Typology of Programming Languages

Memory Safety

May 2025

Section 1

Memory Errors

Exercise

List memory errors in C with examples.

Exercise

List memory errors in C with examples.

- Heap/stack overflow
- Null-ptr derefs
- Dangling pointers
  - Memory leaks
  - Use-after-free
  - Invalid stack pointers
- ...

Exercise

List memory errors in C with examples.

- Heap/stack overflow
- Null-ptr derefs
- Dangling pointers
  - Memory leaks
  - Use-after-free
  - Invalid stack pointers
- ...

Exercise

How problematic are these?

Memory Vulnerability

> *Out of the 58 in-the-wild 0-days for the year, 39, or 67% were memory corruption vulnerabilities. Memory corruption vulnerabilities have been the standard for attacking software for the last few decades and it's still how attackers are having success. Out of these memory corruption vulnerabilities, the majority also stuck with very popular and well-known bug classes:*
>
> - *17 use-after-free*
> - *6 out-of-bounds read & write*
> - *4 buffer overflow*
> - *4 integer overflow*
>
> *– Google Project Zero - A Year in Review of 0-days Used In-the-Wild in 2021*

Memory Vulnerability

> *Out of the 58 in-the-wild 0-days for the year, 39, or 67% were memory corruption vulnerabilities. Memory corruption vulnerabilities have been the standard for attacking software for the last few decades and it's still how attackers are having success. Out of these memory corruption vulnerabilities, the majority also stuck with very popular and well-known bug classes:*
>
> - *17 use-after-free*
> - *6 out-of-bounds read & write*
> - *4 buffer overflow*
> - *4 integer overflow*
>
> *– Google Project Zero - A Year in Review of 0-days Used In-the-Wild in 2021*

Let's try to fix C!

Heap/stack over/underflow

```
// C
#define LEN 10
int main(void) {
  int *arr1 = (int*) calloc(LEN, sizeof(int));
  int arr2[LEN] = { 0 };

  return arr1[LEN] + arr2[-1];
}
```

Compiles, runs, crashes.

How do we modify C to prevent this? (if possible, statically)

Heap/stack over/underflow

```c
// C
#define LEN 10
int main(void) {
  int *arr1 = (int*) calloc(LEN, sizeof(int));
  int arr2[LEN] = { 0 };

  return arr1[LEN] + arr2[-1];
}
```

Compiles, runs, crashes.

How do we modify C to prevent this? (if possible, statically)

- Hard to do statically (possible with dependent types).
- Easy to check a runtime with bounds-checking.
  - Small overhead, but no undefined behavior.
  - Some compilers can optimize away bounds-check (when unnecessary).
    - And we end writing bounds check by hand in C anyway…

The Billion Dollar Mistake

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. 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.*
>
> — *C.A.R. Hoare*

The Billion Dollar Mistake

```c
// C
int main(void) {
  int *n = NULL;
  return *n;
}
```

The Billion Dollar Mistake

```c
// C
int main(void) {
  int *n = NULL;
  return *n;
}
```

How do we modify C to prevent this? (if possible, statically)

The Billion Dollar Mistake

```c
// C
int main(void) {
  int *n = NULL;
  return *n;
}
```

How do we modify C to prevent this? (if possible, statically)

Easy: use option types or nullable types.

```rust
// Rust
fn main() {
  let x : Option<i64> = None;
  let x : i64 = x; // type-error
}
```

Option types are either `None` or `Some(value)`. Often implemented in the runtime as a variant type.

```kotlin
// Kotlin
fun main() {
  val n : Int? = 42
  val n2 : Int = n // type-error
}
```

Nullable types are either `null` or `value`. `null` cannot be used with a non-nullable type.

Dangling pointers

```c
// C
#define LEN 10
int* get_len() {
  int len = LEN;
  return &len; // pointer to stack local
}

int main(void) {
  int* arr1 = (int*) calloc(LEN, sizeof(int));
  int* arr2 = (int*) calloc(LEN, sizeof(int));

  // [...] some code
  free(arr2); // oops
  // [...] some other code

  for (int i = 0; i < *get_len(); i++)
    arr1[i] = arr2[i]; // use after free

  return arr1[0]; // memory leak
}
```

Dangling pointers

How do we modify C to prevent this? (if possible, statically)

Dangling pointers

How do we modify C to prevent this? (if possible, statically)

- Remove all pointers?
  - A bit drastic.
  - Might work, used in some languages: you never have to handle pointers in Python or OCaml.
  - The compiler would still have to use them under the hood.
    - This means you would likely need garbage collecting to handle them.

Dangling pointers

How do we modify C to prevent this? (if possible, statically)

- Remove all pointers?
  - A bit drastic.
  - Might work, used in some languages: you never have to handle pointers in Python or OCaml.
  - The compiler would still have to use them under the hood.
    - This means you would likely need garbage collecting to handle them.

- GC's are fine in most cases.
- If you do systems programming/high-performance computing, you might want to keep pointers around.

Section 2

RAII

Managing resources

Some resources need to be released:

- Dynamic (heap) allocations must be freed.
- Opened files must be closed.
- Locked mutex must be unlocked.
- ...

These have an *indefinite* lifetime. They have a start, and need to end *at some point*.

Forcing them into a *definite* lifetime would free us from having to free them.

Ressource Acquisition is Initialization

Bind resources to a stack object. When it is destructed, free the associate ressource.

```
// C++
template <typename T>
struct RAII {
  T* data;
  RAII(T* data) : data(data) {}
  ~RAII() {
    delete data;
  }
};

int main() {
  auto the_answer =
    RAII(new int(42));
  std::cout
    << *the_answer.data
    << "is the answer\n";
}
```

- `delete` is automatically called for us by the destructor!
- Can be improved with fancy C++ to forward args to `T`'s constructor, dereference with `operator*`, etc.

Ressource Acquisition is Initialization

Bind resources to a stack object. When it is destructed, free the associate ressource.

```
// C++
template <typename T>
struct RAII {
  T* data;
  RAII(T* data) : data(data) {}
  ~RAII() {
    delete data;
  }
};

int main() {
  auto the_answer =
    RAII(new int(42));
  std::cout
    << *the_answer.data
    << "is the answer\n";
}
```

- `delete` is automatically called for us by the destructor!
- Can be improved with fancy C++ to forward args to `T`'s constructor, dereference with `operator*`, etc.

Exercise

Where's the bug?

Ressource Acquisition is Initialization

Bind resources to a stack object. When it is destructed, free the associate ressource.

```
// C++
template <typename T>
struct RAII {
  T* data;
  RAII(T* data) : data(data) {}
  ~RAII() {
    delete data;
  }
};

int main() {
  auto the_answer =
    RAII(new int(42));
  std::cout
    << *the_answer.data
    << "is the answer\n";
}
```

- `delete` is automatically called for us by the destructor!
- Can be improved with fancy C++ to forward args to T's constructor, dereference with `operator*`, etc.

Exercise

Where's the bug?

```
auto arr1 = RAII(new int(10));
auto arr2 = arr1; // copy
```

Ressource Acquisition is Initialization

Bind resources to a stack object. When it is destructed, free the associate ressource.

```
// C++
template <typename T>
struct RAII {
  T* data;
  RAII(T* data) : data(data) {}
  ~RAII() {
    delete data;
  }
};

int main() {
  auto the_answer =
    RAII(new int(42));
  std::cout
    << *the_answer.data
    << "is the answer\n";
}
```

- `delete` is automatically called for us by the destructor!
- Can be improved with fancy C++ to forward args to `T`'s constructor, dereference with `operator*`, etc.

Exercise

Where's the bug?

```
auto arr1 = RAII(new int(10));
auto arr2 = arr1; // copy
```

Double free at the end of the scope!

Let’s be smarter

Exercise

How could we solve this issue?

Two ways:

- Keep a unique owner of the resource, and forbid copies of it.
  The owner must be passed around using C++ move semantics only.
  - `std::unique_ptr`!
- Allow sharing the ownership of the resource, and count references to it.
  If this count reaches 0, release the resource.
  - `std::shared_ptr`!

Unique pointers

```cpp
// C++
template<typename T>
struct my_unique {
  T* data;

  my_unique(T* data) : data (data) {}

  my_unique(const my_unique& other) = delete;

  ~my_unique() {
    delete data;
  }
};
```

Shared pointers

```
// C++
template <typename T>
struct my_shared {
  T* data; int* count;

  my_shared(T* data) : data (data) , count(new int(1)) {}

  my_shared(const my_shared& other) {
    data = other.data; count = other.count;
    *count += 1;
  }

  ~my_shared() {
    *count -= 1;
    if (*count <= 0) {
      delete count; delete data;
    }
  }
};
```

Common bug

```
// C++
int main() {
  int* n = new int (42);
  my_shared s1(n);
  my_shared s2(n);

  return 0;
}
```

Exercise

What's wrong here?

Common bug

```
// C++
int main() {
  int* n = new int (42);
  my_shared s1(n);
  my_shared s2(n);

  return 0;
}
```

Exercise

What's wrong here?

- Another double free!
  - Same issue with `std::shared_ptr` and `std::unique_ptr`...
  - Always use `std::make_shared` or `std::make_unique` (or since C++20 constructors) to avoid this.

In other languages

- RAII is closely associated with C++.
  - Bjarne Stroustrup actually coined the term in the 80s/90s.
- Also present in Ada, Rust, or even C! (with extensions)

In other languages

- RAII is closely associated with C++.
  - Bjarne Stroustrup actually coined the term in the 80s/90s.
- Also present in Ada, Rust, or even C! (with extensions)

```
// C
void my_free(char **p) {
  puts(*p); free(*p);
}

int main(void) {
  __attribute__((cleanup(my_free)))
  char* ptr =
    (char*) calloc(20, sizeof(char));
  strcpy(ptr, "Hello world");
  return 0;
  // ptr out of scope: call my_free
}
```

In other languages

- RAII is closely associated with C++.
  - Bjarne Stroustrup actually coined the term in the 80s/90s.
- Also present in Ada, Rust, or even C! (with extensions)

```c
// C
void my_free(char **p) {
  puts(*p); free(*p);
}

int main(void) {
  __attribute__((cleanup(my_free)))
  char* ptr =
    (char*) calloc(20, sizeof(char));
  strcpy(ptr, "Hello world");
  return 0;
  // ptr out of scope: call my_free
}
```

Deferred statements

Go, Zig... provide a `defer` keyword to delay statements until the scope end to a similar effect.

```go
// Go
package main
import "fmt"
func main() {
    defer fmt.Println("world")
    fmt.Println("hello")
    // prints "hello\nworld\n"
}
```

Lifetime issues

```
// C
int main(void) {
  putchar(*get_charX());
  return 0;
}
```

Exercise

Given this C main and the following functions...

- Do they compile?
- Do they cause errors?

Lifetime issues

```c
// C
int main(void) {
  putchar(*get_charX());
  return 0;
}
```

Exercise

Given this C main and the following functions...

- Do they compile?
- Do they cause errors?

```c
char* get_char1() {
  char s[] = "Hello";

  return &s[0];
}
```

Lifetime issues

```c
// C
int main(void) {
  putchar(*get_charX());
  return 0;
}
```

Exercise

Given this C main and the following functions...

- Do they compile?
- Do they cause errors?

```c
char* get_char1() {
  char s[] = "Hello";

  return &s[0];
}
```

```c
char* get_char2() {
  char *s = "Hello";

  return &s[0];
}
```

Lifetime issues

```c
// C
int main(void) {
  putchar(*get_charX());
  return 0;
}
```

Exercise

Given this C main and the following functions…

- Do they compile?
- Do they cause errors?

```c
char* get_char1() {
  char s[] = "Hello";

  return &s[0];
}
```

```c
char* get_char2() {
  char *s = "Hello";

  return &s[0];
}
```

```c
char* get_char3() {
  char s1[] = "Hello";
  char *s = s1;
  return &s[0];
}
```

Lifetime issues

```
// C
int main(void) {
  putchar(*get_charX());
  return 0;
}
```

Exercise

Given this C main and the following functions...

- Do they compile?
- Do they cause errors?

```
char* get_char1() {
  char s[] = "Hello";

  return &s[0];
}
```

```
char* get_char2() {
  char *s = "Hello";

  return &s[0];
}
```

```
char* get_char3() {
  char s1[] = "Hello";
  char *s = s1;
  return &s[0];
}
```

2 works, 1 and 3 are undefined and likely segfault. Only 1 triggers a *warning*...

Lifetime issues

```c
// C
int main(void) {
  putchar(*get_charX());
  return 0;
}
```

Exercise

Given this C main and the following functions…

- Do they compile?
- Do they cause errors?

```c
char* get_char1() {
  char s[] = "Hello";

  return &s[0];
}
```

```c
char* get_char2() {
  char *s = "Hello";

  return &s[0];
}
```

```c
char* get_char3() {
  char s1[] = "Hello";
  char *s = s1;
  return &s[0];
}
```

2 works, 1 and 3 are undefined and likely segfault. Only 1 triggers a *warning*…

Exercise

Why??

Lifetime issues

```
// C
int main(void) {
  putchar(*get_charX());
  return 0;
}
```

Exercise

Given this C main and the following functions…

- Do they compile?
- Do they cause errors?

```
char* get_char1() {
  char s[] = "Hello";

  return &s[0];
}
```

```
char* get_char2() {
  char *s = "Hello";

  return &s[0];
}
```

```
char* get_char3() {
  char s1[] = "Hello";
  char *s = s1;
  return &s[0];
}
```

2 works, 1 and 3 are undefined and likely segfault. Only 1 triggers a *warning*…

Exercise

Why??

- `char *s = "..."` defines a string literal with a static lifetime
- `char s[] = "..."` defines an array of chars on the stack

Section 3

Borrow Checking

Rust Borrows

In a nutshell:

- A given value has a single owner.
  - The owner is responsible for deallocating this value.
- This value can be:
  - borrowed by a function, in which case the owner stays the same.
  - moved to a function, in which case the ownership is also moved to the function.
- Every borrow has a lifetime, and is valid only within this lifetime.

- Very similar to C++ move semantics but better.

- Rust will *never* let you borrow and return something local to the function.

Moving values

```
// Rust
fn use_vec(_ : Vec<i64>) {}

fn main() {
    let v = vec![1, 2, 3];
    println!("{}", v[0]);
    use_vec(v);
    println!("{}", v[0]);
}
```

Exercise

What's wrong here?

Moving values

```
error[E0382]: borrow of moved value: `v`
 --> test.rs:7:20
  |
4 |     let v = vec![1, 2, 3];
  |         - move occurs because `v` has type `Vec<i64>`, which does
            not implement the `Copy` trait
5 |     println!("{}", v[0]);
6 |     use_vec(v);
  |             - value moved here
7 |     println!("{}", v[0]);
  |                    ^ value borrowed here after move
  |
```

Moving values

```
note: consider changing this parameter type in function `use_vec` to
      borrow instead if owning the value isn't necessary
 --> test.rs:1:16
  |
1 | fn use_vec(_ : Vec<i64>) {}
  |    -------     ^^^^^^^^ this parameter takes ownership of the
  |    |                    value
  |    |
  |    in this function
help: consider cloning the value if the performance cost is
      acceptable
  |
6 |     use_vec(v.clone());
  |              ++++++++

error: aborting due to previous error
```

Borrowing values

If we do not want to `clone` the vector, we can borrow it instead.

```
// Rust
fn use_vec(_ : &Vec<i64>) {}
            // ^ indicates a borrow
fn main() {
    let v = vec![1, 2, 3];
    println!("{}", v[0]);
    use_vec(&v);
         // ^ explicitely borrow v
    println!("{}", v[0]);
}
```

Mutability

- Rust defaults to constant values to prevent unnecessary mutability.
- To modify `v` we need to:
  - declare it as `mut`.
  - borrow it mutably.

```rust
// Rust
fn add_to_vec(v : &mut Vec<i64>) {
    v.push(42)
}

fn main() {
    let mut v = vec![1, 2, 3];
    println!("{}", v[0]);
    add_to_vec(&mut v);
    println!("{}", v[0]);
}
```

Mutability

We could also just clone the array and move it to a function with a mutable argument.

```rust
// Rust
fn add_to_vec(mut v : Vec<i64>) {
    v.push(42)
}

fn main() {
    let v = vec![1, 2, 3];
    println!("{}", v[0]);
    add_to_vec(v.clone());
    println!("{}", v[0]);
}
```

Fields have owners

```rust
// Rust
struct S {
    pub v1: Vec<i64>,
    pub v2: Vec<i64>,
}

fn use_vec(_ : Vec<i64>) {}

fn main() {
    let s = S {
        v1: vec![1, 2, 3],
        v2: vec![4, 5, 6],
    };
    println!("{}", s.v1[0]); // OK.
    use_vec(s.v1);
    println!("{}", s.v1[0]); // ERROR: s.v1 was moved.
    println!("{}", s.v2[0]); // OK.
}
```

Ownership and move semantics concern every lvalue.

Here, only part of `s` is moved. `s.v2` is still accessible after `s.v1` was moved.

Lifetimes

- In this C code, `s` has a *static* lifetime.
- We should be able to access its content with a pointer at any time.

```c
// C
char* get_char() {
  char *s = "Hello";
  return &s[0];
}

int main(void) {
  putchar(*get_char());
}
```

Naive translation to Rust (using `u8` instead of `char`).

```rust
// Rust
// using u8 instead of char because
// UTF-8 is complex.
fn get_char() -> &u8 {
    let s : &str = "Hello";
    &s.as_bytes()[0]
}

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

Compile error: we must indicate a lifetime for the return value.

Lifetimes

- Rust borrows all have a *lifetime* (which can be implicit).
  - Time during which the borrow is valid.
- `&'a mut` is a mutable borrow with lifetime `'a`.
- `'static` is a special lifetime representing static values (alive during the whole program).

```rust
// Rust
fn get_char() -> &'static u8 {
    let s : &str = "Hello";
    &s.as_bytes()[0]
}

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

Lifetimes

What if we passed `s` as an argument?

```
// Rust
fn get_char(s : &str)
  -> &'static u8 {
    &s.as_bytes()[0]
}

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

Same as:

```
// Rust
fn get_char<'a>(s : &'a str)
  -> &'static u8 {
    &s.as_bytes()[0]
}

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

With `get_char` generic over lifetime `'a`.

Lifetimes

What if we passed `s` as an argument?

```
// Rust
fn get_char(s : &str)
  -> &'static u8 {
    &s.as_bytes()[0]
}

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

Same as:

```
// Rust
fn get_char<'a>(s : &'a str)
  -> &'static u8 {
    &s.as_bytes()[0]
}

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

With `get_char` generic over lifetime `'a`.

- Compile error: lifetime may not live long enough.
- Hint: a must outlive static.

Lifetimes

```
// Rust
fn get_char(s : &'static str)
  -> &'static u8 {
    &s.as_bytes()[0]
}

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

Or more simply:

```
// Rust
fn get_char<'a>(s : &'a str)
  -> &'a u8 {
    &s.as_bytes()[0]
}

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

You could also remove explicit lifetimes here.

As long as we can borrow s, we can borrow one of its elements!