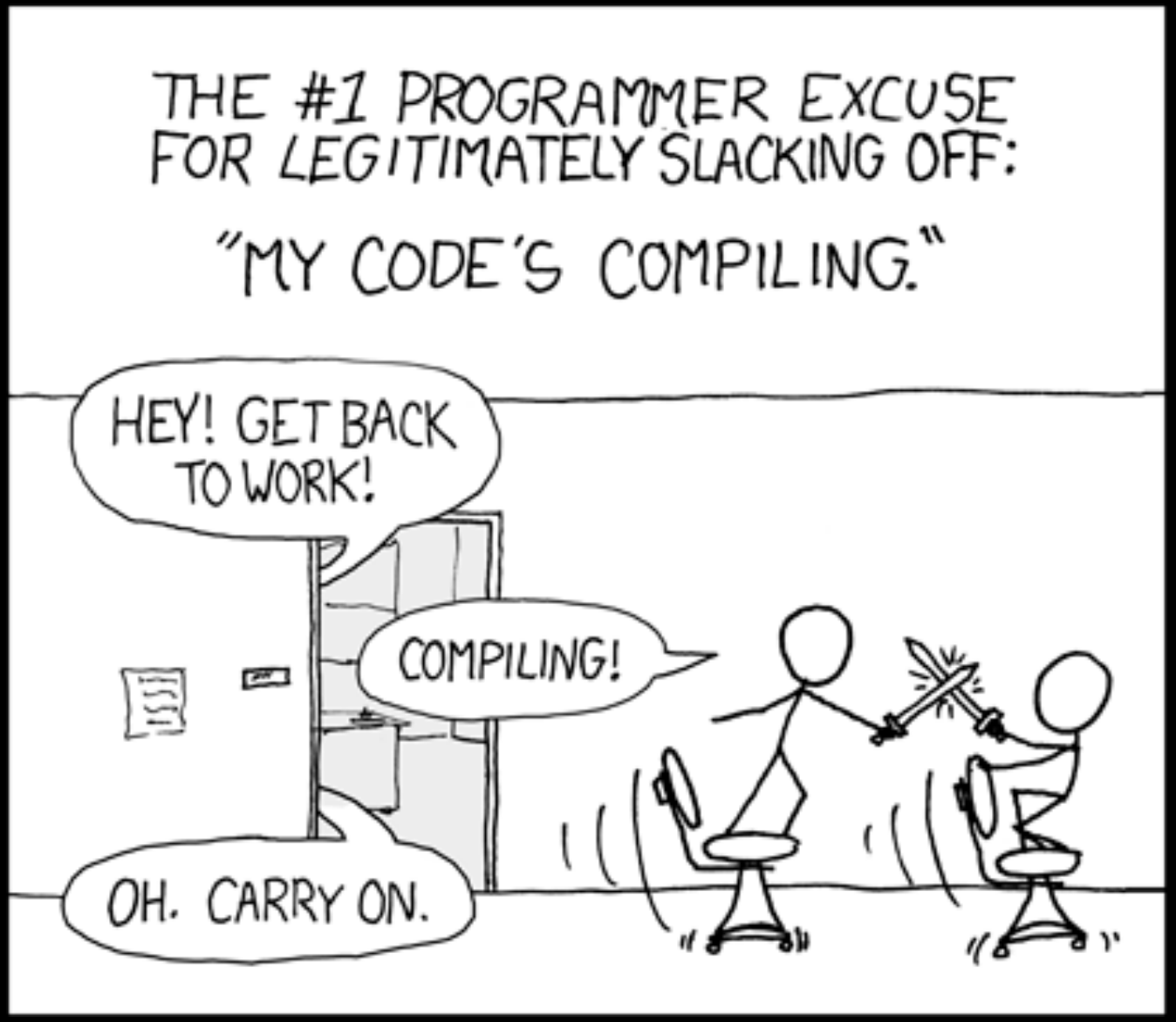

https://xkcd.com/303/

CS 152: *Programming Language Paradigms*

Rust

Prof. Tom Austin
San José State University

What is wrong with C/C++?

- Painfully slow build times
- Not memory safe
- No good concurrency story

"When the three of us [Ken Thompson, Rob Pike, and Robert Griesemer] got started, it was pure research. The three of us got together and decided that we hated C++."

--Ken Thompson on the motivation for Go

"C makes it easy to shoot yourself in the foot;
C++ makes it harder, but when you do it
blows your whole leg off."

--Bjarne Stroustrup

C++ is a horrible language.

--Linus Torvalds

Tony Hoare's billion dollar mistake

"But I couldn't resist the temptation to put in a `null` reference, simply because it was so easy to implement. This has led to **innumerable errors, vulnerabilities, and system crashes**, which have probably caused a billion dollars of pain and damage in the last forty years."

Challenges with C

A buggy C method

```
int* zero_negs(int a[], int len){
  int res[len];
  for (int i=0; i<len; i++) {
    if (a[i] < 0) res[i] = 0;
    else res[i] = a[i];
  }
  return res;
}
```

Fixed?

```
int* zero_negs(int a[], int len){
  int *res=malloc(sizeof(int)*len);
  for (int i=0; i<len; i++) {
    if (a[i] < 0) res[i] = 0;
    else res[i] = a[i];
  }
  return res;
}
```

A consumer of data, which frees the data.

```
void print_arr(int a[], int len){
  for (int i=0; i<len; i++) {
    printf("%d ", a[i]);
  }
  printf("\n");
  free(a);
}
```

But what if the consumer is called twice?

```
int main(int argc, char** argv) {
  int nums[] = {0,12,5,-42,9,7,-18,0};
  int n = 8;
  int *no_negs = zero_out_negs(nums,n);
  print_arr(no_negs, n);
  // ... Sometime later in the code.
  // Freeing memory twice.
  print_arr(no_negs, n);
}
```

Memory Management

- C/C++ force the programmer to manage memory, which can cause:
  - Memory leaks
  - Dangling pointers
- Java uses a *garbage collector*
  - Stop-the-world gc.
  - Applications stops while gc runs.

Rust history

- Developed by Graydon Hoare of Mozilla
- Latest version: Rust 1.39
- Used in
  - Project Servo: layout engine for Firefox
  - The Rust compiler
- Emphasis:
  - Safety
  - Control of memory layout
  - Concurrency

hello_world.rs

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

Denotes that `println` is a macro

```
$ rustc hello_world.rs
$ ./hello_world
Hello, world!
```

Simple Rust program

(in-class)

Primitive Types

- signed integers: `i8`, `i16`, `i32`, `i64`
- unsigned integers: `u8`, `u16`, `u32`, `u64`
- pointer sizes: `isize` (signed), `usize` (unsigned)
- floating point: `f32`, `f64`
- `char`, `bool`
- arrays `[1,2,3]` and tuples `(1,true)`
- the unit type `()`

Memory management approaches revisited

- C/C++
  - manually managed
  - let the programmer beware
- Java
  - Virtual machine with garbage collector
  - Run-time enforcement of key properties
  - Performance overhead

Rust memory management

- No run-time or garbage collection
- Compiler statically enforces memory safety
- Uses RAII strategy
  - **Resource Acquisition Is Initialization**
  - resource allocation done at initialization
  - resource deallocation done when the object goes out of scope

Handling arrays in C, Java, & Rust

(in-class)

Ownership & borrowing

- Creating a variable grants ownership
- Assignment transfers ownership
- "Borrowing" allows a section of code to use a variable without taking ownership. At one time, you can have **either**
  - 1 *mutable* borrow, **OR**
  - Limitless immutable borrows

Ownership Transfer Example

```
fn f(x: Box<i32>) {
    println!("{}", x);
}
fn main() {
    let a = Box::new(42_i32);
    println!("{}", a);
    f(a);
}
```

Error

```
fn f(x: Box<i32>) {
    println!("{}", x);
}
fn main() {
    let a = Box::new(42_i32);
    println!("{}", a);
    f(a);
    println!("{}", a);
}
```

Fixed: `f` Modified to Borrow

```
fn f(x: &Box<i32>) {
    println!("{}", x);
}
fn main() {
    let a = Box::new(42_i32);
    println!("{}", a);
    f(&a);
    println!("{}", a);
}
```

Complex number example

(in-class)

Typechecking in Rust

- Is Rust statically or dynamically typed?
- Sample code:

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

This Code Won't Compile

```
fn main() {
    let s = "hello";
    let x = s + 42;
    println!("{}", x);
}
```

Compilation Error

```
$ rustc typing.rs
error[E0369]: binary operation `+`
cannot be applied to type `&str`
 --> typing.rs:3:15
  |
3 |     let x = s + 42;
  |             - ^ -- {integer}
  |             |
  |             &str
  |
```

Type Inference

- Rust *can* have type annotations:

```
let x: i32 = 99;
```

- For local variables, types are optional.
- For functions, types are mandatory.

Function Type Annotations

```
fn double(n: i32) -> i32 {
    // No semicolon on the next line.
    n*2
}

// No arguments, no return value.
fn main() {
    let x = 45;
    let y = double(3);
    println!("{}", x+y);
}
```

Rust documentation

Rust programming language "book"

https://doc.rust-lang.org/nightly/book/

Rust by Example

http://rustbyexample.com/

Lab: Implement Quicksort

- Use sort0.rs, sort1.rs, and sort2.rs for reference (available online)