Lessons Learned

Ryan Eberhardt and Julio Ballista
June 3, 2021

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!
- Project 2 due today. Please let us know if we can help!

Lessons learned: Safety in Systems Programming

- Make bad things hard/impossible to happen
  - Use a strong type system to your advantage
  - Design with a human user in mind
- Institute processes to avoid making the same mistakes again

Use a strong type system to your advantage

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.*

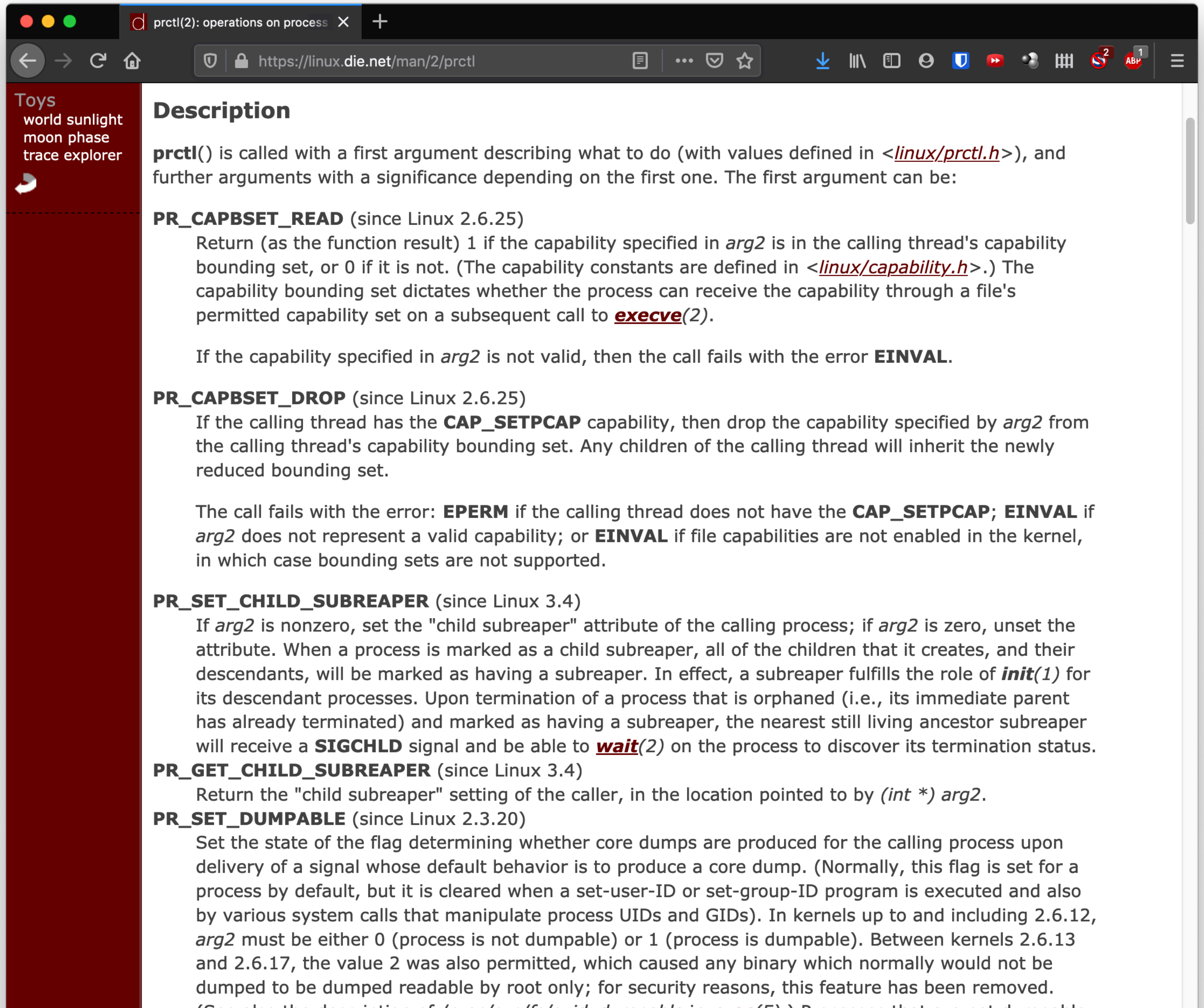
Yet this is exactly what happens in many computer languages. In C/C++, the programmer (boss) can write `"house"[-1] * 37`*. It's not clear what was intended, but clearly some mistake has been made. It would certainly be possible for the language (the worker) to report it, but what does C/C++ do?*

- *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?

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



Description

prctl() is called with a first argument describing what to do (with values defined in <*linux/prctl.h*>), and further arguments with a significance depending on the first one. The first argument can be:

PR_CAPBSET_READ (since Linux 2.6.25)
Return (as the function result) 1 if the capability specified in *arg2* is in the calling thread's capability bounding set, or 0 if it is not. (The capability constants are defined in <*linux/capability.h*>.) The capability bounding set dictates whether the process can receive the capability through a file's permitted capability set on a subsequent call to ***execve***(2).

If the capability specified in *arg2* is not valid, then the call fails with the error **EINVAL**.

PR_CAPBSET_DROP (since Linux 2.6.25)
If the calling thread has the **CAP_SETPCAP** capability, then drop the capability specified by *arg2* from the calling thread's capability bounding set. Any children of the calling thread will inherit the newly reduced bounding set.

The call fails with the error: **EPERM** if the calling thread does not have the **CAP_SETPCAP**; **EINVAL** if *arg2* does not represent a valid capability; or **EINVAL** if file capabilities are not enabled in the kernel, in which case bounding sets are not supported.

PR_SET_CHILD_SUBREAPER (since Linux 3.4)
If *arg2* is nonzero, set the "child subreaper" attribute of the calling process; if *arg2* is zero, unset the attribute. When a process is marked as a child subreaper, all of the children that it creates, and their descendants, will be marked as having a subreaper. In effect, a subreaper fulfills the role of ***init***(1) for its descendant processes. Upon termination of a process that is orphaned (i.e., its immediate parent has already terminated) and marked as having a subreaper, the nearest still living ancestor subreaper will receive a **SIGCHLD** signal and be able to ***wait***(2) on the process to discover its termination status.
PR_GET_CHILD_SUBREAPER (since Linux 3.4)
Return the "child subreaper" setting of the caller, in the location pointed to by *(int *) arg2*.
PR_SET_DUMPABLE (since Linux 2.3.20)
Set the state of the flag determining whether core dumps are produced for the calling process upon delivery of a signal whose default behavior is to produce a core dump. (Normally, this flag is set for a process by default, but it is cleared when a set-user-ID or set-group-ID program is executed and also by various system calls that manipulate process UIDs and GIDs). In kernels up to and including 2.6.12, *arg2* must be either 0 (process is not dumpable) or 1 (process is dumpable). Between kernels 2.6.13 and 2.6.17, the value 2 was also permitted, which caused any binary which normally would not be dumped to be dumped readable by root only; for security reasons, this feature has been removed.

Why is C frustrating?

Why is C frustrating?

- 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

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?

C’s type system

- 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

Richer type systems: Just say what you mean!

- Richer type systems allow you to encode higher-level *ideas* into the language
    - You can just say what you mean — in the code!
- Comments are great, but…
    - They can become stale (comments no longer match the code)
    - Often some construct will be used in multiple places, but you only comment in one place (e.g. documenting int return value for network functions, or documenting when you're supposed to hold a particular lock)
- If possible, find ways to express what you're saying directly in the code
    - 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

Example: Memory

```
/* Get status of the virtual port (ex. tunnel, patch).
 *
 * Returns '0' if 'port' is not a virtual port or has no errors.
 * Otherwise, stores the error string in '*errp' and returns positive errno
 * value. The caller is responsible for freeing '*errp' (with free()).
 *
 * This function may be a null pointer if the ofproto implementation does
 * not support any virtual ports or their states.
 */
int (*vport_get_status)(const struct ofport *port, char **errp);
```

- Instead, make this explicit
  - Function takes ownership and will free the memory:
    fn take_ownership(var: MyType) {}
  - Function borrows a reference, caller needs to free:
    fn borrow(var: &MyType) {}
- Compiler can sanity check memory usage
- Programmers don't need to remember all the comments. Can tell what's happening from the function signature

Example: Error handling

- In what ways can a function fail?
  - Sometimes document this in a comment or manpage
    - Often this documentation becomes stale as the code evolves
  - Can you make this part of the code itself by putting the information in the function signature?
- Related: use enums to represent a range of possibilities, instead of picking arbitrary numbers or strings

```c
static inline int l2t_send(struct t3cdev *dev, struct sk_buff *skb,
               struct l2t_entry *e)
{
    ...
}
```

Lecture 5, slide 16

```rust
enum ConnectionState {
    Ok,
    Error(ErrorInfo),
    Congested,
}

fn l2t_send(...) -> ConnectionState {
    ...
}
```

Example: File descriptors

- ```
  if (close(fds[1] == -1)) {
      printf("Error closing!");
  }
  ```
  - This compiles perfectly fine because `close` takes an integer, and booleans can be implicitly casted to integers
- If file descriptors were their own type, this couldn’t happen
  ```

Example: Authentication/Authorization

- Rocket is a web server framework for Rust. Code looks like this:

```
#![feature(proc_macro_hygiene, decl_macro)]

#[macro_use] extern crate rocket;

#[get("/")]
fn index() -> &'static str {
    "Hello, world!"
}

fn main() {
    rocket::ignite().mount("/", routes![index]).launch();
}
```

Example: Authentication/Authorization

```
#[get("/getAllUserData")]
fn super_secret_endpoint(admin: Admin) -> String {
    database.log_access(&admin);
    load_user_data(admin)
}
```

- We can define “request guards,” which are values that Rocket extracts from requests and passes to our request handler functions
- A request guard type implements a from_request function; Rocket passes the request, and gets a value of that type to pass to our handlers
- These types can be added as parameters to other helper functions in the codebase, e.g. database functions. This makes the authorization policy *part of the code*, i.e. it is impossible to load all user data without having an admin making the request

Design with a human user in mind

Design for humans

- UI/UX people spend a lot of time thinking about how to design interfaces that are easy to use and hard to misuse. As systems people, we really need to do the same!
- I also think we suffer from what I will call “macho gatekeeping systems culture”:
    - Systems programming is hard, so you better be top-notch and super smart. If you mess up, that’s your fault for not being careful or smart enough

Design for humans

- Systems programming *is* hard!!
  - There is a tremendous amount of complexity involved
  - Sometimes we need to do things in complex ways to match what the underlying hardware/systems are doing, or else we'd never get good performance
- But that's no excuse to not *try* to find ways to our systems as safe and easy to use as possible. There are always good abstractions hiding somewhere that are easy to understand *and* map well to how the machine works
- 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

Example: Most of Rust!

- Many of the concepts in Rust are also present in C++
- But we taught this class in Rust because the compiler forces you to do things the safe way, whereas in C++, every “safety feature” has a dozen ways to mess up

Example: Mutexes

- Keeping track of when to lock/unlock is hard
- The “monitor pattern” (putting data *inside* the lock) makes it impossible to have a data race!

Example: Channels

- Again, concurrency with shared memory is really hard
- As programmers, it might be easier to think in terms of separate “actors” that exchange messages, instead of fighting over shared memory
- At first, this seems to be a model not worth pursuing, because message passing can be very expensive due to memory copy overhead
- But there are clever ways to make this work! Shallow copies + transfer of ownership provides the illusion of full message passing with no shared memory, even though our threads do share memory under the hood

Example: Asynchronous programming

- Programming with bare event-driven, callback-oriented nonblocking I/O facilities matches the machine model well, but is terrible for programmers
  - From David Mazieres' blog post on C++ coroutines:

```
void cmd_rcpt (str cmd, str arg);
void cmd_rcpt_0 (str cmd, str arg, int, in_addr *, int);
void cmd_rcpt_2 (str addr, int err);
void cmd_rcpt_3 (str addr, str errmsg);
void cmd_rcpt_4 (str addr, str errmsg, int local);
void cmd_rcpt_5 (str addr, str errmsg, str err);
void cmd_rcpt_6 (str addr, str err);
```

- But with the right abstractions and some compiler ingenuity, it's possible to provide ways to write normal-looking code that compiles down to this!
  - Still some rough edges… Still room for improvement!

Non-Example: Database security

- For a long time, databases were *insecure* by default
- Now, Elasticsearch is technically secure by default in that it doesn't accept remote connections…
  - But this isn't very useful, so it's just inviting inexperienced sysadmins to open up the database to external connections without adding more security
  - Need more innovative solutions in this area!

Institute processes that prevent mistakes and promote improvement

Process improvement

- Writing good code is hard.
  - Even with all the previous suggestions in place, you will still have bugs
  - On top of that, best practices are constantly changing. Something might seem like a sensible idea today, but we might realize it's a bad idea tomorrow (e.g. strcpy)
- More than just writing good code, we need to institute *processes* that will catch problems early and prevent regressions

Typical process

- Typical process:
  - Implement feature/fix/whatever
  - Submit for review
  - Merge change into codebase
  - Release
  - Gather metrics, watch for signs that something has gone wrong
- Continuous integration (CI):
  - Every time you push, a CI server can run some tests/tools to catch mistakes early
  - This can happen at every stage! Issues show up before review
  - What should the CI server do to check your code?

Static analysis

- Linters help identify potentially problematic patterns in code
  - E.g. don't call strcpy!
- More sophisticated static analysis can find deeper problems
  - In this class, we focused a lot on memory safety, but static analysis can be applied to find any specific problem
  - CodeQL is a sophisticated general-purpose static analyzer for many different languages. Can catch improper usage of libraries, bad cryptography, insecure networking practices, much more!
- False positives and false negatives can both be issues, but this is essential to integrate into your process

Dynamic analysis

- Sanitizers: add some code to the program that observes what it does and raises an alarm if it does something bad
  - Examples: use of uninitialized memory, buffer overflow, data race, etc
  - Not limited to memory issues. If you can clearly define the bad behavior you're looking for, you can build a dynamic analyzer to catch it!
- Fuzzers: generate semi-random inputs and feed them to a program, exploring its possible behaviors, until it does something bad
  - Traditionally used to find memory errors, but the technique is so simple and could be used for so much more
- Many other creative kinds of dynamic analysis are possible: e.g. if you're implementing a website, when you change something, can use a system to screenshot parts of the website on different browsers and do a visual diff of what looks different, in order to catch browser-specific bugs

Automated testing

- It's crucial to have tests that exercise the different parts of your codebase
  - May not seem important at the beginning, but as the codebase grows, it becomes harder to refactor anything without worrying about breaking everything
  - Manual testing is important but takes way too much time to do regularly (e.g. on every commit) on a sophisticated codebase
- Also important: regression testing
  - Any time there is a notable bug that slipped past your existing tests, you should be adding a new test for whatever broke to make sure it does not happen again

Recap: Lessons learned

- Make bad things hard/impossible to happen
  - Use a strong type system to your advantage
  - Design with a human user in mind
- Institute processes to avoid making the same mistakes again

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!

- Have a wonderful summer, and please keep in touch! We'd love to hear what you do next and where you end up.

Extra slides: 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)

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

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