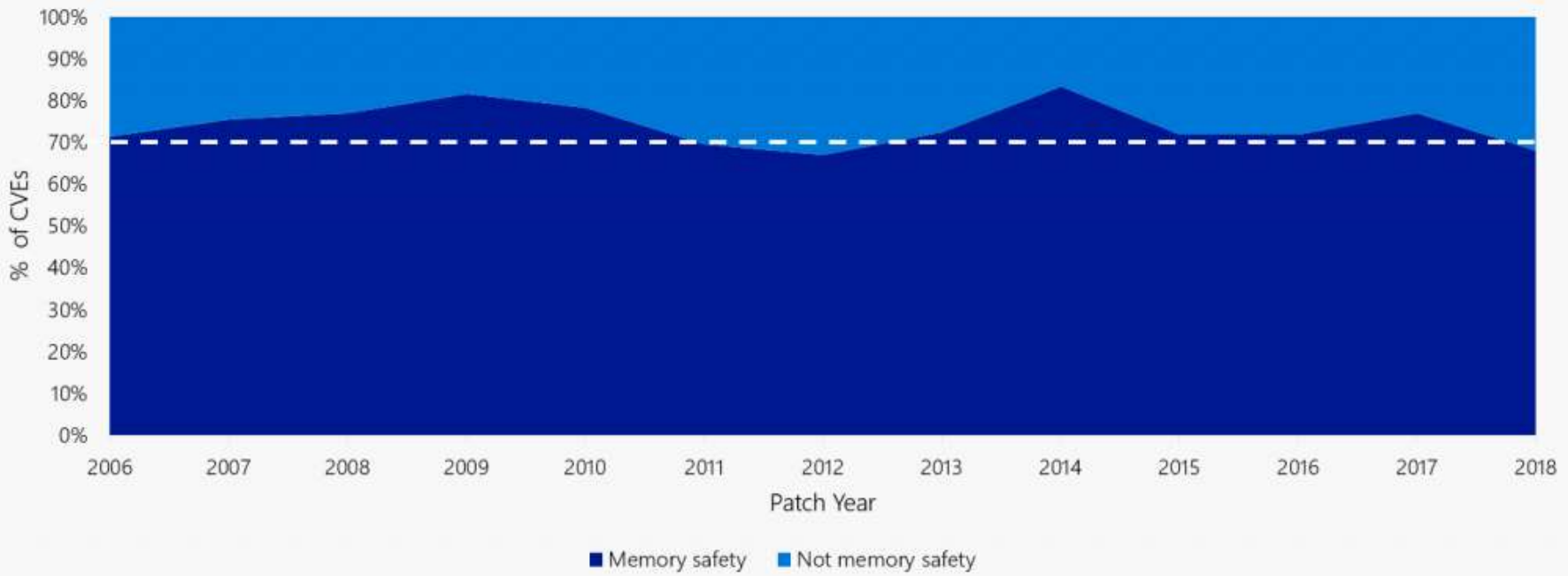
- Lack of Memory Safety
  - Use after free (dangling pointer access)
  - Double free/Invalid free
  - Use of uninitialized memory
  - Buffer overflow* (overreads/overwrites)
  - Out-of-bounds access (access past end of array)
  - Null pointer access (Dereferencing)
  - Memory leaks (stack/heap exhaustion)
  - Data races (concurrent mutable access)

Image Sources: safetysignsupplies.co.uk, https://www.fool.com/retirement/general/2015/01/13/the-time-bomb-lurking-in-your-social-security.aspx

Why does memory safety matter?

"The majority of vulnerabilities fixed and with a CVE assigned are caused by developers inadvertently inserting memory corruption bugs into their C and C++ code."

"As Microsoft increases its code base and uses more Open Source Software in its code, this problem isn't getting better, it's getting worse."



- Gavin Thomas, principal security engineering manager at the Microsoft Security Response Center (MSRC)

Image Source: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/

Thoughts on turning Systems Programmers into Security Experts

- Listening to security experts – things have not improved, the same mistakes are repeatedly made
- Perhaps a better solution is security by default

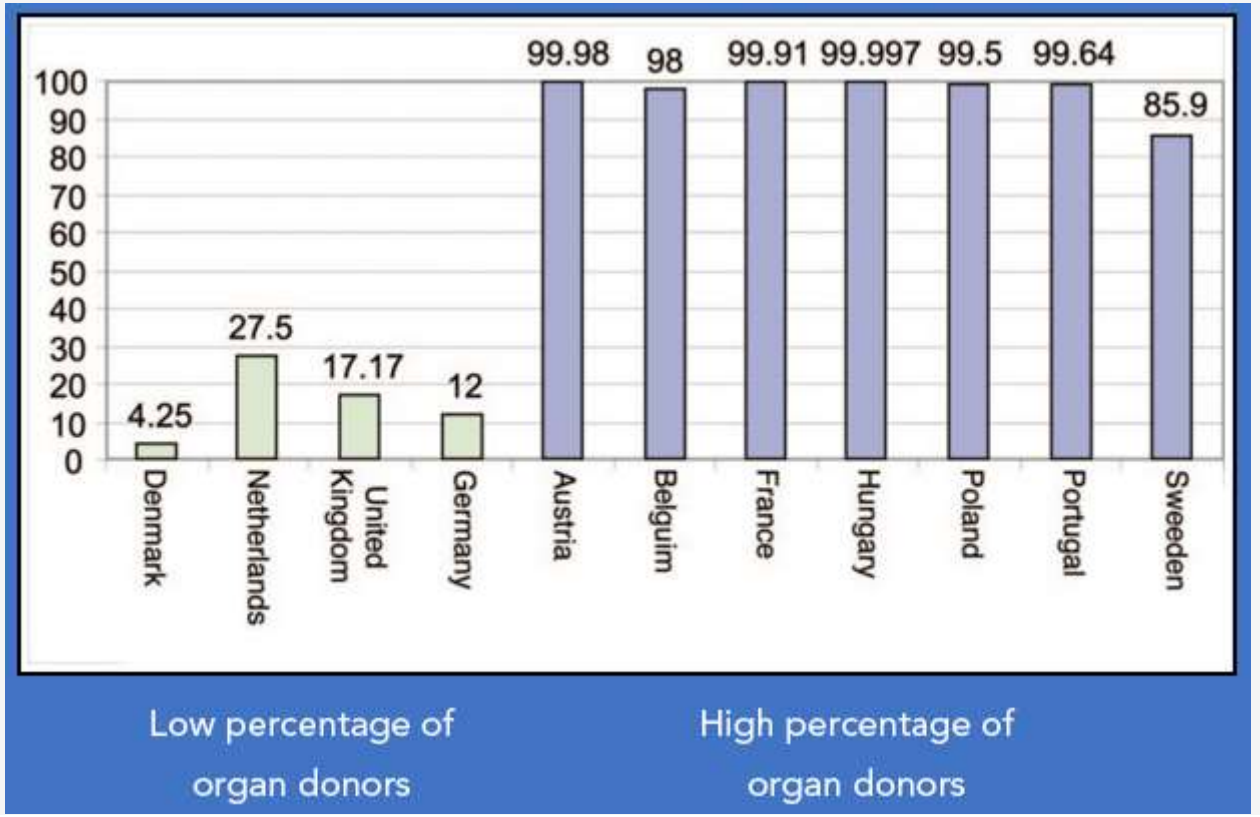
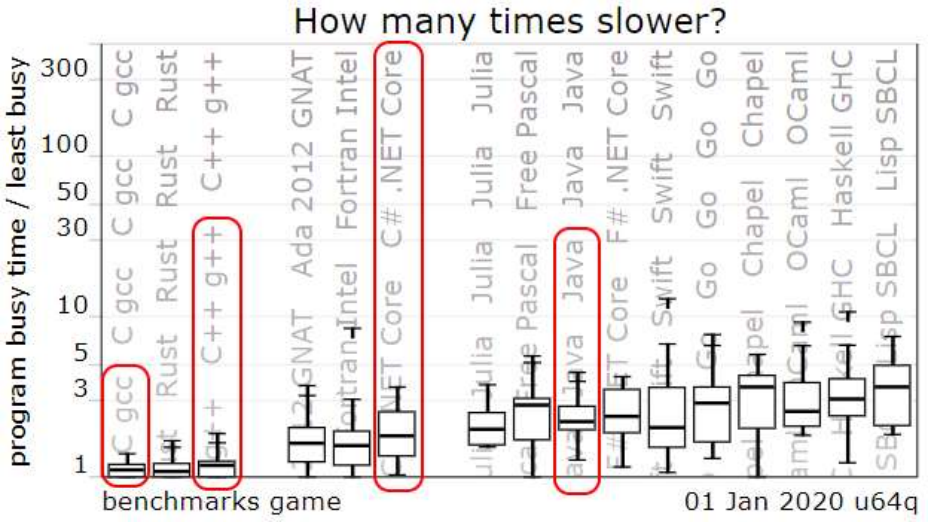


Image Source: https://jamesclear.com/environment-design-organ-donation

Just Use Memory Safe Languages?

The Computer Language Benchmarks Game



- Languages like C# and Java can work well in many contexts, but not all.
- Microsoft tried to rewrite the Windows shell, making extensive use of C# and Windows Presentation Foundation, for the first attempt at Windows Vista.
  - The result was impossibly slow and the company made a famous "reset" back towards native code in 2004.
- As shown to the left, C# and Java tend to be about twice as slow compared to C/C++

Image Source: https://benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html

Thoughts on C++

"C++ is a great language as long as you don't make any mistakes – no pressure!"

"Rather than investing in more and more tools and training and vulnerability fixes, what about a development language where they can't introduce memory safety issues into their feature work in the first place?" - Microsoft Security Response Center

"If only developers could have all the memory security guarantees of languages like .NET C# combined with all the efficiencies of C++. Maybe we can with Rust." - Microsoft Security Response Center

Image Source: https://luckybjj.com/should-i-lose-weight-before-starting-jiu-jitsu/person-thinking-png-hd-thinking-man-png-png-person-thinking-1169/

Developing in C/C++

- Standard joke about C/C++ from programmers in other languages:
  - Segmentation fault (core dumped) – good luck
- When a program crashes:
  - No stack trace
  - No guarantee of memory integrity
  - Often you have little idea where the problem is
- Assumption is you have excellent tooling and superb debugging skills

Image Source: https://www.ministrymatters.com/all/entry/4068/7-of-the-most-frustrating-things-pastors-experience

Assumptions about Performance Critical Code

- Must be native (compiled into machine code)
- Cannot use a Garbage Collector (GC) – results in non-deterministic performance when GC runs
- Must use manual memory management – consequence of no GC
- Rules out most programming languages besides C/C++
  - There are languages like D, but they have not become mainstream

Image Source: ul.com

Current Programming Language Landscape

Language	TIOBE	PYPL	RedMonk	StackOverflow	SlashData	IEEE
C	2	6	9	11 {D-4}	5	3
C++	4	6	5	10 {D-9}	5	4
Java	1	2	2	5 {D-10}	3	2
PHP	8	5	4	8 {D-5}	6	13
JavaScript	7	3	1	1 {-*}	1	6
C#	5	4	5	7 {L-10}	4	7
Python	3	1	3	4 {L-2}	2	1
Swift	9	9	11	14 {L-6}	8	9
Rust	30	18	21	21 {L-1}	-	17
D	17	-	Low	-	-	42
Ada	38	23	-	-	-	43

Current Programming Language Landscape

Systems Programming Languages	Comments
Ada	Fading away
C	Dominant embedded language
C++	Along with C, dominates systems category
D	Niche language
Rust	Mozilla's up and coming systems language – the most loved language 4 years running (StackOverflow)
Swift	Apple's systems language – gaining in popularity (replacing Objective-C)

Graydon Hoare – Inventor of Rust

- Started as side project in 2006
  - Frustrated with C++ complexity, lack of safety
  - Wanted C-like performance/efficiency with safety and strong concurrency support
- Mozilla picked up in 2009, announced in 2010
- 1.0 released in May 15, 2015 with stability commitment (currently 1.40)
- 6 week cadence with 3 channels – nightly, beta, stable
  - Major changes done through Editions (2015, 2018, …)

Image Sources: Mozilla, https://adainitiative.org/2012/10/13/graydon-hoare-i-donated-because-id-like-to-see-the-culture-change/, Lars Bergstrom

Enter Rust

- Native binaries
- Automatic memory management without GC
- Memory safety
- Fearless concurrency
- C-like performance/efficiency
- Much easier to troubleshoot
  - Friendly compiler diagnostics
  - Stack traces on by default
  - No Undefined Behavior!
  - No Segmentation Faults!

Image Source: Celebrations Magazine

C, C++, Rust – A Comparison

C, vintage 1971

C++, vintage 1979

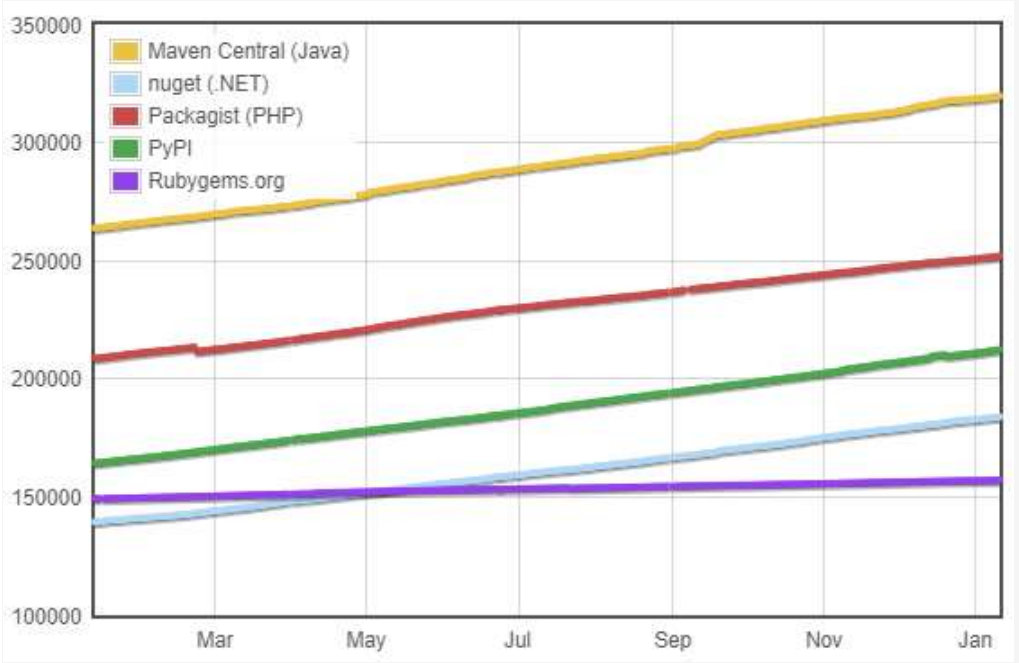
Rust, vintage 2006

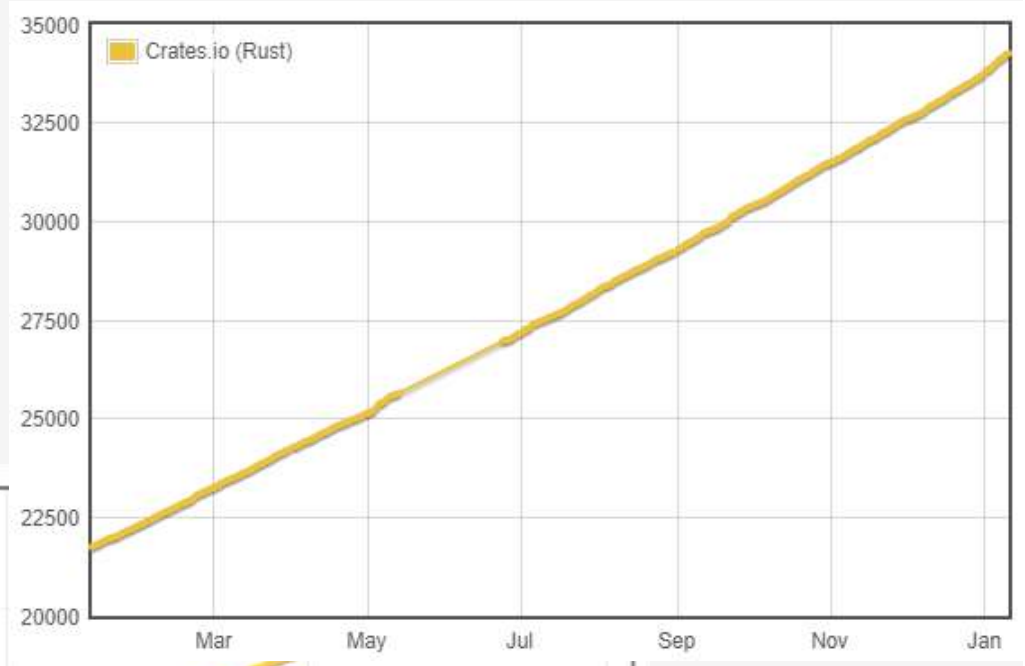
Image Sources: Speed Hunters, Beverly Hills Car Club, Free Classifieds

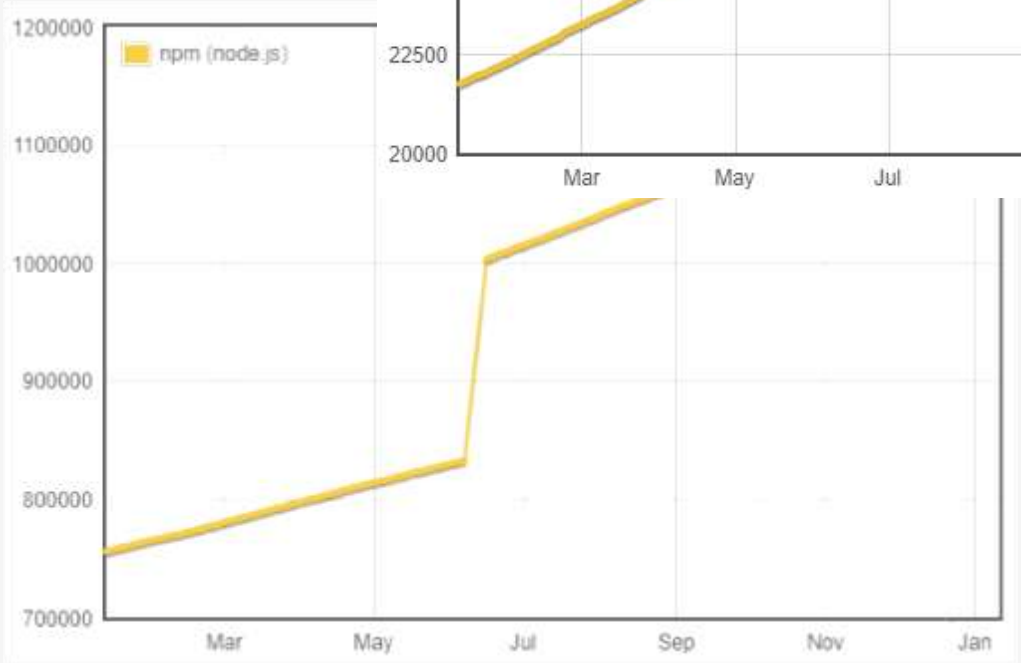
C/C++ versus Rust – Ecosystem Comparison

	C/C++	Rust
Build System	No standard – CMake, Visual Studio Project, Makefiles, …	Cargo
Dependency System	No standard – Use Build/Packaging System	Cargo
Packaging Format	No standard – Use Packaging System Format	Crate
Packaging System	No standard – Buckaroo, Conan, vcpkg, …	Cargo
Package Registry	No standard, all have < 1,000 packages: Use Packaging System's	crates.io, > 34k packages
Reference Compiler	No standard – Microsoft/Windows – MSVC, Linux – GCC, Apple/Facebook/Google – Clang	Yes

Programming Language Packaging System Comparison (Last Year)







- The established mainstream languages have 100s of thousands of packages
- JavaScript (Node) has > 1 million
- Rust has solid growth
- C/C++ don't even have one sizeable package registry

Image Source: http://www.modulecounts.com/

C/C++ versus Rust – Language Comparison

	C/C++	Rust
Undefined Behavior	UB Everywhere, "… despite … excellent advances in tooling … UB-related problems are far from solved…"*	No UB in safe Rust
Strings	Poor Unicode support, weak typing, security problems, use of in-band terminator	Solid, but some complexity w/ String v. str
Arrays	Decay to pointer, don't know size, weak dimensioning	Arrays don't decay, know size, same dim issue though
Generics	Type enforcement complex using type traits, SFINAE, and other techniques on templates	Type enforcement easy through traits
Macros	Don't understand C/C++	Understand Rust, hygienic

What is the Spectrum of Language Type Systems?

Language	Different Types	Variables Bound to Type	Function/Method Parameters Typed and Enforced	Type Enforcement	Implicit Conversions?	Type Safe?	Memory Safe?
Bash/Tcl[1]	No[2]	No	No	No	N/A	N/A	Yes
Python	Yes	No[3]	Optional[4,5]	Yes	Some	Yes	Yes
C	Yes	Yes	Optional/Weak[6]	Weak[6]	Yes (Unsafe)	Weak[6]	No
C++	Yes	Yes	Weak[6]	Weak[6]	Yes (Unsafe)	Weak[6]	No
Rust	Yes	Yes[7]	Yes	Yes[8]	No	Yes[8]	Yes[8]
ML/Haskell	Yes	Yes[7]	Yes	Yes	No	Yes	Yes

Intro to Rust

- Systems programming language
  - Includes high-level scripting like features
  - Designed for everyone
  - Inclusive, warm, and friendly community – beginners welcome
  - Ergonomics at the center of the language

Rust – Hello World

- Install Rust: Retrieve and run rustup-init from https://rustup.rs
  - For Windows, also need Build Tools for Visual Studio
- Using cargo
  - cargo new hello
  - Use Visual Studio Code to inspect (has excellent plugin support for Rust)
  - cargo run from hello directory (looks for Cargo.toml)
  - Look at what cargo run created (Cargo.lock, target directory)
  - Documentation: rustup doc

Rust Variable Types – Numerics

Length	Signed	Unsigned	Floating-Point	Examples
8-bit	i8	u8		`Decimal: 101_439`
16-bit	i16	u16		`Hex: 0xff`
32-bit	i32	u32	f32	`Octal: 0o77`
64-bit	i64	u64	f64	`Binary: 0b1111_0000`
128-bit	i128	u128		`Byte*: b'A'`
Arch Native	isize (32 or 64)	usize (32 or 64)		*(u8 only)

Rust Variable Types – Additional Scalars

Type	Keyword	Values	Notes
Boolean	bool	true | false	
Character	char	‘c’, ‘Ł’, ‘β’, ‘♬’, ‘€’, ‘ﬓ’, ‘🏀’, ‘✔’, ‘😎’	4 bytes must use single quotes Unicode Scalar: U+0000 – U+D7FF, U+E000 – U+10FFF

Rust Variables – Compound Types

Type	Example	Notes
tuple	(1, ‘a’, true)	Size fixed at creation Supports any type
array	[1, 2, 3]	Size fixed at creation All elements must be same type
slice	&*variable*[#..#]	Non-owning (more later) Elements from *variable* in specified range

Rust – Some Popular Collections from the Standard Library

Type	Keyword	Example	Notes
String	String	String::from("Hello")	UTF-8 encoding
String slice	str	"Goodbye"	Often borrowed: &str (more on this soon)
Vector	Vec<T>	vec![1, 2, 3]	Often use vec! macro to create
Hash Map	HashMap<K, V>	Coming…	

Rust – Using Variables

```
// Typical:
let variable = value;

// Optionally specify type
let variable: type = value;

// Allow mutation:
let mut variable = value;
```

Rust – Using Variables

- Immutable by default

```rust
// Default entry point for programs:
fn main() {
    // Declare a variable:
    let number = 41;

    // Wait - isn't that 42?
    // But variables immutable by default - error:
    number += 1;

    // Use println macro to display
    // Anything ending with "!" is a macro
    println!("The answer is {}", number);
}
```

```
C:\apps\working\rust\code\myproject [master +4 ~0 -0 !]> cargo run
   Compiling myproject v0.1.0 (C:\apps\working\rust\code\myproject)
error[E0384]: cannot assign twice to immutable variable `number`
 --> src\main.rs:8:5
  |
4 |     let number = 41;
  |         ------
  |         |
  |         first assignment to `number`
  |         help: make this binding mutable: `mut number`
...
8 |     number += 1;
  |     ^^^^^^^^^^^ cannot assign twice to immutable variable

error: aborting due to previous error

For more information about this error, try `rustc --explain E0384`.
error: could not compile `myproject`.

To learn more, run the command again with --verbose.
```

Rust – Using Variables

- Mutability is opt-in

```rust
// Default entry point for programs:
fn main() {
    // Declare a mutable variable:
    let mut number = 41;

    // Wait - isn't that 42?
    number += 1;

    // Use println macro to display
    // Anything ending with "!" is a macro
    println!("The answer is {}", number);
}
```

- Type Inference – Thanks ML!
  - Can optionally specify

```rust
// Default entry point for programs:
fn main() {
    // Declare a mutable variable:
    // Optionally specify a type, <variable>: <type>
    let mut number: i32 = 41;

    // Wait - isn't that 42?
    number += 1;

    // Use println macro to display
    // Anything ending with "!" is a macro
    println!("The answer is {}", number);
}
```

Rust – Using Variables

- The compiler is strict
  - Complains about unused variables!

```
C:\apps\working\rust\code\myproject [master +4 ~0 -0 !]> cargo run
   Compiling myproject v0.1.0 (C:\apps\working\rust\code\myproject)
warning: unused variable: `item1`
  --> src\main.rs:20:10
   |
20 |     let (item1, item2, item3, item4) = row;
   |          ^^^^^ help: consider prefixing with an underscore: `_item1`
   |
   = note: `#[warn(unused_variables)]` on by default

warning: unused variable: `item2`
  --> src\main.rs:20:17
   |
20 |     let (item1, item2, item3, item4) = row;
   |                 ^^^^^ help: consider prefixing with an underscore: `_item2`

warning: unused variable: `item3`
  --> src\main.rs:20:24
   |
20 |     let (item1, item2, item3, item4) = row;
   |                        ^^^^^ help: consider prefixing with an underscore: `_item3`

warning: unused variable: `item4`
  --> src\main.rs:20:31
   |
20 |     let (item1, item2, item3, item4) = row;
   |                               ^^^^^ help: consider prefixing with an underscore: `_item4`

    Finished dev [unoptimized + debuginfo] target(s) in 1.31s
     Running `target\debug\myproject.exe`
number:  1, conditional:  true, failed:  false
prime_numbers:  [2, 3, 5, 7, 11], second_prime:  3, some_primes:  [3, 5, 7]
row:  (1, 'a', "apple", true), row_item:  apple, my_floats:  [1.1, 2.2, 3.3]
```

```rust
// Default entry point for programs:
fn main() {
    // Declare a bunch of variables
    let number = 1;
    let conditional = true;
    let failed = false;

    // Array:
    let prime_numbers = [2, 3, 5, 7, 11];
    // Index into array:
    let second_prime = prime_numbers[1];
    // Slice - last element not included:
    let some_primes = &prime_numbers[1..4];  // 3, 5, 7

    // Tuple:
    let row = (1, 'a', "apple", true);
    // Index into tuple:
    let row_item = row.2;  // "apple"
    // Assignment from tuple via "destructuring":
    let (item1, item2, item3, item4) = row;

    // Vector of floating-points:
    let my_floats = vec![1.1, 2.2, 3.3];

    // Display:
    println!("number:  {}, conditional:  {}, failed:  {}",
             number, conditional, failed);
    println!("prime_numbers:  {:?}, second_prime:  {}, some_primes:  {:?}",
             prime_numbers, second_prime, some_primes);
    println!("row:  {:?}, row_item:  {}, my_floats:  {:?}",
             row, row_item, my_floats);
}
```

Rust – Using Functions

```
// Typical:
fn function1(param1: type, param2: type) -> return_type {
    // body...
}

// Without parameters or return value:
fn function2() {
    ...
}
```

Rust – Using Functions

- Many Rust keywords follow the UNIX naming convention – abridged and vowelless
  - function = fn (“fun”)
- Unlike variable names, function parameters must have their type declared
- Return type (if used) must also be declared
- Unlike C/C++, functions don’t have to be declared/defined before use within the module (file)

```rust
fn main() {
    let number = 11;
    // String literals are string slices (str) in Rust:
    let name = "George";

    // Call functions before declaring/defining:
    println!("{} squared:  {}", number, square(number));
    multigreet(name, 5);
}

// Don't need to declare/define functions before use!
// Function parameters:  <variable_name>: <type>, ...
// Function return type:  -> <type> (optional)
fn square(number: i32) -> i32 {
    // Rust does have a return statement
    // However, the idiomatic way to return a value is to
    // omit the terminating semi-colon on the last line
    number * number
}

// &str is a reference to a string slice - more shortly:
fn multigreet(name: &str, count: i32) {
    // Use "_" for index variable since we don't use it:
    for _ in 0..count {
        println!("Greetings {}!", name);
    }
}
```

Rust – Control Flow

```
// Don't need parenthesis
// around condition:
if condition {
    ...
} else if condition {
    ...
} else {
    ...
}
```

```
// Loop forever
loop {
    ...
    // Optional exit:
    if condition {
        break;
    }
}
```

```
// Don't need parenthesis:
while condition {
    ...
    if condition {
        // start over
        continue;
    } else {
        // do stuff...
    }
}
```

Rust – Control Flow

```
// Iterate over collection
for item in collection {
    ...
}
```

```
// if-else/switch/case improved:
match variable {
    // Specific cases:
    value1 => // do stuff,
    value2 => // do stuff,
    // Optional wildcard case:
    _ => // do stuff
}
```

Rust – Control Flow

- Note – unlike C/C++, can't omit {}:
  - The source of many errors…
  - Probably for the best.

```rust
fn main() {
    let number = 55;

    if number > 100
        println!("Nice!");
}
```

```
C:\apps\working\rust\code\myproject [master +4 ~0 -0 !]> cargo run
   Compiling myproject v0.1.0 (C:\apps\working\rust\code\myproject)
error: expected `{`, found `println`
 --> src\main.rs:5:9
  |
4 |     if number > 100
  |     -- this `if` statement has a condition, but no block
5 |         println!("Nice!");
  |         ^^^^^^^-----------
  |         |
  |         expected `{`
  |         help: try placing this code inside a block: `{ println!("Nice!"); }`

error: aborting due to previous error

error: could not compile `myproject`.

To learn more, run the command again with --verbose.
```

Rust – Control Flow

- Examples:

```rust
// Bring these libraries in scope:
use std::io;

fn main() {
    let mph = 55;

    if mph > 100 {
        println!("That's pretty fast!");
    } else if mph < 10 {
        println!("That's pretty slow.");
    } else {
        println!("That's a cruise on the highway.");
    }

    let mut counter = 1;
    while counter <= 10 {
        println!("I'm chatty!");
        counter += 1;
    }
```

```rust
    let mut guess = String::new();
    // Loop forever!
    loop {
        println!("What's the best programming language?");
        // Read a line - must pass variable mutably!
        io::stdin().read_line(&mut guess)
            // read_line returns a Result which could be either
            // the value or an error.  Expect handles the
            // error case:
            .expect("Failed to read line of input.");

        // Strip off whitespace:
        guess = guess.trim().to_string();

        // String literals are slices (references)
        // To compare a String to a str (slice) we need to
        // convert the String to a reference:
        match guess.as_ref() {
            "Rust" => {
                println!("That's right!");
                break;
            },
            _ => {
                println!("Hmmm...I think you need to try again.");
                // This seems to accumulate rather than overwrite,
                // so reset string:
                guess = "".to_string();
            }
        }
    }
}
```

Rust – Control Flow

- More Examples:
  - For loop with iterators

```rust
fn main() {
    let numbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29];

    for number in numbers.iter() {
        print!("{}, ", number);
    }

    // Cheasy - use backspaces and space to overwrite
    // trailing comma...
    println!("\x08\x08 ")
}
```

- No ternary expression
  - Conditionally assign like this:

```rust
fn main() {
    let flag = true;

    // No ternary operator, instead:
    let number = if flag {
        42
    } else {
        41
    };

    println!("The answer is {}.", number);
}
```

Rust – Fizzbuzz

- Are you ready for an interview in Rust?

```rust
use std::io;

fn main() {
    let mut input = String::new();

    println!("Fizzbuzz up to:");
    io::stdin().read_line(&mut input)
        .expect("Failed to read line of input.");

    let number: u32 = input.trim().parse()
        .expect("Please enter a positive whole number!");

    fizzbuzz_to(number);
}

fn fizzbuzz_to(number: u32) {
    // 1..=# - Use a range that includes end number
    for i in 1..=number {
        fizzbuzz(i);
    }
}

fn fizzbuzz(number: u32) {
    if number % 15 == 0 {
        println!("{}:  fizzbuzz!", number);
    } else if number % 3 == 0 {
        println!("{}:  fizz", number);
    } else if number % 5 == 0 {
        println!("{}:  buzz", number);
    } else {
        println!("{}", number);
    }
}
```

Image Source: x

Rust – Enumerations

- A collection where one element can be valid
- Matches on enums must be exhaustive (cover all cases)!

```
C:\apps\working\rust\code\myproject [master +4 ~0 -0 !]> cargo run
   Compiling myproject v0.1.0 (C:\apps\working\rust\code\myproject)
error[E0004]: non-exhaustive patterns: `Goalie` not covered
  --> src\main.rs:17:11
   |
1  | / enum HockeyPosition {
2  | |     Center,
3  | |     Wing,
4  | |     Defense,
...  |
7  | |     Goalie,
   | |     ------ not covered
8  | | }
   | |_- `HockeyPosition` defined here
...
17 |       match position {
   |             ^^^^^^^^ pattern `Goalie` not covered
   |
   = help: ensure that all possible cases are being handled, possibly by adding wildcards or more match arms

error: aborting due to previous error
```

```rust
enum HockeyPosition {
    Center,
    Wing,
    Defense,
    // Can use final comma
    // or not...
    Goalie,
}

fn main() {
    let player1 = HockeyPosition::Wing;

    describe(player1);
}

fn describe(position: HockeyPosition) {
    match position {
        HockeyPosition::Center => println!("Does it all."),
        HockeyPosition::Wing => println!("Left or Right forward."),
        HockeyPosition::Defense => println!("Left or Right defense."),
        // HockeyPosition::Goalie => println!("Better be good!"),
    }
}
```

Rust – Structs

- General User-Defined Type

```rust
// Data only!
struct Point {
    name:  String,
    x: i16,
    y: i16,
}

fn main() {
    let point1 = Point {
        name: String::from("a"),
        x: 1,
        y: 1,
    };
    let point2 = Point {
        name: String::from("b"),
        x: 7,
        y: 5,
    };

    println!("The distance between point {} and point {} is {}.",
             point1.name, point2.name, distance(&point1, &point2));
}

fn distance(point1: &Point, point2: &Point) -> f64 {
    let x_diff = (point2.x - point1.x).pow(2);
    let y_diff = (point2.y - point1.y).pow(2);

    ((x_diff + y_diff) as f64).sqrt()
}
```

Image Source: x

Rust – Methods

```rust
struct Point {
    name:  String,
    x: i16,
    y: i16,
}

// Method defined within implementation block:
impl Point {
    // Method, first parameter always self:
    fn quadrant(&self) -> u8 {
        // Quadrants:
        // 1:  0 - 90 (x & y positive)
        // 2:  91 - 180 (x negative, y positive)
        // 3:  181 - 270 (x & y negative)
        // 4:  271 - 359 (x positive, y negative)
        if self.x >= 0 && self.y >= 0 {
            1
        } else if self.x < 0 && self.y >= 0 {
            2
        } else if self.x < 0 && self.y < 0 {
            3
        } else {
            4
        }
    }
}
```

```rust
fn main() {
    let point1 = Point {
        name: String::from("a"),
        x: 1,
        y: 1,
    };
    let point2 = Point {
        name: String::from("b"),
        x: -7,
        y: -5,
    };

    println!("Point {} is in quadrant {}.",
             point1.name, point1.quadrant());
    println!("Point {} is in quadrant {}.",
             point2.name, point2.quadrant());
}
```

Rust – Memory Management and Ownership

- Systems programming languages
  - No GC
  - Traditionally = manual memory management
  - And…not memory safe
- Other programming languages
  - Use GC
  - Automatic memory management
  - But non-deterministic performance when GC runs…

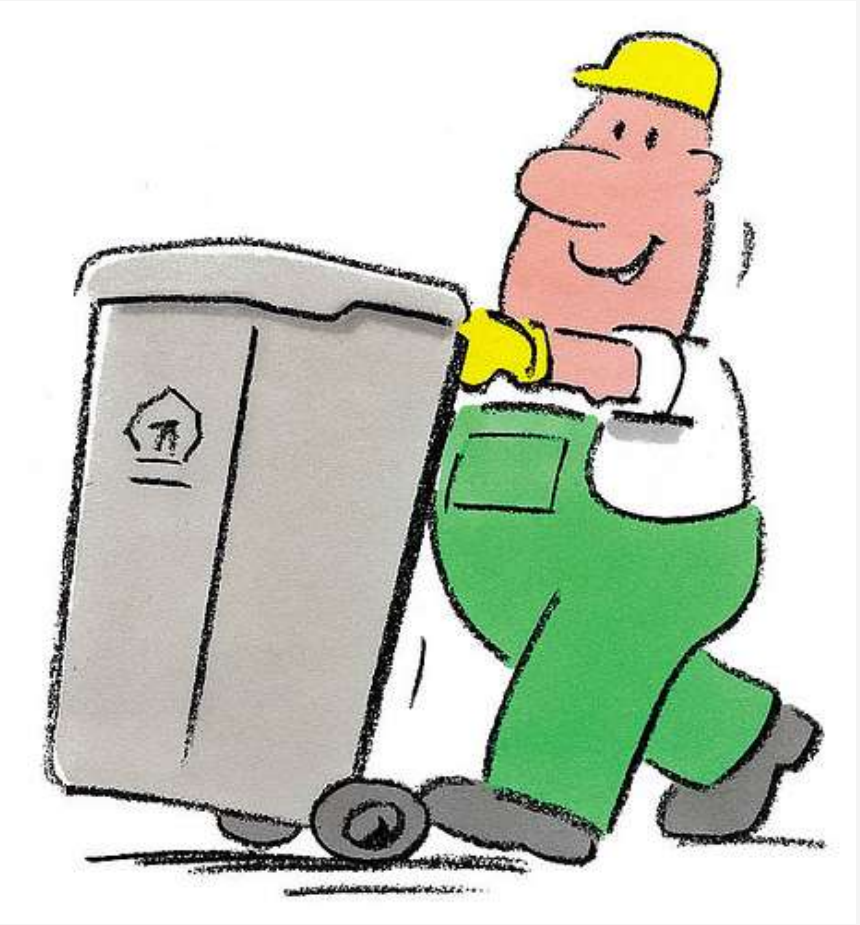


Image Source: Ivan Kuznetsov

Rust – Memory Management and Ownership

- Rust is a systems programming language
  - But automatic memory management
  - Provides memory safety
  - Implemented through ownership

```rust
fn main() {
    // Create new object example which owns allocated string data:
    let example = String::from("Hello!");

    println!("{}", example);
}   // Right before end of main, example de-allocated
```

Image Source: x

How Does Memory Safety Relate to Undefined Behavior (UB)?

- One result of compromising memory integrity
- What is UB? From the C FAQ:
  - Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.
  - In Internet parlance, "When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose."
- The current C Standard defines about 200 conditions which lead to UB!

Image Source: Wikipedia

UB Example – Use After Free 1 (C++)

Windows 10 (MSVC) – varies:

- Sometimes Access Violation
- Sometimes appears to run fine:

```
new_point:  15,-30
My point:  15,-30
```

Ubuntu 18 (GCC) – different values, no crash:

```
new_point:  15,-30
My point:  0,0
```

```cpp
#include <iostream>

struct Point {
    int x;
    int y;
};

Point* new_point() {
    // Heap allocated - OK to return new_point:
    Point* new_point = new Point;

    new_point -> x = 15;
    new_point -> y = -30;

    std::cout << "new_point:  " << new_point -> x << ','
              << new_point -> y << '\n';

    // Create another pointer
    Point* array_point = new_point;

    // Done - clean up:
    delete new_point;

    // Oops - returned pointer to freed memory!
    return array_point;
}

int main() {
    Point* my_point = new_point();
    // Oops - use after free, UB...
    std::cout << "My point:  " << my_point -> x << ','
              << my_point -> y << '\n';
}
```

UB Example – Use After Free 2 (C++)

Windows 10 (MSVC) – works:

```
new_point:  15,-30
My point:  15,-30
```

Ubuntu 18 (GCC) – different values, no crash:

- Values change each run:

```
new_point:  15,-30
My point:  15,32764
```

```cpp
#include <iostream>

struct Point {
    int x;
    int y;
};

Point* new_point() {
    // Stack allocated - cannot return new_point!
    Point new_point;
    Point* npptr = &new_point;

    npptr -> x = 15;
    npptr -> y = -30;

    std::cout << "new_point:  " << npptr -> x << ','
              << npptr -> y << '\n';

    // Oops - returned pointer to stack allocated (local) variable!
    return npptr;
}

int main() {
    Point* my_point = new_point();
    // Oops - use after free (dangling pointer), UB...
    std::cout << "My point:  " << my_point -> x << ','
              << my_point -> y << '\n';
}
```

UB Example – Double Free (C++)

Windows 10 (MSVC) – heap corruption:

- Values change each run:

```
new_point:  15,-30
My point:  19609608,19529920
```

Ubuntu 18 (GCC) – different values, no crash:

```
new_point:  15,-30
My point:  0,0
```

```cpp
#include <iostream>

struct Point {
    int x;
    int y;
};

Point* new_point() {
    Point* new_point = new Point;

    new_point -> x = 15;
    new_point -> y = -30;

    std::cout << "new_point:  " << new_point -> x << ','
              << new_point -> y << '\n';

    // Create another pointer
    Point* array_point = new_point;

    // Done - clean up:
    delete new_point;

    // Oops - returned pointer to freed memory!
    return array_point;
}

int main() {
    Point* my_point = new_point();
    // Oops - use after free, UB...
    std::cout << "My point:  " << my_point -> x << ','
              << my_point -> y << '\n';

    // Oops - double free!
    delete my_point;
}
```

UB Example – Uninitialized Variable Usage (C++)

Windows 10 (MSVC) – works:

- Values change each run:

```
My point:  20093154,6193152
```

Ubuntu 18 (GCC) – different values, no crash:

```
My point:  0,0
```

```cpp
#include <iostream>

struct Point {
    int x;
    int y;
};

int main() {
    // Declare new variable:
    Point my_point;

    // Oops - use of uninitialized vairable!
    std::cout << "My point:  " << my_point.x << ','
              << my_point.y << '\n';
}
```

UB Example – Buffer Overflow (C++)

Windows 10 (MSVC) – stack buffer overrun:

```
What's your name?
George_Longer_Than_Your_Buffer!
Hello, George_Longer_Than_Your_Buffer!!
```

Ubuntu 18 (GCC) – crash:

```
What's your name?
George_Longer_Than_Your_Buffer!
Hello, George_Longer_Than_Your_Buffer!!
*** stack smashing detected ***: <unknown>
terminated
Aborted (core dumped)
```

```cpp
#include <iostream>

int main() {
    char buffer[10];

    std::cout << "What's your name?\n";

    // Oops - we're asking for trouble here:
    // If the user inputs more than 9 characters for
    // a name (remember need terminating NULL character)
    // we have a buffer overrun!
    std::cin >> buffer;

    std::cout << "Hello, " << buffer << "!\n";
}
```

UB Example – Out-of-Bounds Access (C++)

Windows 10 (MSVC) – works:

```
Past the end:  42
```

Ubuntu 18 (GCC) – crash:

```
*** stack smashing detected ***:
<unknown> terminated
Aborted (core dumped)
```

```cpp
#include <iostream>

int main() {
    int numbers[10];

    // Oops - out-of-bounds access!
    numbers[11] = 42;

    std::cout << "Past the end:  " << numbers[11];
}
```

UB Example – Null Pointer Access (C++)

Windows 10 (MSVC) – crash:

```
No output…
```

Ubuntu 18 (GCC) – crash:

```
Segmentation fault (core dumped)
```

```cpp
#include <iostream>

int main() {
    char data[20] = "My example string.";
    // Initialize pointer to Null:
    char* dataptr = nullptr;

    // Oops - dereferencing Null pointer:
    // Forgot to assign dataptr to address of data...
    std::cout << "data:  " << *dataptr << '\n';
}
```

UB Example – Memory Leak (C++)

Windows 10 (MSVC) – stack buffer overrun:

```
.....................
```

Ubuntu 18 (GCC)* – crash:

```
.....................................
.terminate called after throwing an
instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted (core dumped)
```

```cpp
#include <iostream>

struct stuff {
    char data[1024];
};

void process() {
    stuff* my_stuff = new stuff[1024];

    // Do stuff...

    // Oops - forgot to cleanup stuff, leaked allocation...
}

int main() {
    for (int i = 0; i < 1'000'000; ++i) {
        process();
        if (i % 100 == 0) {
            std::cout << '.' << std::flush;
        }
    }
}
```

UB Example – Iterator Invalidation (C++)

Windows 10 (MSVC) – access violation:

```
No output
```

Ubuntu 18 (GCC) – crash:

```
Segmentation fault (core dumped)
```

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers {1, 15, 23, 30, 42};

   for (auto i = numbers.begin(); i != numbers.end(); ++i) {
        if (*i % 15 == 0) {
            // Oops - iterator invalidation:
            numbers.push_back(*i);
        }
    }

    for (auto i: numbers) {
        std::cout << i << ' ';
    }
    std::cout << '\n';
}
```

UB Example – Signed Integer Overflow (C++)

Windows 10 (MSVC) – works:

```
int size:  4
max signed int:  2,147,483,647
int now:  -2,147,483,648
```

Ubuntu 18 (GCC) – same…

```cpp
#include <iostream>
#include <limits>
#include <locale>

int main() {
    int number = std::numeric_limits<int>::max();
    int numsize = sizeof(int);

    // Use commas for big numbers:
    std::cout.imbue(std::locale("en_US.utf8"));

    std::cout << "int size:  " << numsize << '\n'
              << "max signed int:  " << number << '\n';

    // Oops - signed int overflow:
    number += 1;
    std::cout << "int now:  " << number << '\n';
}
```

UB Example – Divide by Zero (C++)

Windows 10 (MSVC) – divide by zero exception:

```
No output
```

Ubuntu 18 (GCC) – crash:

```
Floating point exception (core
dumped)
```

```cpp
#include <iostream>

int main () {
    int numerator = 1;
    int denominator = 0;

    // Oops - divide by zero:
    int res = numerator/denominator;

    std::cout << "Result:  " << res << '\n';
}
```

Rust – Borrowing and the Borrow Checker

- Default behavior is to move values

```rust
fn main() {
    // Create new object example which owns allocated string data:
    let example = String::from("Hello!");

    // Send to function - default is to move (transfer ownership):
    // Once this occurs, example is "de-initializled"
    display(example);

    // Error - example in de-initialized state:
    println!("{}", example);
}

fn display(stuff: String) {
    println!("stuff:  {}", stuff);
}
```

```
C:\apps\working\rust\code\myproject [master +4 ~0 -0 !]> cargo run
   Compiling myproject v0.1.0 (C:\apps\working\rust\code\myproject)
error[E0382]: borrow of moved value: `example`
  --> src\main.rs:10:20
   |
3  |     let example = String::from("Hello!");
   |         ------- move occurs because `example` has type `std::string::String`, which does not implement the `Copy` trait
...
7  |     display(example);
   |             ------- value moved here
...
10 |     println!("{}", example);
   |                    ^^^^^^^ value borrowed here after move

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
error: could not compile `myproject`.
```

Rust – Borrowing and the Borrow Checker

- Can also copy object:

```rust
fn main() {
    // Create new object example which owns allocated string data:
    let example = String::from("Hello!");

    // Send to function - use .clone to copy:
    display(example.clone());

    // OK - example still valid:
    println!("{}", example);
}

fn display(stuff: String) {
    println!("stuff:  {}", stuff);
}
```

- Or loan out a reference to it:

```rust
fn main() {
    // Create new object example which owns allocated string data:
    let example = String::from("Hello!");

    // Loan out a copy (really a reference or alias):
    display(&example);

    // OK - example still valid:
    println!("{}", example);
}

// For string references use &str:
fn display(stuff: &str) {
    println!("stuff:  {}", stuff);
}
```

Rust – Borrowing and the Borrow Checker

- Mutating object:

```rust
fn main() {
    // Create new object example which owns allocated string data:
    let mut example = String::from("Hello!");

    // Loan out a copy (really a reference or alias):
    augment(&mut example);

    // OK - example still valid:
    println!("{}", example);
}

// For mutable string references, must use String:
fn augment(stuff: &mut String) {
    stuff.pop();
    stuff.push_str(", World!");
}
```

Rust – Borrowing and the Borrow Checker

Type	Ownership	Alias?	Mutate?	Owner Access
T	Owned	N/A	✔	Full
&T	Shared reference	✔	✘	Read-only
&mut T	Mutable reference	✔	✔	None

Rust – Compared to other Languages, Classes

```java
// Java:
public class MyClass {
    // Data:
    private int number = 42;
    private MyOtherClass c = new MyOtherClass();

    // Methods:
    public int count() {
        // do stuff...
    }
}
```

```cpp
// C++:
class MyClass {
private:  // Default
    // Data:
    int width;
    int height;

public:
    // Methods:
    void set_values(int, int);
    int area() { return width * height; }
};
```

```rust
// Rust:

// Data:
struct MyClass {
    number: i32,
    other: MyOtherClass,
}

// Methods:
impl MyClass {
    fn myMethodCountHere(&self) -> i32 {
        // do stuff...
    }
}
```

Rust – Compared to other Languages, Interfaces

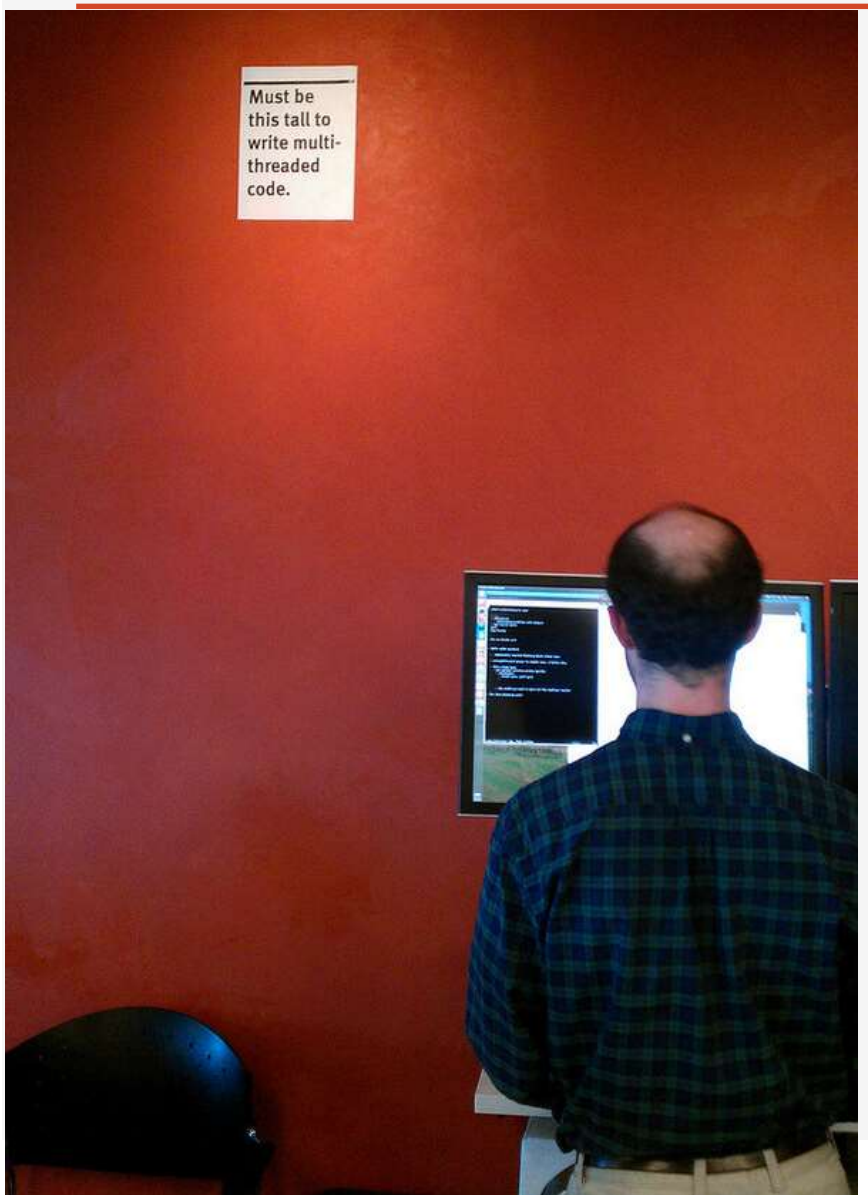
```java
// Java:
public interface MyInterface {
    void someMethod();

    default void someDefault(String str) {
        // Implementation
    }
}
```

```rust
// Rust:
trait Animal {
    fn noise(&self) -> &'static str;
    fn talk(&self) {
        println!("I do not talk to humans.");
    }
}

struct Horse { breed: &'static str }

impl Animal for Horse {
    fn noise(&self) -> &'static str {
        "neigh!"
    }

    fn talk(&self) {
        println!("{}!!!", self.noise());
    }
}
```

Rust – Functional Example

```rust
fn main() {
    println!("Find the sum of all squared odd numbers < 1000");
    let upper = 1000;
    let sum_of_squared_odd_nums: u32 =
        (0..).map(|n| n * n)
             .take_while(|&n| n < upper)
             .filter(|n| is_odd(*n))
             .fold(0, |sum, i| sum + i);  // Higher order functions
    println!("Result:  {}", sum_of_squared_odd_nums);
}

fn is_odd(n: u32) -> bool {
    n % 2 == 1
}
```

Thoughts on concurrent programming in C/C++



- Mozilla has a developer hierarchy from junior to senior
  - In addition, four engineers have achieved the highest level – distinguished engineer
  - One of them, David Baron, has a sign by his desk about concurrent programming
- Firefox's CSS rendering engine is an example of an "embarrassingly parallel" opportunity
  - Firefox is written in C++
  - The team found it too difficult to develop a highly concurrent secure and maintainable solution in C++

Image Source: https://bholley.net/blog/2015/must-be-this-tall-to-write-multi-threaded-code.html

UB Ex – Data Race (C++)

Windows 10 (MSVC) – works:

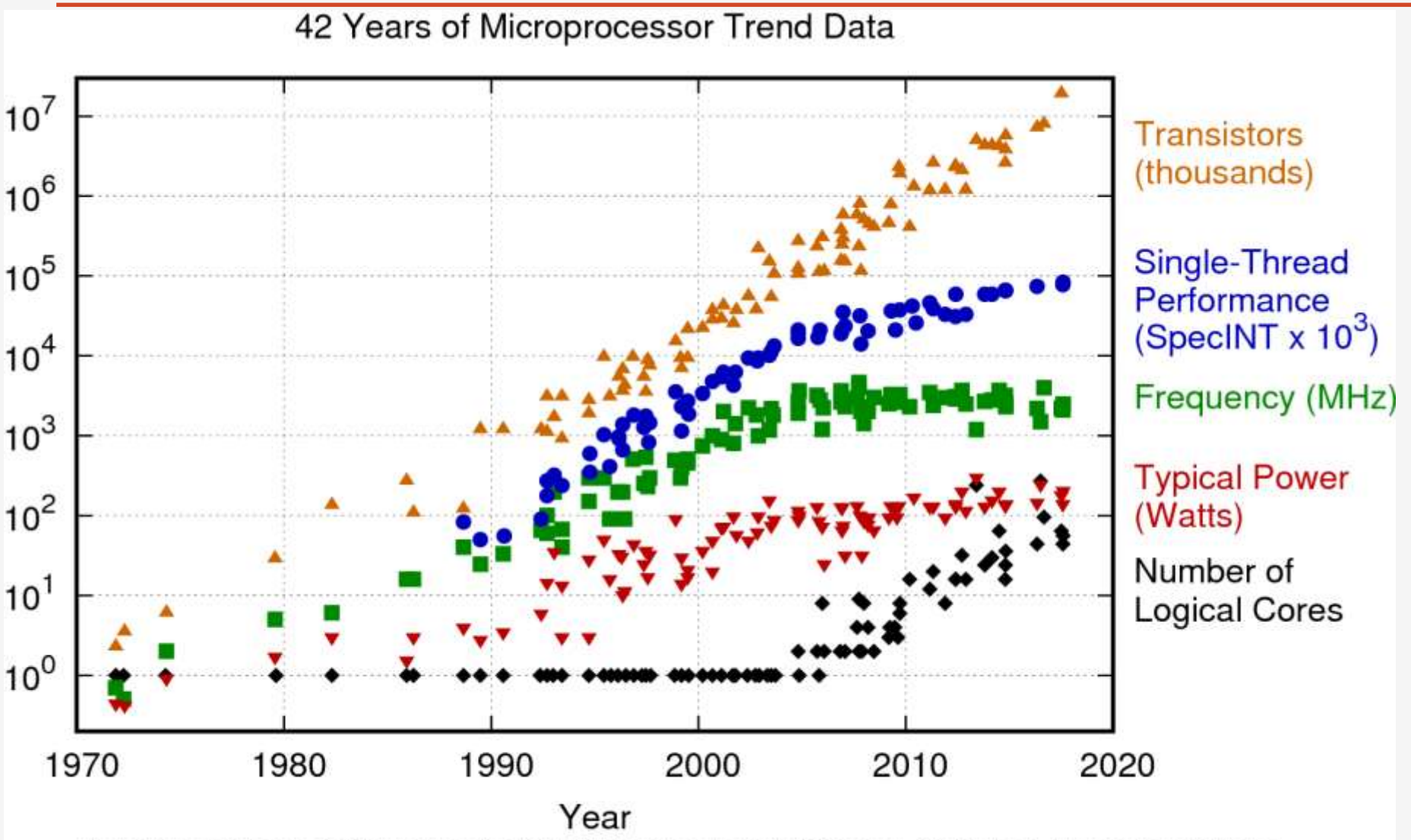
```
Results vary (e.g.):
Insufficient funds!
account1.balance: 60
account2.balance: 140
```

Ubuntu 18 (GCC) – works:

```
Results vary (e.g.):
account1.balance: 185
account2.balance: 15
```

```cpp
// Based on https://www.modernescpp.com/index.php/race-condition-versus-data-race
#include <functional>
#include <iostream>
#include <thread>

struct Account{
    int balance {100};
};

void transferMoney(int amount, Account& from, Account& to) {
    if (from.balance >= amount) {
        from.balance -= amount;
        to.balance += amount;
    } else {
        std::cout << "Insufficient funds!\n";
    }
}

int main(){
    std::cout << std::endl;

    Account account1;
    Account account2;

    std::thread thr1(transferMoney, 50, std::ref(account1), std::ref(account2));   // 50, 150
    std::thread thr2(transferMoney, 130, std::ref(account2), std::ref(account1));  // 180, 20
    std::thread thr3(transferMoney, 120, std::ref(account1), std::ref(account2));  // 60, 140
    std::thread thr4(transferMoney, 125, std::ref(account2), std::ref(account1));  // 185, 15

    thr1.join();
    thr2.join();
    thr3.join();
    thr4.join();

    std::cout << "account1.balance: " << account1.balance << '\n'
              << "account2.balance: " << account2.balance << '\n';
}
```

CPU Trends



- CPU Speeds flat since 2005
- Single-threaded (non-concurrent) performance gains near flat
- Efficient/ Concurrent/ Parallel/ Distributed programming key

Image Source: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

One Measure of Performance

- TechEmpower Web Framework Benchmarks, Round 18 (July 9, 2019)
  - Rust has top 2 slots
  - Next closest is C at 65% of speed, Go at 62.1% of speed
  - Then Java at 57.4% and C++ at 51.3%

Best fortunes responses per second, Dell R440 Xeon Gold + 10 GbE (371 tests)

Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng
1	actix-core	702,165	100.0%	0	Plt	Rus
2	actix-pg	632,672	90.1%	0	Mcr	Rus
3	h2o	456,058	65.0%	0	Plt	C
4	atreugo-prefork-quicktemplate	435,874	62.1%	0	Plt	Go
5	vertx-postgres	403,232	57.4%	0	Plt	Jav
6	ulib-postgres	359,874	51.3%	0	Plt	C++
7	fasthttp-postgresql-prefork	352,914	50.3%	0	Plt	Go
8	greenlightning	341,347	48.6%	0	Mcr	Jav
9	cpoll_cppsp-raw	326,149	46.4%	0	Plt	C++
10	fasthttp-postgresql	324,171	46.2%	0	Plt	Go

Image Source: https://www.techempower.com/benchmarks/

Rust – Concurrent Example

```rust
// Parallel quick sort:
// Takes mutable reference to slice of integers
fn qsort(vec: &mut [i32]) {
    if vec.len() <= 1 { return; }
    let pivot = vec[random(vec.len())];
    let mid = vec.partition(vec, pivot);
    // Split into left and right "views"
    // Trading in one mutable reference to array for two
    let (less, greater) = vec.split_at_mut(mid);
    // Use Rayon crate to start two threads and join results
    // Compiler enforces that can't use same/overlapping
    // mutable references here (no data races!)
    rayon::join(|| qsort(less),
                || qsort(greater));
    // Safe to use multiple threads because each slice is
    // unaliased!
}
```

Intro to Rust – Using Cargo

```
> cargo search bingrep
bingrep = "0.8.1"    # Cross-platform binary parser and colorizer
goblin = "0.1.3"     # An impish, cross-platform, ELF, Mach-o, and PE binary parsing and loading crate

> cargo install bingrep
    Updating crates.io index
  Downloaded bingrep v0.8.1
  Downloaded 1 crate (617.2 KB) in 1.19s
  Installing bingrep v0.8.1
  Downloaded cpp_demangle v0.2.14
  Downloaded prettytable-rs v0.8.0
  (…)
   Compiling proc-macro2 v1.0.7
   Compiling unicode-xid v0.2.0
   (…)
   Compiling bingrep v0.8.1
    Finished release [optimized] target(s) in 2m 28s
  Installing C:\Users\js646y\.cargo\bin\bingrep.exe
   Installed package `bingrep v0.8.1` (executable `bingrep.exe`)

> bingrep …
```

Rust – Additional Topics

- Error Handling
  - Basics with panicking
  - Basics with Optional and Result
- Lifetimes
- Object Oriented Features
  - Can’t inherit from structs but can specialize methods
  - Traits
  - Philosophy of separating data and behavior

Questions

Image Source: Machine Design

References

- Coming…