Reflecting on Rust

Ryan Eberhardt and Armin Namavari
June 4, 2020

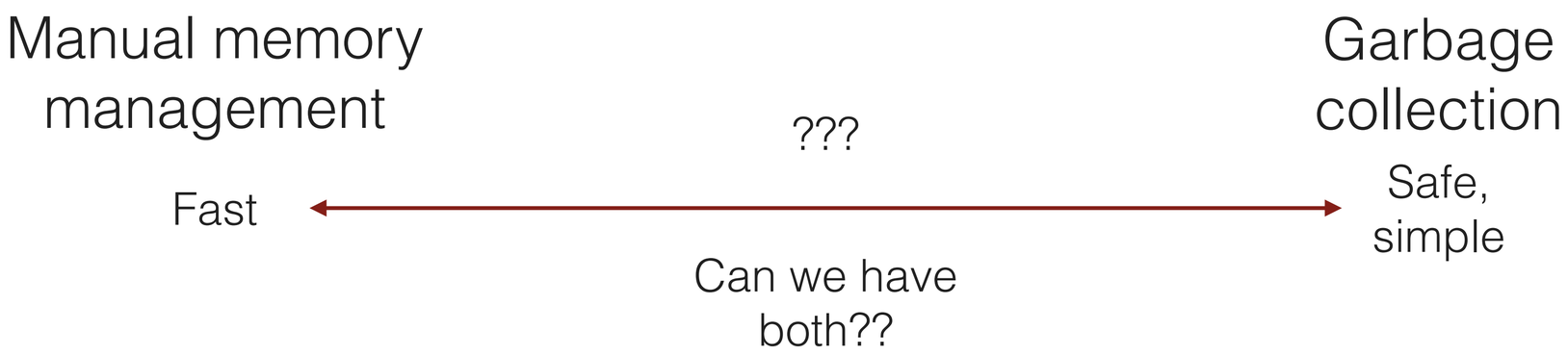
Logistics

- This is our last lecture together 😢
  - We are so, so proud of everything you have learned this quarter, and we hope you are too!
- Next Tuesday, will have a guest lecture from Sergio Benitez
  - Please come!
- Please fill out Week 8 survey if you haven’t already (link in Slack)
- Project 2 due Wednesday, but we will accept it until Saturday with no penalty, and we are happy to give further accommodations if you aren’t graduating

Why are we here?

- What was the point of this class?
  - Learn about common safety issues in designing/building systems
  - Learn about how people are responding to those problems
  - Get first-hand experience with those responses
- This lecture will focus on Rust in particular
  - Why do we care about Rust?
  - Why is Rust effective and what can we learn from its design?
  - How can we work to write safer C++?

Why do we care about Rust?



- Manual memory management has led to countless security vulnerabilities (70% of Chrome security bugs are memory safety issues)
- Garbage collection introduces unpredictable latency and unacceptable overhead for many applications
- Is there some way we get the benefits of both approaches?
  - Rust seems to do this for us! In this lecture, we'll look at *why* it works, and how we might be able to apply lessons from Rust to other languages

Why is Rust effective?

Why is Rust effective?

- Using a strong type system
- Safety by default

Type systems

Why is C frustrating?

*Imagine you are a construction worker, and your boss tells you to connect the gas pipe in the basement to the street's gas main. You go downstairs, and find that there's a glitch; this house doesn't *have* a basement. Perhaps you decide to do nothing, or perhaps you decide to whimsically interpret your instruction by attaching the gas main to some other nearby fixture, perhaps the neighbor's air intake. Either way, suppose you report back to your boss that you're done.*

KWABOOM! *When the dust settles from the explosion, you'd be guilty of criminal negligence.*

Yet this is exactly what happens in many computer languages. In C/C++, the programmer (boss) can write `"house"[-1] * 37`*. It's not clear what was intended, but clearly some mistake has been made. It would certainly be possible for the language (the worker) to report it, but what does C/C++ do?*

- *It finds some non-intuitive interpretation of* `"house"[-1]` *(one which may vary each time the program runs!, and which can't be predicted by the programmer),*
- *then it grabs a series of bits from some place dictated by the wacky interpretation,*
- *it blithely assumes that these bits are meant to be a number (not even a character),*
- *it multiplies that practically-random number by 37, and*
- *then reports the result, all without any hint of a problem.*

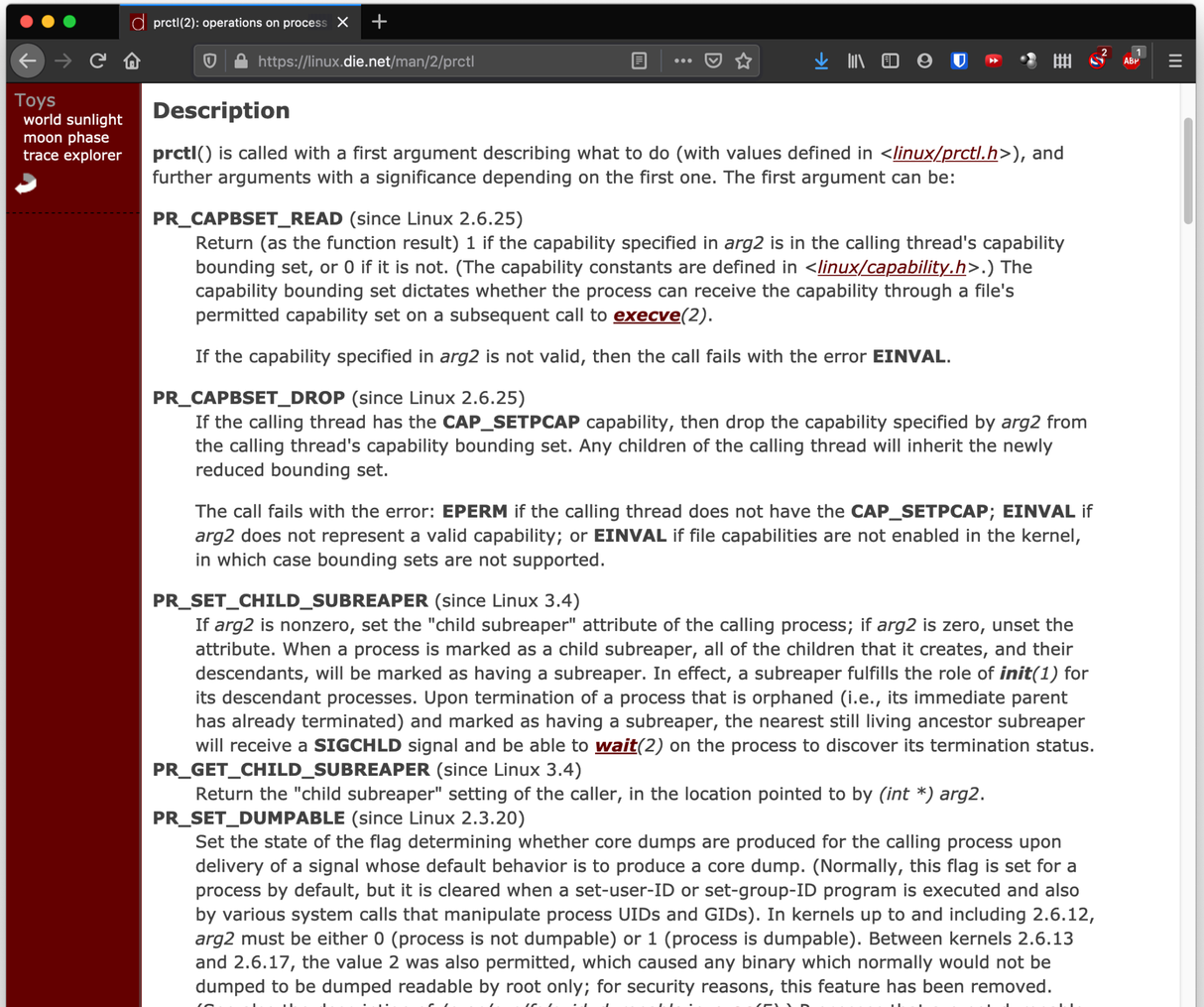
https://www.radford.edu/ibarland/Manifestoes/whyC++isBad.shtml

Why is C frustrating?

```
typedef struct vector {
    size_t length;
    size_t capacity;
    size_t elem_size;
    void *data;
} vector;
```

Why is C frustrating?

```
int prctl(int option, unsigned long arg2, unsigned long arg3, unsigned long arg4, unsigned long arg5);
```



Why is C frustrating?

16 bytes

```
struct sockaddr {
    unsigned short   sa_family;
    char             sa_data[14];
};
```

16 bytes

```
struct sockaddr_in {
    short int          sin_family;
    unsigned short int sin_port;
    struct in_addr     sin_addr;
    unsigned char      sin_zero[8];
};
```

16 bytes

```
struct sockaddr_in6 {
    sa_family_t     sin6_family;   /* AF_INET6 */
    in_port_t       sin6_port;     /* port number */
    uint32_t        sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr;     /* IPv6 address */
    uint32_t        sin6_scope_id; /* Scope ID (new in 2.4) */
};
```

Why is C frustrating?

Why is C frustrating?

- C is designed and used by brilliant people. What is the reason for the madness?
- C is tightly coupled to the machine executing the code
  - Machines don't have notions of vectors, generic types, or polymorphism
  - Prctl is the way it is because of how the syscall call/return mechanism passes arguments through registers, not because it's convenient for anyone to think about in that way
  - C code is often the way it is because it maps well to how computers work
- Side note: When introduced, C was actually a big advancement in that it targeted an abstract machine model, so C programmers don't need to think about the specifics of the processors they are running on
  - However, it is tightly coupled to that abstract machine model

Rust’s perspective

- By contrast, Rust is designed to match how programmers think
- Want a vector containing strings? Just create one and add the desired strings!
- No need to think about allocating memory, casting void* pointers appropriately, passing the correct number of bytes each element occupies, or freeing memory

Type systems

- Types are the *unit of dialogue* in a language
  - When you talk in a language, what do you talk about?
- A compiler uses types to figure out:
  - What are you trying to say?
  - Does what you’re saying make sense?

Type systems

- C's type system is oriented around primitives, structs, and pointers
  - When you write `"house"[-1] * 37`, the compiler figures out what you're saying in terms of pointers and verifies that it makes sense
- C has a very small language surface, which is nice
- However, because of the limited constructs it can express, you must do a lot of work to translate ideas into C code
- Similarly, when reading C code, it's difficult to build a mental model of what the authors were thinking when writing the code
  - E.g. when reading a codebase, it may take a while to figure out where the authors intended for some memory to be freed
  - Consequently, the compiler has very little understanding of the *intent* of a programmer

Type systems

- Rust's type system tries to encode high-level *ideas* into the language
  - When you write code, there is a notion of ownership *in the code*
  - When you have a vector, there is a notion of the type of elements in the vector
  - When you have a type, there is a notion of whether it's safe to share values of that type between threads (Sync/Send)
- Because we express high-level ideas in the language, the compiler can understand what we're trying to do, and can warn us when we do something dumb
- By the same token, other programmers can more easily understand what is going on from reading your code
- This is about much more than just memory safety! This is why Rust was designed with powerful macros: We can extend the language with new *ideas* that can be expressed and checked at compile time

Takeaways for systems design

- You may never design a programming language, but you will very likely need to design or implement an API
  - E.g. creating a software library, or implementing a service over HTTP
- Good API design is very hard and has a lot of overlap with good language design
- Design from the client's perspective with the implementation in mind, not the other way around
  - If you expose a complex interface, *every* client will need to deal with that complexity
  - If you expose a simple interface with a complex implementation, it may be hard to build, but you can do it once and move on
  - It's *so* tempting to build an API that directly maps to the implementation, since that's what you understand as an implementor. But take extra time to consider what "types" a client thinks in terms of!

Safety by default

Safety by default

- Rust safety features are a core part of the language
- You can opt-out using `unsafe`, but it's discouraged and sometimes more trouble than it's worth
- C++ has many safety features, but they are opt-in
  - Worse, even the "safe" STL classes have unsafe parts that are easy to accidentally use

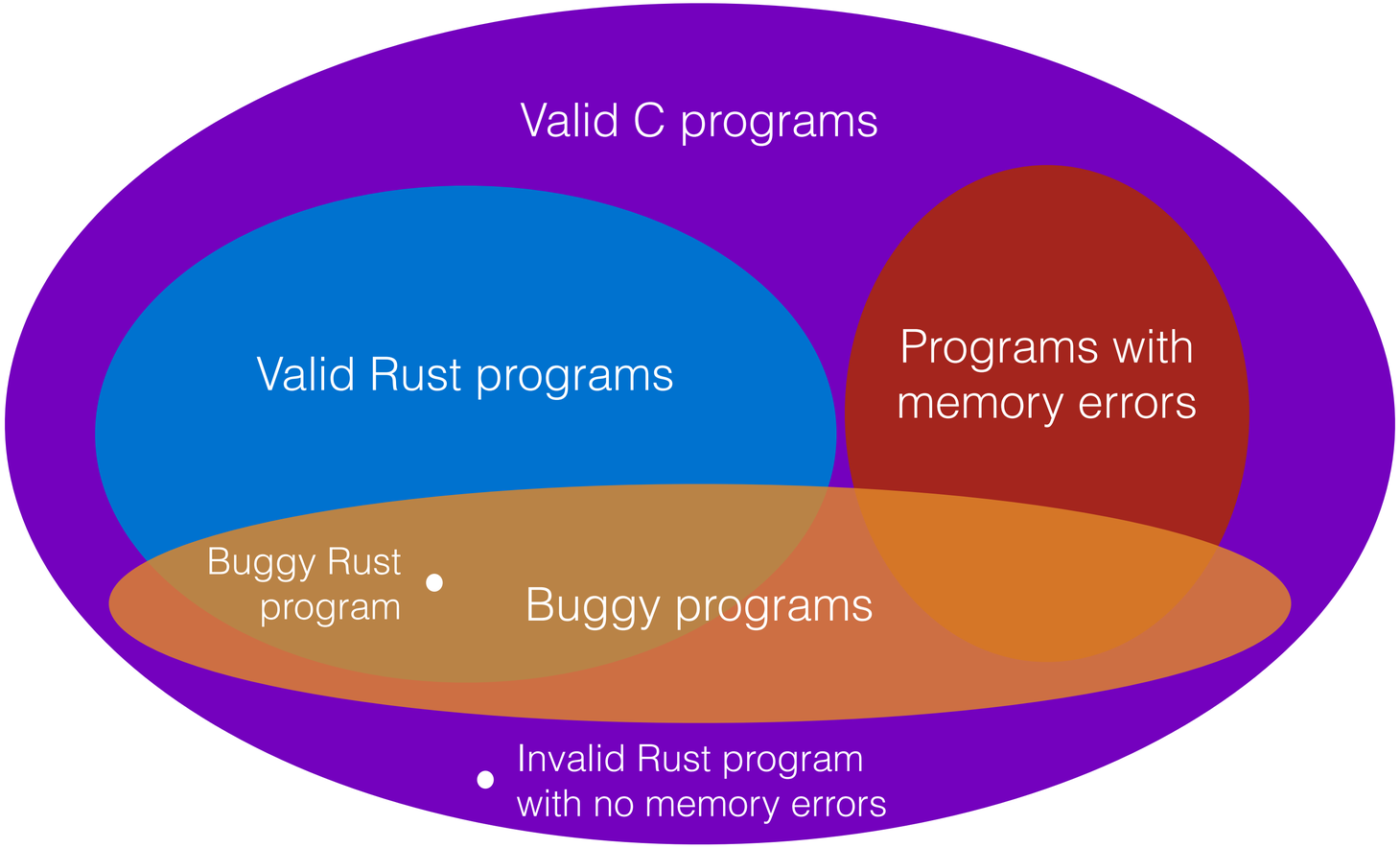
Takeaways for systems design

- Design systems that are *harder* to misuse than they are to use correctly
- As we saw in the information security lecture, sometimes safety by default isn’t good enough (e.g. if disabling safety features makes life significantly easier for a user)

Rust is not a panacea!

Rust is not a panacea!

- I think you've learned this from your assignments, but it's worth stating explicitly



Rust is not a panacea!

- I think you've learned this from your assignments, but it's worth stating explicitly
- Additionally, some Rust code uses `unsafe`. If you use libraries that have incorrect usage of `unsafe`, your code will also be susceptible to vulnerabilities
- We still have a ways to go in making the language fast and usable

Safety in C++

You still need to learn C/C++

- C and C++ suck, but in many cases, we don’t have a choice
- There is lots of existing code that must be supported
- Rewriting projects introduces bugs (and sometimes reintroduces old, long-fixed bugs)
  - I have never heard of a real-life project where this wasn’t the case
  - Mozilla’s experience rewriting Firefox CSS engine in Rust
- People are still writing in Fortran… There’s no way we’re ditching C/C++ any time in the near future

Applying Rust to C++

- In many ways, Rust codifies best practices that you should be doing in other languages anyways
- Writing good code may not be as natural as it is in Rust, but many of the same ideas can be applied
- There is a ton of material in the next few slides. We don’t expect you to understand it all; we just want you to know it exists so that you can look it up when you recognize a need to use it

Allocating/freeing memory

- The traditional (and error-prone) way to initialize objects is to have functions like `vec_init` that allocate memory and `vec_destroy` that free associated resources
- RAII is a horrible name for the practice of acquiring resources (e.g. allocating memory) in the constructor of an object and freeing the memory in the destructor
  - The destructor is called when the object goes out of scope
  - No memory leaks or double frees!
  - Most C++ STL classes are RAII (e.g. `vector` manages the memory allocations for you)
  - Applies to more than just memory (e.g. `lock_guard` releases the lock when it goes out of scope)

Ownership

- When RAII is used, we can talk about ownership similar to Rust. A variable “owns” the value inside
- The = operator copies by default
  - You may have encountered this in the form of unexpected performance hits
- You can use std::move() to indicate you would like to move instead of copying
  - E.g. `string val2 = move(val1);`
  - Note that the compiler will *not* complain if you subsequently use `val1`. Use linters like clang-tidy to catch mistakes like this
- You can “borrow” references to a value of type T by assigning to variables/parameters of type `&T`
  - Not as explicit as Rust about when references are being borrowed, but the same thing is happening
  - Beware: Unlike Rust, there is no borrow checker doing lifetime analysis, so dangling pointers are still a thing. 36.1% of Chrome high-severity security bugs (52% of memory-related security bugs) caused by use-after-free!

Smart pointers

- Similar to Rust, C++ objects are stack-allocated by default
- Heap allocation can be done with `new` and `delete`, but this is error-prone
- Smart pointers are wrapper objects that automatically manage memory allocations for you
- std::unique_ptr is like Box: single owner, ownership can be transferred (can also borrow references, as long as owner lives long enough)
  - ```
    unique_ptr<string> s = make_unique<string>("hello world");
    cout << *s << endl;
    unique_ptr<string> s2 = move(s);
    cout << *s2 << endl;
    ```
    (cplayground)
    ```

Smart pointers

- std::shared_ptr is like Rc: multiple owners (via reference counting)
  - ```
    shared_ptr<string> s = make_shared<string>("hello world");
    cout << *s << endl;
    shared_ptr<string> s2 = s; // makes a copy, inc refcount
    cout << *s2 << endl;
    ```
    (cplayground)
    ```

Arrays/vectors

- `std::vector` is like `Vec` (allocates a growable vector on the heap), except the `[]` operator *does not do bounds checks!* Use the `.at(i)` method to get an element with bounds checking
- `std::array` encapsulates a C array with its length
  - Never need to worry about remembering to pass the proper length
  - Can use the `.at(i)` method to do bounds checking
  - Automatically frees the array when it goes out of scope
- `std::span` is like a slice (provides a view into a segment of a vector or array)

Avoiding null dereferences

- C++17 introduced `std::optional`, which is like Option
  - An `optional<T>` can either be `std::nullopt` or a value of type `T`
  - Example: https://en.cppreference.com/w/cpp/utility/optional#Example
  - Use `.value()` to get the value inside an `optional` (an exception is thrown if the `optional` is empty)
  - Unfortunately, `optional` also defines the `*` and `->` operators to get the value inside, which return uninitialized values if the optional is empty :-/
- C++20 introduces `map`, `and_then`, and `or_else` functions like ones you may have used in Rust
- Be aware that `nullptr` is widely used in C++ code, and `optional` is mostly used in places where `nullptr` doesn't work well
  - Pretty good blog post from Microsoft here

Error handling

- There is no consensus on how to do error handling in C++
- Exceptions only work if *all* of your code is RAII
  - Imagine function A has a try/catch that calls function B, which calls function C, which calls some other functions
  - One of the functions called by function C throws an unexpected exception
  - Function A catches the exception, but function B is “skipped” and never has a chance to free the resources
  - In general, exceptions also complicate control flow
- There is a Result-like type being debated, but it hasn’t made it into the standard library yet
- A whole lot of code uses `int` return values to indicate errors. This has its own problems
  - So many bugs caused by forgetting to check the return value, or from doing it incorrectly
  - Pain in the butt to do everywhere

Error handling

- Google style guide forbids exceptions: https://google.github.io/styleguide/cppguide.html#Exceptions
- Mozilla also forbids exceptions in Firefox:
  - https://firefox-source-docs.mozilla.org/code-quality/coding-style/using_cxx_in_firefox_code.html
  - https://firefox-source-docs.mozilla.org/code-quality/coding-style/coding_style_cpp.html#error-handling (good read on error handling in general)
- Microsoft doesn't have a public, general style guide, but their language reference encourages using exceptions: https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp?view=vs-2019

Multithreading

- Use RAII wrappers for synchronization primitives whenever possible (e.g. lock_guard)
- Use higher-level communication abstractions when applicable (e.g. channels)

Use code quality tools!

- These language features help a *lot*, but they don't even come close to addressing C++'s safety issues
  - The language features only help if you use them
  - Trying to use these features in an existing codebase has the same problem that switching to Rust does: you still have a lot of legacy code using a lot of antipatterns
- Since C++ does not ensure safety by default, you should use tools to get better assurances about your code
- These tools typically fall into two categories: static analysis (done on the source code) and dynamic analysis (done on a running program)

Static analysis

- Limited in what it can say about a program (for fully general programs, you don’t really know what the program will do until you execute it)
  - Can’t really follow the control flow of a program at a high level
  - Often simply analyze code at a function level
  - Often define a set of rules for safe behavior. Code that violates those rules might not be unsafe, but the static analysis tools will give you errors anyways. (Better safe than sorry)

Built-in static analysis

- The compiler already does some amount of static analysis and can be configured to give you different warnings/errors. You can pass various -W flags to enable certain warnings
- -Wall does *not* enable all warnings!! It enables “all the warnings about constructions that some users consider questionable, and that are easy to avoid” (GCC manual)
  - 🙄
- -Wextra adds some extra warning flags (but not all of them)
  - 🙄
- It’s not uncommon to end up with compiler invocations like this: -Wall -Werror -Wextra -Wpedantic -Wvla -Wextra-semi -Wnull-dereference -Wswitch-enum -fvar-tracking-assignments -Wduplicated-cond -Wduplicated-branches -rdynamic -Wsuggest-override
- https://github.com/lefticus/cppbestpractices/blob/master/02-Use_the_Tools_Available.md#compilers
- https://kristerw.blogspot.com/2017/09/useful-gcc-warning-options-not-enabled.html

Linting

- A “linter” enforces code style rules
  - Bad style (e.g. deeeeeeeply nested code) obscures logic and makes it much harder to spot bugs
  - Linters also commonly do some basic static analysis to spot obvious errors (e.g. calling unsafe functions like `strcpy`, or using a value after it has been moved out of a variable)
- clang-tidy is one of the most powerful and commonly used linters

Higher-level static analysis

- More powerful static analyzers attempt to build a graph of the flow of data in a program, in order to spot buffer overflows, null pointers, integer overflows, and other common errors
- Pretty good open source project: Cppcheck
- This is an area of active research!
  - E.g. symbolic execution can theoretically audit all control flow paths a program can take, but it’s currently too slow to be practical for large programs

Dynamic analysis

- Dynamic analysis involves inspecting the behavior of a running program
  - Not comprehensive: can only complain about behavior that it actually observes (if a program only does something bad 0.00001% of the time, dynamic analysis may not catch it)
  - However, not many false positives: observed problems are usually real problems

Sanitizers

- Sanitizers are LLVM compiler plugins that inject extra code to keep track of what your program is doing and record dangerous behavior
- AddressSanitizer (detects accesses to invalid addresses), LeakSanitizer (detects memory leaks), MemorySanitizer (detects use of uninitialized memory), ThreadSanitizer (detects data races and deadlocks), and more
- Example: ThreadSanitizer tracks accesses to data and locks. If two threads read/write without first acquiring a lock, ThreadSanitizer will log an error
- This is similar to what valgrind does, except the instrumentation is generated by the compiler instead of being injected just-in-time while the program is executing

Fuzzers

- Fuzzers are programs that repeatedly provide semi-random input to your program until it crashes
- AFL and libFuzzer are the two most common fuzzers
  - The former is very useful for end-to-end fuzzing programs that take input via stdin
  - The latter is built into LLVM and fuzzes individual functions. Faster than AFL and useful when the code in question doesn’t take input via stdin

Bonus: Test, test, test!

- Automated tests are absolutely critical to any large project
- This is true for any project in any language
- Any time there is a bug, add a test case to protect against regressions

Summary: Using C++

- Use safety features when you can
- Often, you may not be able to use safety features. Even when you do, it's easy to screw up
- As a result, it's important to set up a development environment with automated code quality tests
  - Not too hard to set up infrastructure that runs a linter, automated test suite, and sanitizer checks on each commit
- Side note: Automatic code checking is an active area of research! If you're interested, we can connect you to people in the CS department that work on these sorts of things

Closing remarks

Closing remarks

- Thank you for taking this class! It has been such a pleasure having you, and we hope you've enjoyed it
- You all have come so far!

- Please come next Tuesday for our guest speaker — it should be a really interesting talk