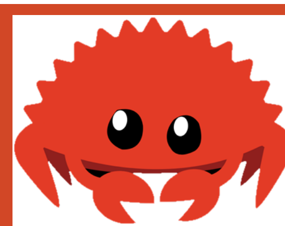
Rust, a C/C++ Replacement?

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

- Dennis Ritchie and Ken Thompson – Pioneers of modern technology
- In the 1960s, Bell Labs was collaborating with MIT and General Electric on Multics – a next-generation time-sharing operating system
    - Dissatisfied with the progress, Bell Labs pulled out in 1969
- One of the project participants, Ken Thompson, really liked the hierarchical file system and the shell ideas from Multics though
    - He implemented his own file system on a spare PDP-7, but decided he really needed to test it
    - So he did what any good engineer would, he wrote what became UNIX to test his file system
    - At the same time, Thompson developed the B programming language based off the BCPL language he used in the Multics project
- Thompson's coworker, Dennis Ritchie also worked on developing B
    - Ritchie liked B, but when they migrated it from a PDP-7 to a PDP-11 several deficiencies became apparent
    - As a side note, in today's terms these computers cost over $500,000 and filled a room
        - The first popular Personal Computer, the Altair 8800, wouldn't be

    released until 1975
  - One of the biggest one was that B was typeless and Ritchie determined that a typing scheme was necessary
  - He began incorporating this into B and New B evolved into C
- It is worth noting the environment that Thompson and Ritchie worked in
  - A PDP-7 had a single 18-bit CPU/core with up to 144kb of RAM – a completely different architecture than today
  - External storage consisted of a tape drive – hard drives weren't yet that popular
  - A printer served as a display – you'll note no monitor in the picture
  - Computers were stand alone – the first Ethernet networking standard wouldn't emerge until 1980
  - Security wasn't a consideration
- In addition, per Ritchie none of BCPL, B, and C have strong character data support – rather characters are special cases of integers with strings being vectors of integers
  - Furthermore, while BCPL included a character count at the beginning of a string, B choose to remove this and instead use a special termination character
  - C retained this design which in hind sight was an unfortunate choice with many painful security consequences

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

We’re learned a lot about programming languages and type systems in the intervening decades
That’s why there are a bunch of relatively new popular programming languages including Swift, Go, and Rust among others

However, this will be a frank discussion of some of the weaknesses of C and C++ so if you’re a fan, brace yourself!

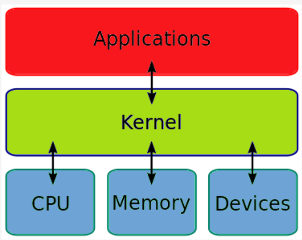
Using The Right Tool…

- 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

This talk is designed to challenge your assumptions and make you think!
Also note that many in the C++ community think it takes 3 – 5 years of professionally using C++ to really learn it
So I can show you the basics of the language and hopefully make you curious, but I can't teach you Rust in 75 minutes!

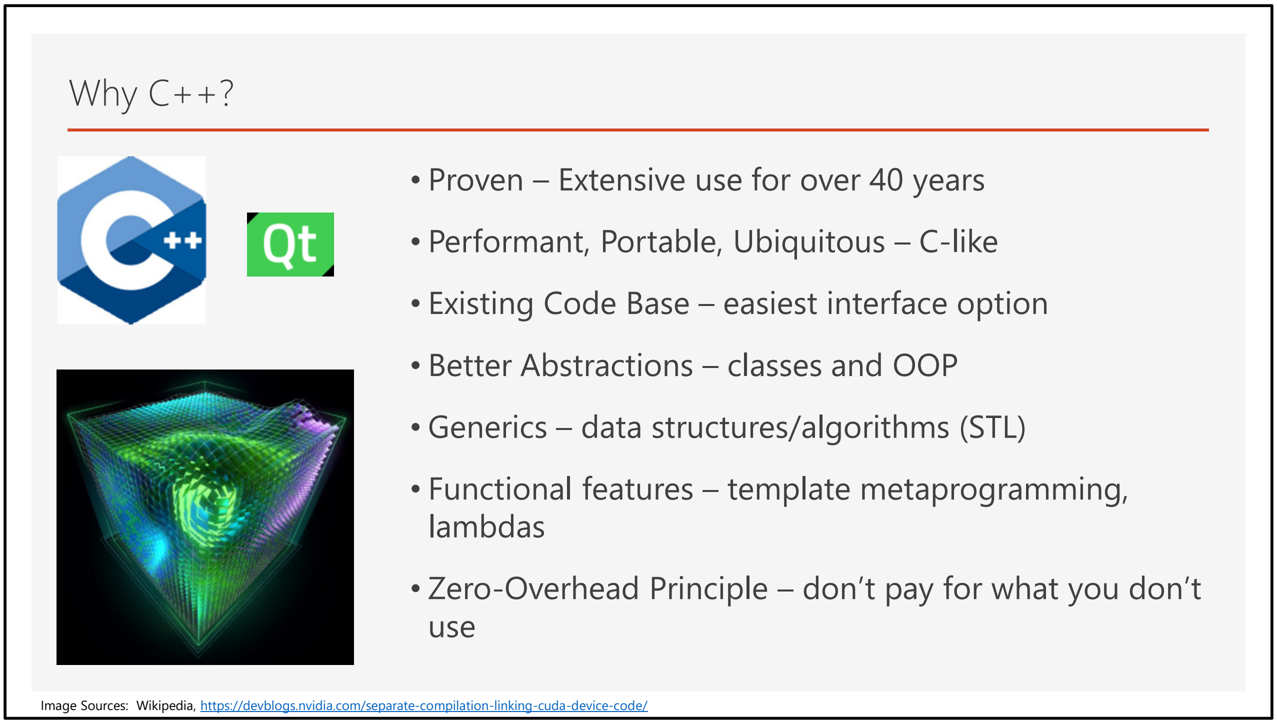
Why C?



- 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/

- Although many refer to C as “portable assembler”, this is a contentious claim
    - C was developed alongside UNIX
    - C was created to be efficient and performant enough to use instead of assembler
    - UNIX was ported from assembly to C
    - As UNIX was ported to a variety of other platforms in addition to the original PDP architecture, C and the C standard libraries were a key tool
- C is both the dominant embedded and operating system programming language
    - While higher level operating system components – especially graphics typically use C++, the kernel and lower levels are virtually all C, even in Windows



- C++ tries to tighten up C's type system
    - It does offer some improvements
    - However, to maintain backwards compatibility, it still has most of C's weaknesses
- C++ is used extensively in graphics and graphics subsystems
    - This includes the most popular cross-platform framework – Qt
    - It also includes WxWidgets and Microsoft's MFC and WTL frameworks
    - However there are some notable exceptions such as GTK which is written in C

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

- The lack of a standard build and packaging tools along with no central registry causes many systemic issues for the C and C++ communities
  - Sharing programs and libraries is a challenge
  - Because of this, many wheels are continuously reinvented
  - Collaboration is challenging and the community is fragmented
  - The standard example or joke about this for C/C++ is JSON
    - Whereas all modern languages have a library which implements JSON serialization/deserialization, C/C++ don't
    - Instead, you will find dozens of options with no standard
    - Because there is no standard build/packaging system each one takes different approaches
    - So things which are trivial in other languages actually become a project in C/C++
    - Even Herb Sutter, the C++ ISO WG Chairman will joke about this
- There are many build systems and packaging systems to choose from but no standard one
  - These will include collections of popular "packages" which number in the 100s to 1000s

    - However, modern languages have package registries in the 100s of thousands to over a million
- Furthermore, the lack of a reference implementation exacerbates the problems and makes its extremely difficult to come up with standards
    - Instead, each vendor/implementer has their own ideas and goals which are often incompatible

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

- C-strings are null terminated
  - In addition to the use of an "in-band" character and security issues, this is also a performance problem
  - Determining the length of a string requires scanning for the null character
  - O(n) operation instead of O(1) when length is part of a string type
  - C++ std::string knows its length, but there are still many places where C-strings are required
- *Source of confusion and security issues
  - Because strings must add a null termination character, the length of a string must account for this
  - In addition, using a special character which is part of the standard character set results in issues, limitations, and vulnerabilities
- **Most programming languages keep track of the length of a string
  - This provides better performance because you know how long the string is rather than having to parse it
  - This provides better security as there are no off-by-one errors or worrying about interpreting special characters incorrectly
  - Ironically, the predecessor to C, BCPL, did use string lengths and not a

termination character – this was an unfortunate design choice

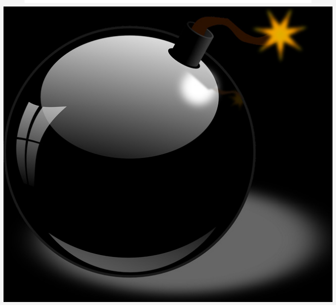
Issues with C/C++ versus Modern Programming Languages?

- 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 a problem?
  - Arrays not convenient to pass around
    - Source of security issues
  - Templates are challenging to keep type safe
    - The diagnostics are awful and difficult to understand/troubleshoot
  - Since macros don't understand the language they can cause problems/bugs

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)

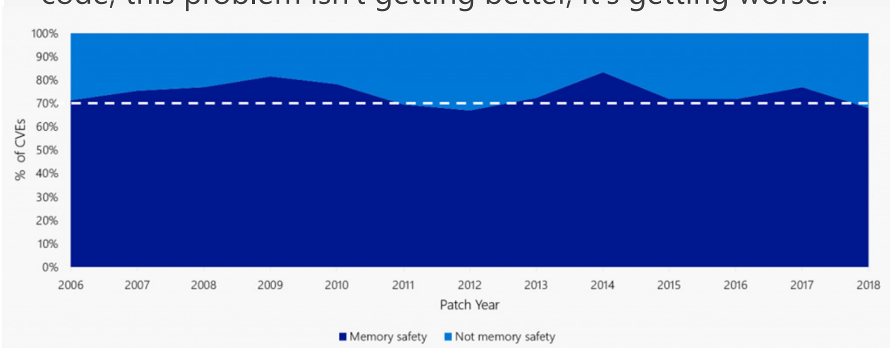
Image Sources: safetysignsupplies.co.uk, https://www.fool.com/retirement/general/2015/01/13/the-time-bomb-lurking-in-your-social-security.aspx

- *Source of many (most?) security issues
- See also: https://en.wikipedia.org/wiki/Memory_safety

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/

- Can't we just use better tools like static analyzers/sanitizers, and safe coding guidelines?
    - According to Microsoft, this isn't helping
- An interesting question is this:
    - Is the job of a software developer to focus on mastering incredibly difficult security problems?
    - Or is it to focus on producing applications that users value?

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

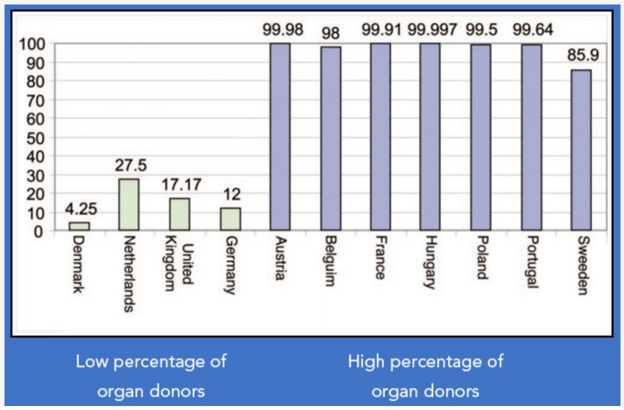
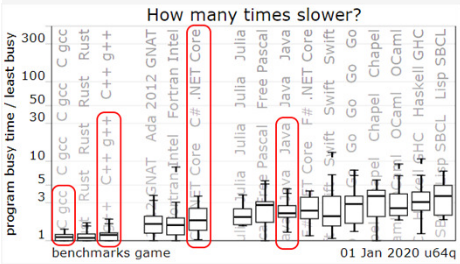


Image Source: https://jamesclear.com/environment-design-organ-donation

- I wasn't able to find a definitive study of the impact of better tooling and secure coding practices
    - However, the general consensus seems to be that it's not enough
- Sometimes it can help to look at different disciplines for ideas
- In terms of solving critical issues, on average 20 people per day die waiting for an organ transplant
    - Looking across countries, why do some have near universal participation in organ donation and yet others struggle to find enough?
    - It turns out it's a policy issue.
        - Countries with high donor rates have an opt-out policy
        - Countries with low donor rates have an opt-in policy

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

- Traditional memory safe languages like C#, Java, Python, JavaScript, and PHP work well in many contexts
- However, when raw computational speed and efficiency are most important, systems programming languages tend to be faster
  - While benchmarks are never perfect, in general C/C++ outperform memory safe languages
- There is one interesting new entry – Rust
  - It has performance on par with C/C++ and we'

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/

Mozilla engineers such as Jim Blandy describe that when you program in C++ it’s stressful because if you aren’t perfect, millions of people will get owned by your mistake.
That’s because Firefox is still developed primarily in C++

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

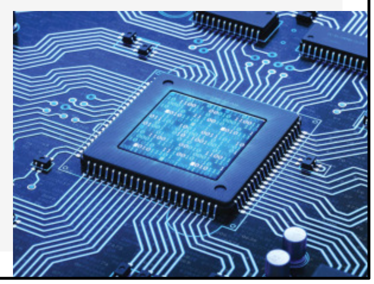


Image Source: ul.com

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

This is a list of the 8 most prominent/popular general programming languages plus three extra ones in blue at the bottom

I'm using six of the most popular sources for ranking programming languages

Note – For StackOverflow, L = Most Loved rank, D = Most Dreaded rank

- JavaScript is over 10 for both loved and dreaded, but slightly more loved

I don't include VB.Net in the programming language list.

- C# is the Microsoft .Net language targeted at developers
- VB is focused more on non-traditional developers

The first 7 languages are clearly dominant players.

Swift is debatable depending on which source you consider

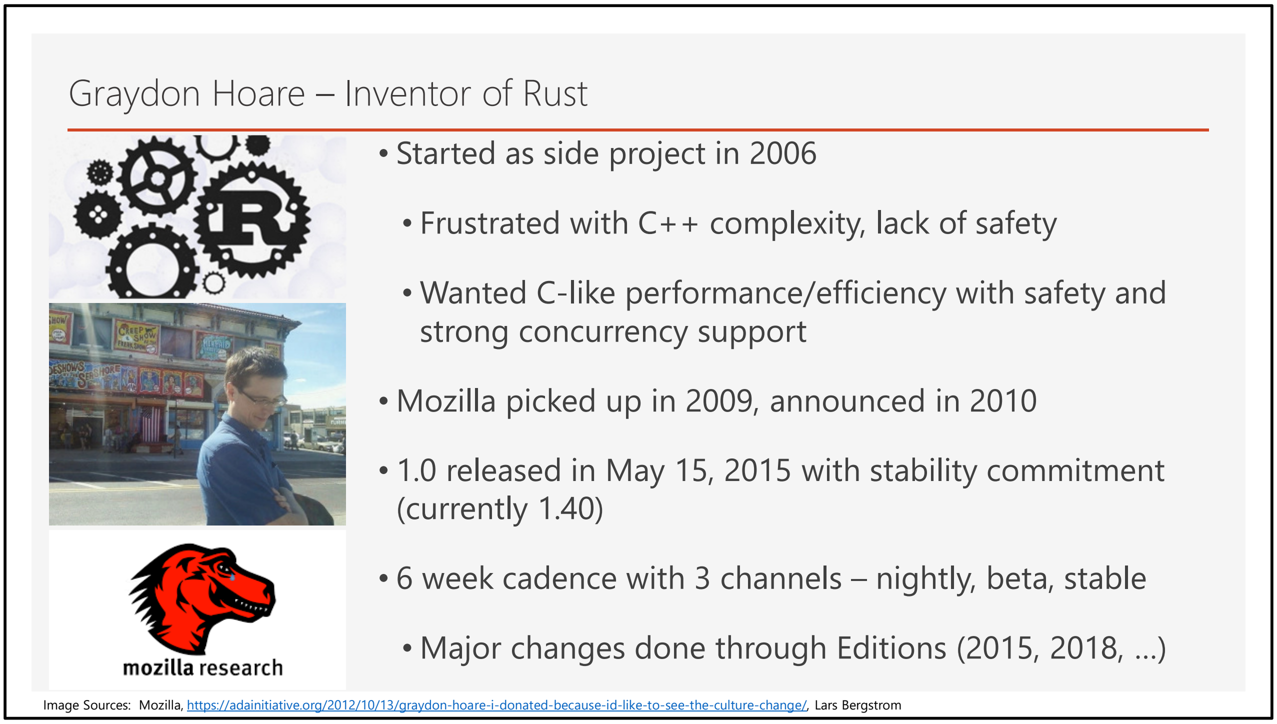
- However, from consulting half a dozen sources, Swift was in the top more than any other alternative such as Ruby or Go

I also excluded non-traditional programming languages and DSLs like HTML, CSS, SQL, and SQL Dialects

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)

Note that Go is not in the systems programming list – Wikipedia doesn't include it and I don't either.
I classify a systems programming language as one you'd write an operating system kernel with.
That definition excludes Go which uses a Garbage Collector for automatic memory management.
Writing a kernel requires tight control over memory allocation and deallocation.
That's not to say Go isn't a great programming language, just that it's choice to use GC excludes it from this category.



See: https://github.com/graydon/rust-prehistory

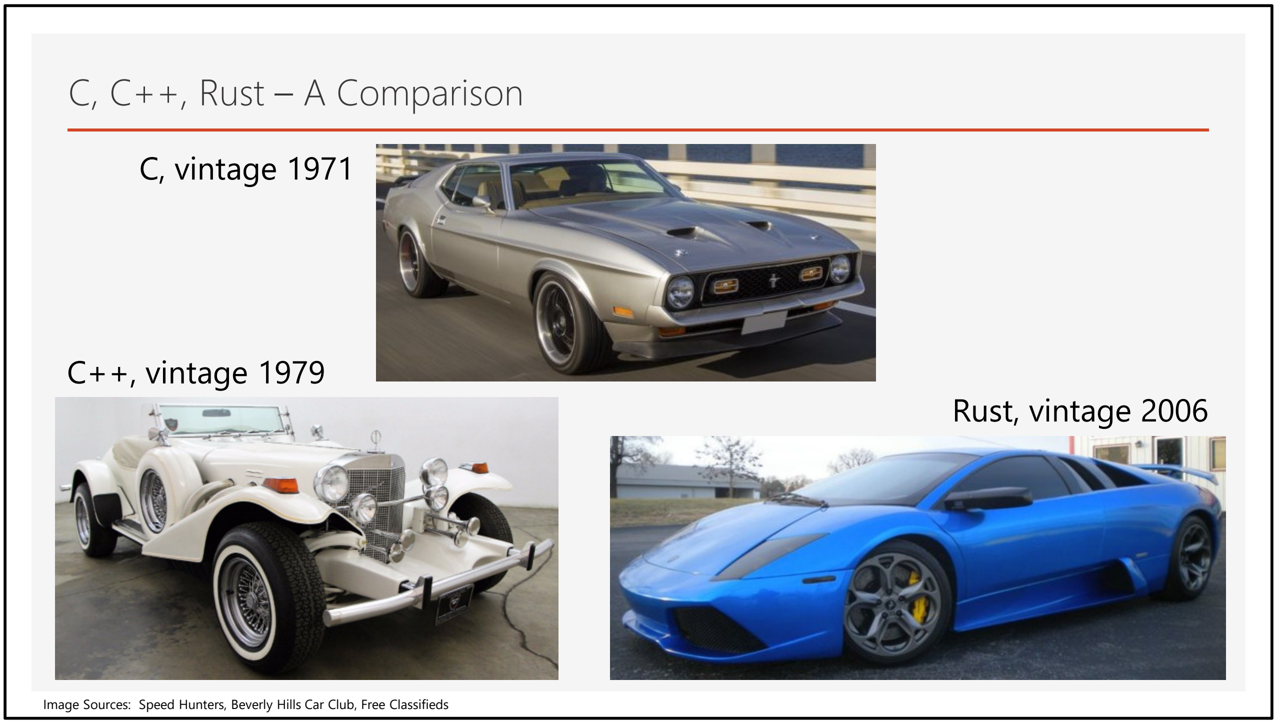
It's worth noting that the C++ community really likes the concept of editions and is considering its adoption

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

Rust has a provably correct type system which is much stronger than what's available for C or even C++
Rust descends from ML, leveraging its type inference and like many functional languages provides a provably correct type system
As a result, Rust is able to guarantee memory integrity and safety
As a result of this there is no UB and segmentation faults aren't possible
In addition, this also prevents data races and makes parallel programming vastly easier



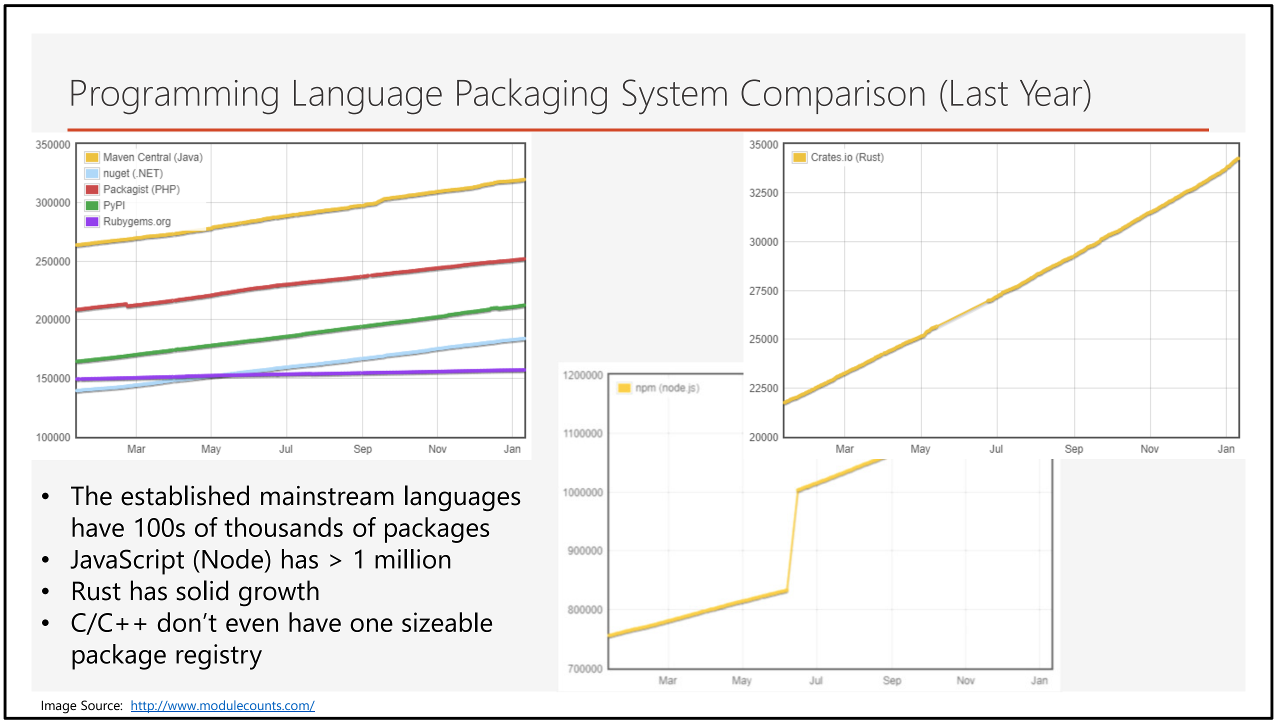
Can you improve on a classic and bring along the old guard?

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

Note: For C/C++ there seem to be 3 distinct prominent communities:

- Microsoft/Windows users who use Visual Studio or MSVC and primarily C++
- UNIX/Linux users who use GCC and primarily C
- Multi-platform users who primarily use C++ seem to gravitate towards Clang



On an interesting note, Emacs (4420) and Vim (5536) both have more available packages through a package manager than C or C++!

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

*Undefined Behavior in 2017, by John Regehr and Pascal Cuoq – compiler/static analysis experts

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

- Type = roughly defined as assigning meaning to bits of computer memory, i.e., how to interpret them – integer, floating point, string?
- Type System = set of rules for working with types; It attempts to limit errors by preventing operations that don't make sense
- [1]Assembly is an extreme with no types/typing whatsoever!
- [2]In Bash/Tcl everything is a string
- [3]Python objects have a type, but variable names are essentially just references to objects and have no type themselves
- [4]Functions/Methods can enforce typing through Duck Typing or explicit checks
- [5]Functions/Methods can enforce typing through static analysis with type annotations and tooling (e.g., mypy)
- [6]C/C++ have a weak type system in the sense that they do:
    - unsafe implicit conversions/coercions
    - allow type punning (subversion) through void*
    - allow raw (unsafe) pointer access/dereferencing
- ML/Haskell use a provably correct type system (formal verification)
    - [7]ML/Haskell provide static type inference so variable type annotations are typically not needed
- Rust is like C++ combined with ML/Haskell – has a mostly proven type system and guarantees memory safety

- [7]Rust provides static type inference so variable type annotations are typically not needed
- [8]Rust has an unsafe mode which allows C-like access

See: https://en.wikipedia.org/wiki/Strong_and_weak_typing

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

The C++ community in particular has gotten more including
It’s come a long way from the early days of UNIX where RTFM and you have the source code were typical answers to questions
However, Rust incorporates inclusivity at its very core

- As a contrast the Linux Kernel and git communities can be somewhat harsh and expert-only friendly
  - Famously in 2018 Linus Torvalds apologized for his “unprofessional rants”
- The Rust community seeks out everyone include beginners
  - When something isn’t clear the community tries to improve the documentation or the language
  - As a result Rust has excellent ergonomics
  - When you start using the compiler you will really appreciate this – especially if coming from C++

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)	

Note – in an interview on CppCast episode 200, ISO C++ Chairman Herb Sutter said it they had it to do over again, this is how they'd do it for C++ (for integers and floating points)

Examples:

- 32-bit Signed Integer: $-2^{31}...2^{31}-1$ (-2,147,483,648…2,147,483,647)
- 32-bit Unsigned Integer: $0...2^{32}-1$ (0…4,294,967,295)

What Every Programmer Should Know About Floating-Point Arithmetic:
https://floating-point-gui.de/

Rust Variable Types – Additional Scalars

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

Unicode support means we can represent all the world's languages and symbols including scientific and mathematical symbols, music, currency, check boxes, sports, emojis, and more.

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

Most scripting language have the concept of slices

- You can use a range of values from an array or collection

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…	

Note – many collections use generics to support a variety of types

- T here represents a generic type
- K, V represents a generic type for the key and a separate type for the value

Rust – Using Variables

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

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

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

Variables immutable by default to provide strong concurrency support
Shared mutable state is the root of all evil for concurrency
Variables can opt-in to being mutable with the mut keyword

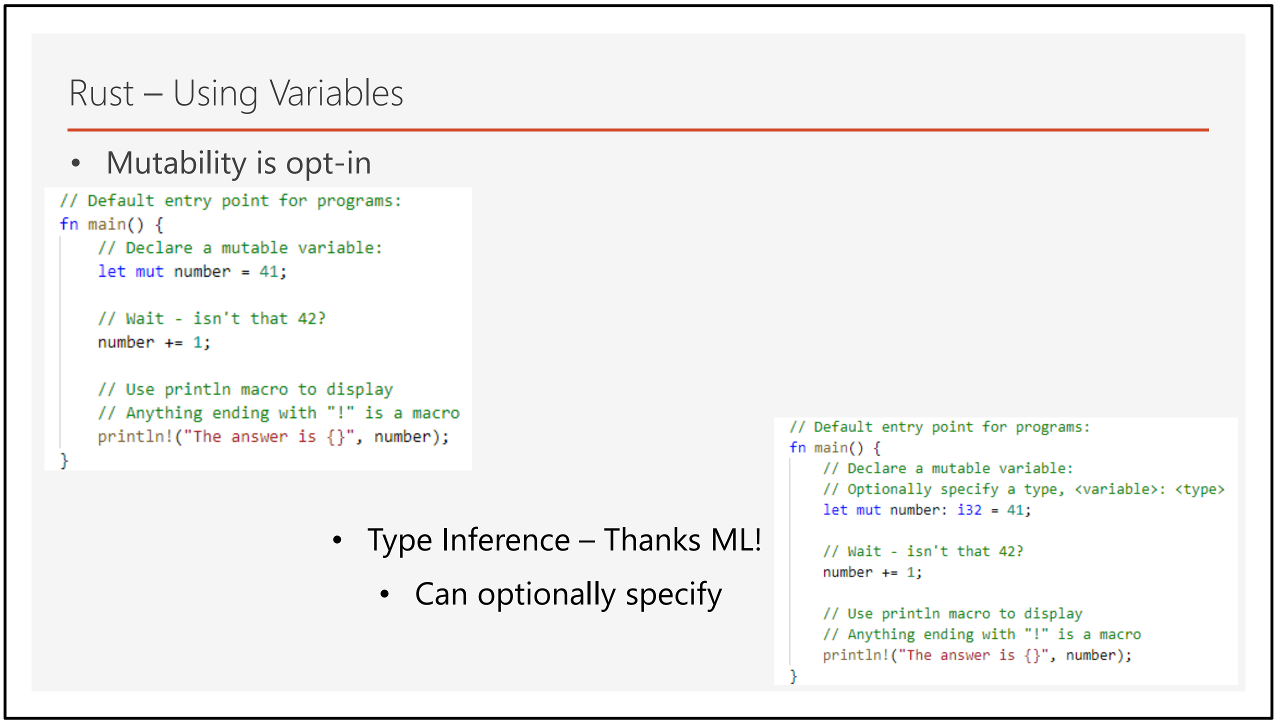
Rust – Using Variables

- 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.
```

Variables immutable by default to provide strong concurrency support
Shared mutable state is the root of all evil for concurrency
Variables can opt-in to being mutable with the mut keyword



Use mut keyword to make variable mutable
One thing I really like about Python is while I can specify types, it's not required
That said, requiring types provides better safety, documentation, and facilitates refactoring
Rust derives from ML among other languages

- This gives it type inference
- Static typing without having to explicitly specify the type!
- The benefits of strong typing without the hassle of specifying every type!
- Can specify type if desired though

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);
}
```

Notice the orange squiggles in Visual Studio Code – these are unused variables

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

```
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:

```
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)!

```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!"),
    }
}
```

```
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 – 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

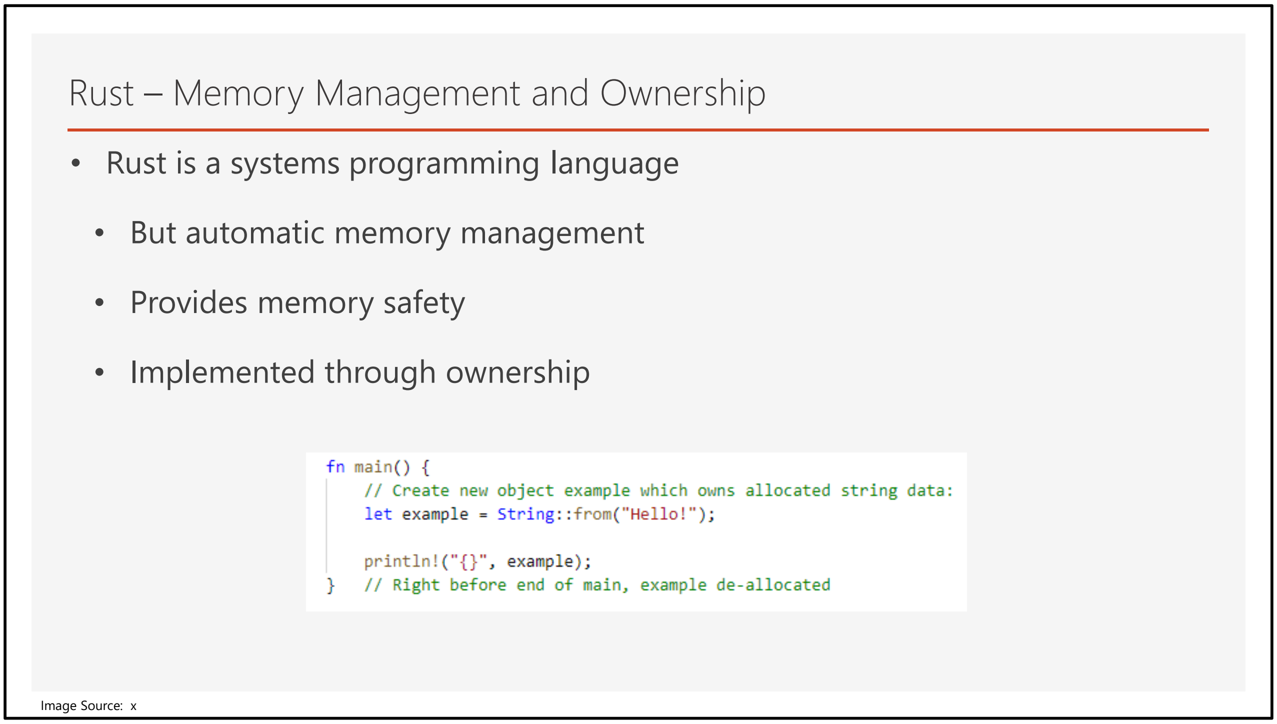
```rust
struct Point {
    name:  String,
    x: i16,
    y: i16,
}

// 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());
}
```

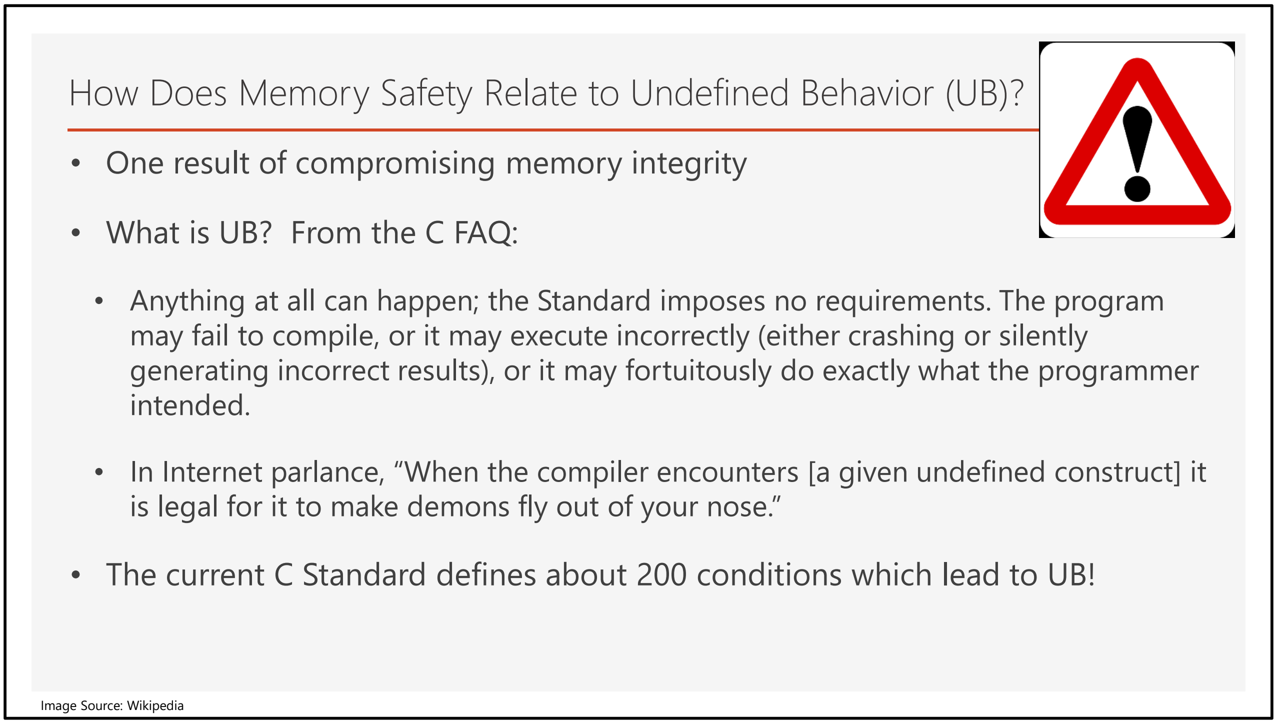
Rust – Memory Management and Ownership

- 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



- Uses similar model as C++ RAII
- Memory allocated when object created
  - Variable owns data
- Memory deallocated when object goes out of scope
- Access to variable after deallocation not allowed – compiler error



Even more fun – UB allows aggressive compiler optimizations so a so-called working program can break at any time when re-compiled as the compiler may implement new optimizations.

Understanding Integer Overflow in C/C++:

https://www.cs.utah.edu/~regehr/papers/overflow12.pdf

http://catb.org/jargon/html/N/nasal-demons.html

Current C17 Standard has over 10 pages of small print describing conditions which result in Undefined Behavior and 58 references to UB!

Current C++17 standard has 75 references to UB!

- Unfortunately there isn't an explicit list of conditions like for C

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++)

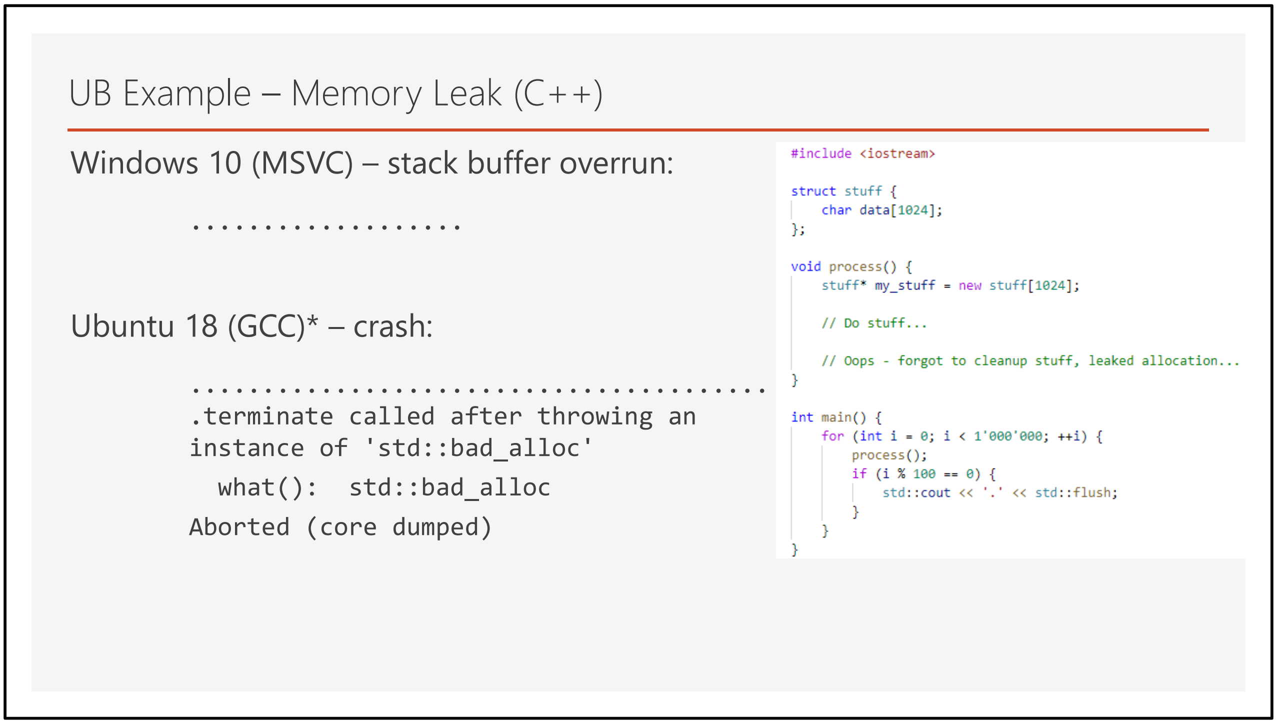
Windows 10 (MSVC) – crash:

```
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';
}
```



*Had to pass –m32 flag to create 32 bit binary, otherwise doesn't crash! (Default is 64 bit binary with insanely large address space…)

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:

```
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:

```
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:

```
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

- In a modern concurrent language, shared mutable state is ticking time bomb
- Rust's compiler enforces a reader-writer lock model
- The owner has full access
    - If the owner loans out one or more read-only references, his access becomes read-only until all references are destroyed
        - While any read-only references exist, it's not possible to mutate the variable
    - If the owner loans out a mutable reference then no other references can be given and the mutable reference has exclusive access
        - Even the owner will be blocked from accessing the object while the mutable reference exists

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

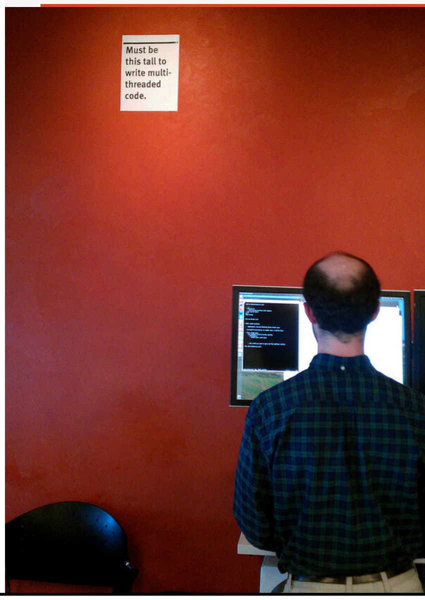
```java
// Java:
public interface MyInterface {
    void someMethod();

    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

- David Baron, who works out of Mozilla's San Francisco Office, has a sign next to his cubicle
- Although David is tall at well over six feet, the sign towers over him
- The sign reads, "Must be this tall to write multi-threaded code."

UB Ex – Data Race (C++)

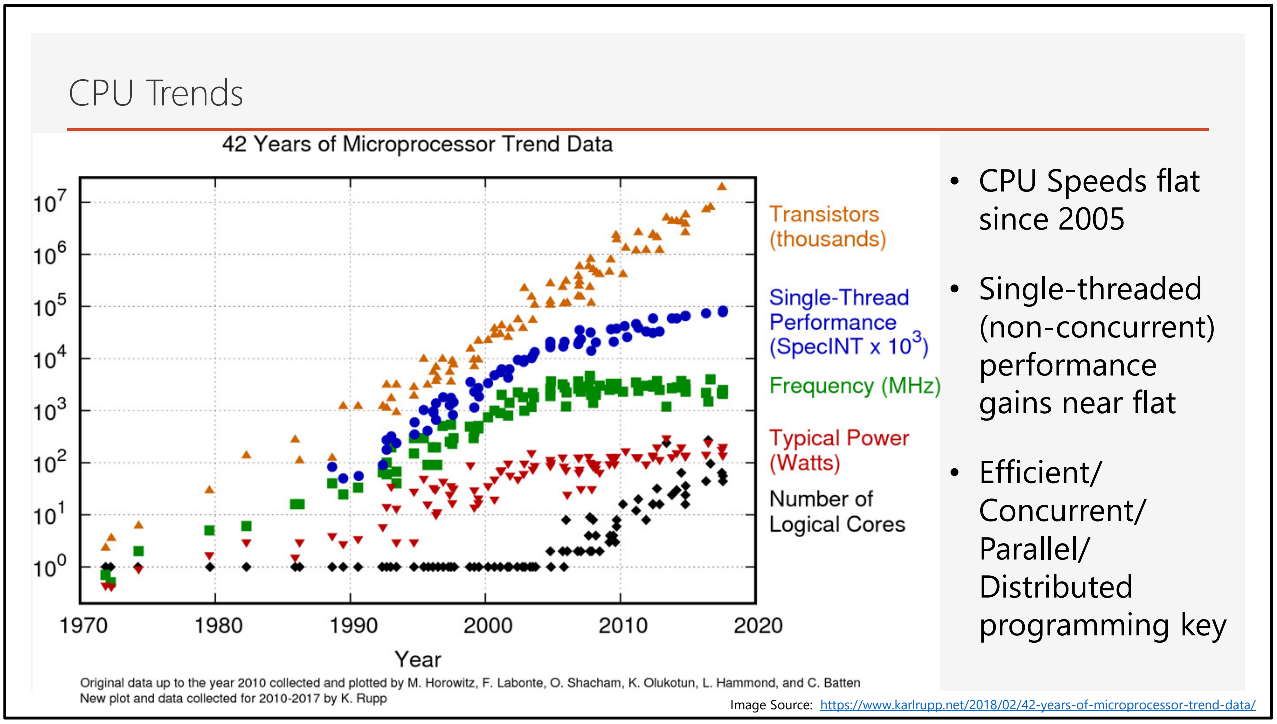
Windows 10 (MSVC) – works:

```
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';
}
```



- For a long time, the theory was don’t worry about performance – Moore’s law will save you
- However, many think Moore’s law is in trouble
- Virtually all performance gains for the past 15 years have come from concurrency

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…