

# Error Handling

Ryan Eberhardt and Julio Ballista  
April 13, 2021

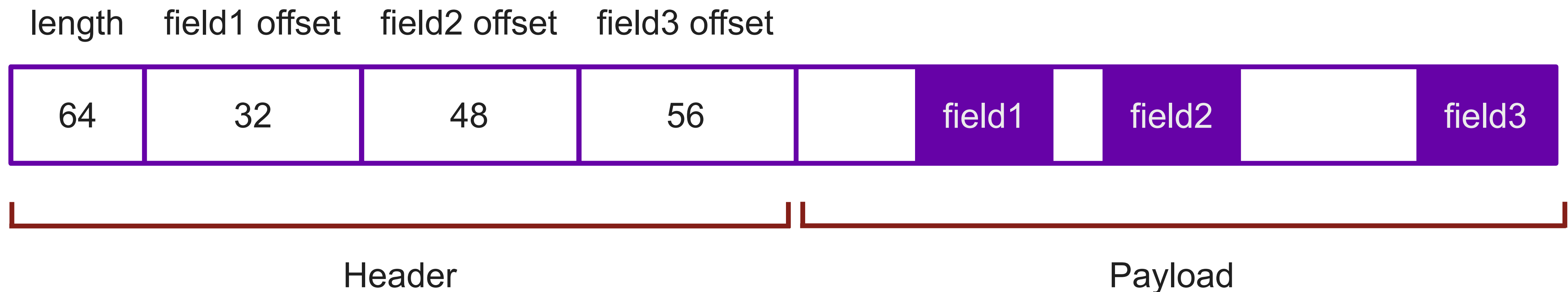
# Logistics

- Congrats on making it to week 3! 🎉
  - Already 1/3 through the quarter!
- Please submit Github usernames ASAP
  - May also want to accept the Github invite even if you aren't starting the assignment right away
- Week 2 exercises due Thursday
  - Let us know if you need more time
  - Blog posts welcome
- Ownership code examples on the website (see lecture notes)
- Please submit reflections 📝
  - Also would love to hear if this is not so helpful, or if we could make this more helpful to you

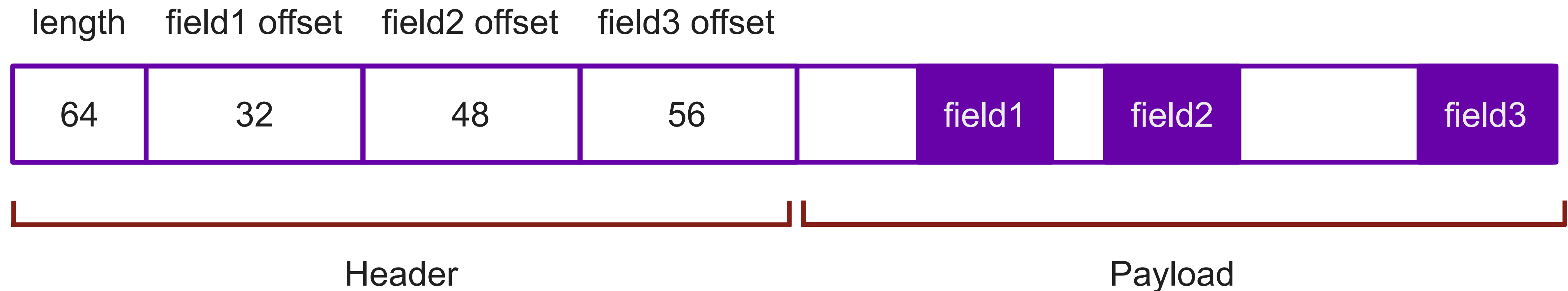
# Error handling

# More Remote Code Execution

- Imagine a server receives messages from the network
- Message is variable length. A header at the start of the message specifies where data is located within the message payload



# More Remote Code Execution



```
struct message {
    size_t length;
    size_t field1_offset;
    size_t field2_offset;
    size_t field3_offset;
    char payload[];
}
```

```
// Processing part of a received message:
void *local_copy = malloc(message->length);
// Validate field1_offset is within the buffer (no buffer overflows!)
assert(message->field1_offset + FIELD_1_LENGTH <= message->length);
// Copy the field and process it
memcpy(local_copy + message->field1_offset,
        message->payload + message->field1_offset,
        FIELD_1_LENGTH);
process_field_1(local_copy + message->field1_offset);
```

# Two issues

- Lack of proper error handling
- Use of NULL in place of a real value

# Handling errors

# Error handling in C

- If a function might encounter an error, its return type is made to be `int` (or sometimes `void*`).
- If the function is successful, it returns `0`. Otherwise, if an error is encountered, it returns `-1`. (If the function is returning a pointer, it returns a valid pointer in the success case, or `NULL` if an error occurs.)
- The function that encountered the error sets the global variable `errno` to be an integer indicating what went wrong. If the caller sees that the function returned `-1` or `NULL`, it can check `errno` to see what error was encountered

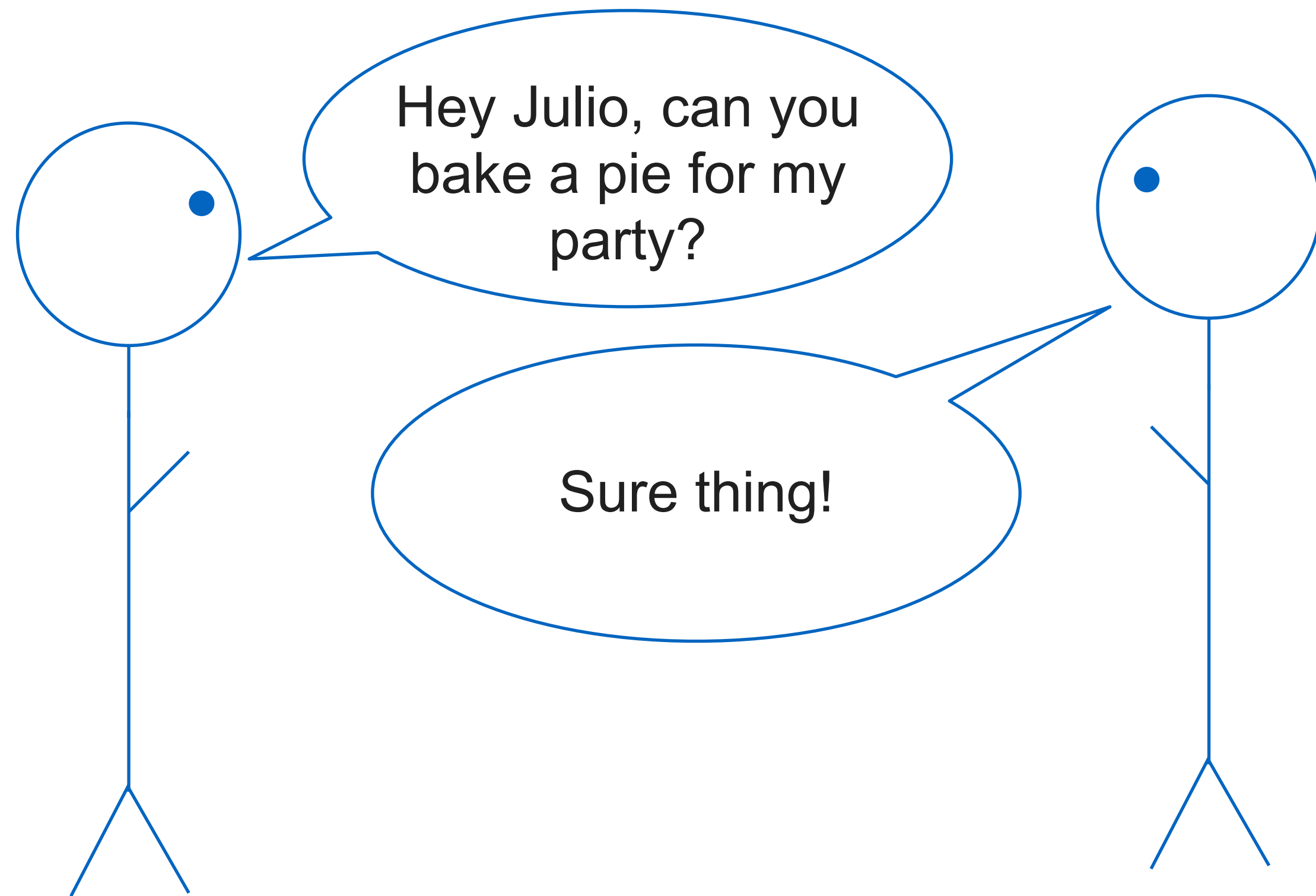


```

#define EPERM          1      /* Operation not permitted */
#define ENOENT         2      /* No such file or directory */
#define ESRCH          3      /* No such process */
#define EINTR          4      /* Interrupted system call */
#define EIO            5      /* I/O error */
#define ENXIO          6      /* No such device or address */
#define E2BIG          7      /* Arg list too long */
#define ENOEXEC        8      /* Exec format error */
#define EBADF          9      /* Bad file number */
#define ECHILD         10     /* No child processes */
#define EAGAIN         11     /* Try again */
#define ENOMEM         12     /* Out of memory */
#define EACCES         13     /* Permission denied */
#define EFAULT         14     /* Bad address */
#define ENOTBLK        15     /* Block device required */
#define EBUSY          16     /* Device or resource busy */
#define EEXIST         17     /* File exists */
#define EXDEV          18     /* Cross-device link */
#define ENODEV         19     /* No such device */
#define ENOTDIR        20     /* Not a directory */
#define EISDIR         21     /* Is a directory */
#define EINVAL         22     /* Invalid argument */
#define ENFILE         23     /* File table overflow */
#define EMFILE         24     /* Too many open files */
#define ENOTTY         25     /* Not a typewriter */
#define ETXTBSY        26     /* Text file busy */
#define EFBIG          27     /* File too large */
#define ENOSPC         28     /* No space left on device */
#define ESPIPE         29     /* Illegal seek */
#define EROFS          30     /* Read-only file system */
#define EMLINK         31     /* Too many links */
#define EPIPE          32     /* Broken pipe */
#define EDOM           33     /* Math argument out of domain of func */
#define ERANGE         34     /* Math result not representable */
#define EDEADLK        35     /* Resource deadlock would occur */
#define ENAMETOOLONG   36     /* File name too long */
#define ENOLCK         37     /* No record locks available */
#define ENOSYS         38     /* Function not implemented */
#define ENOTEMPTY      39     /* Directory not empty */
#define ELOOP          40     /* Too many symbolic links encountered */
#define EWOULDBLOCK    EAGAIN /* Operation would block */
#define ENOMSG         42     /* No message of desired type */
#define EIDRM          43     /* Identifier removed */
#define ECHRNG         44     /* Channel number out of range */
#define EL2NSYNC       45     /* Level 2 not synchronized */
#define EL3HLT         46     /* Level 3 halted */
#define EL3RST         47     /* Level 3 reset */
#define ELNRNG         48     /* Link number out of range */
#define EUNATCH        49     /* Protocol driver not attached */
#define EL2HLT         51     /* Level 2 halted */
#define EBADE          52     /* Invalid exchange */
#define EBADR          53     /* Invalid request descriptor */
#define EXFULL         54     /* Exchange full */
#define ENOANO         55     /* No anode */
#define EBADRQC        56     /* Invalid request code */
#define EBADSLT        57     /* Invalid slot */
#define EBFONT         59     /* Bad font file format */
#define ENOSTR         60     /* Device not a stream */
#define ENODATA        61     /* No data available */
#define ETIME          62     /* Timer expired */
#define ENOSR          63     /* Out of streams resources */
#define ENONET         64     /* Machine is not on the network */
#define ENOPKG         65     /* Package not installed */
#define EREMOTE        66     /* Object is remote */
#define ENOLINK        67     /* Link has been severed */
#define EADV           68     /* Advertise error */
#define ESRMNT         69     /* Srmount error */
#define ECOMM          70     /* Communication error on send */
#define EPROTO         71     /* Protocol error */
#define EMULTIHOP      72     /* Multihop attempted */
#define EDOTDOT        73     /* RFS specific error */
#define EBADMSG        74     /* Not a data message */
#define EOVERFLOW      75     /* Value too large for defined data type */
#define ENOTUNIQ       76     /* Name not unique on network */
#define EBADFD         77     /* File descriptor in bad state */
#define EREMCHG        78     /* Remote address changed */
#define ELIBACC        79     /* Can not access a needed shared library */
#define ELIBBAD        80     /* Accessing a corrupted shared library */
#define ELIBSCN        81     /* .lib section in a.out corrupted */
#define ELIBMAX        82     /* Attempting to link in too many shared libraries */
#define ELIBEXEC       83     /* Cannot exec a shared library directly */
#define EILSEQ         84     /* Illegal byte sequence */
#define ERESTART       85     /* Interrupted system call should be restarted */
#define ESTRPIPE       86     /* Streams pipe error */
#define EUSERS         87     /* Too many users */
#define ENOTSOCK       88     /* Socket operation on non-socket */
#define EDESTADDRREQ   89     /* Destination address required */
#define EMSGSIZE       90     /* Message too long */
#define EPROTOTYPE     91     /* Protocol wrong type for socket */
#define ENOPROTOOPT    92     /* Protocol not available */
#define EPROTONOSUPPORT 93     /* Protocol not supported */
#define ESOCKTNOSUPPORT 94     /* Socket type not supported */
#define EOPNOTSUPP     95     /* Operation not supported on transport endpoint */
#define EPFNOSUPPORT   96     /* Protocol family not supported */
#define EAFNOSUPPORT   97     /* Address family not supported by protocol */
#define EADDRINUSE     98     /* Address already in use */
#define EADDRNOTAVAIL 99     /* Cannot assign requested address */
...

```

# Error handling in C

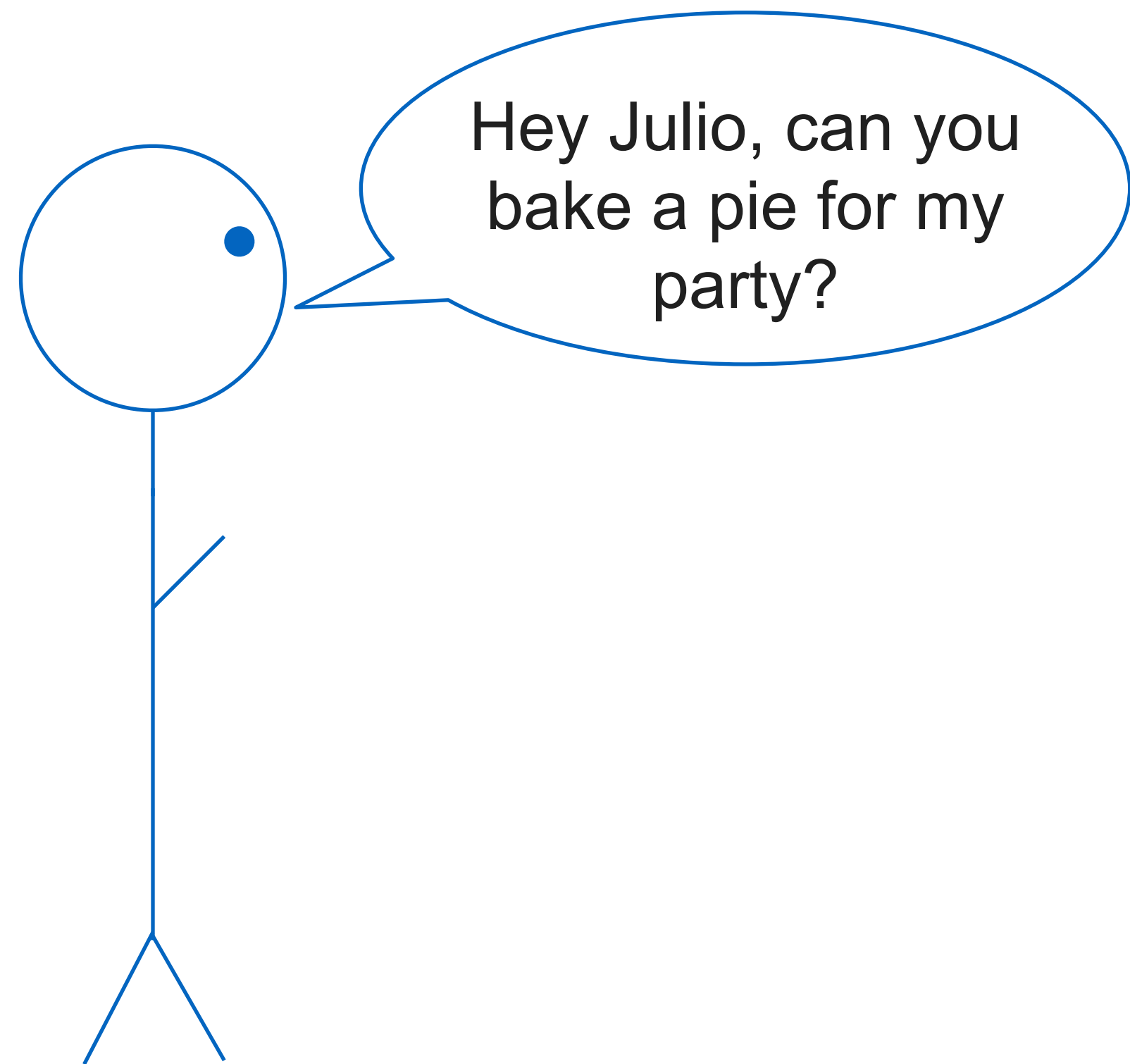


`int main()`

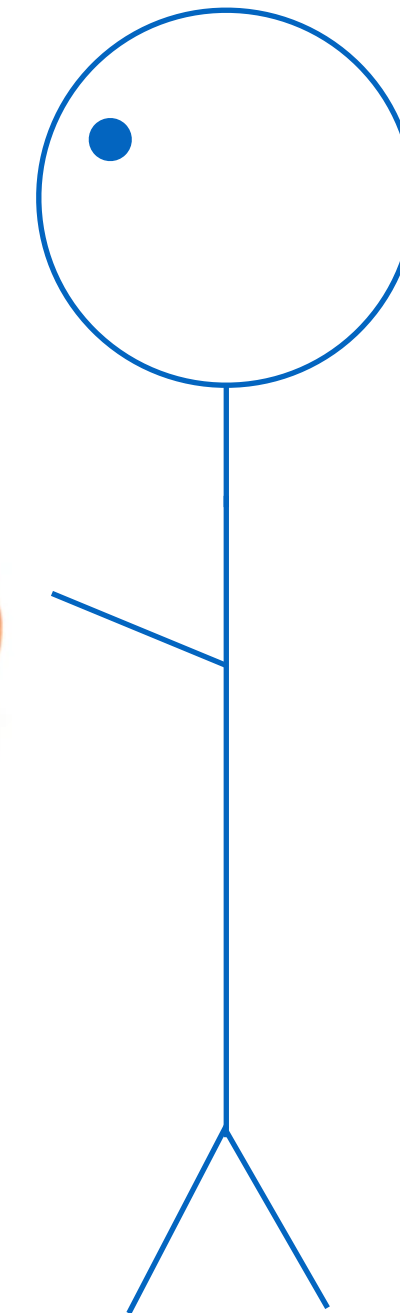
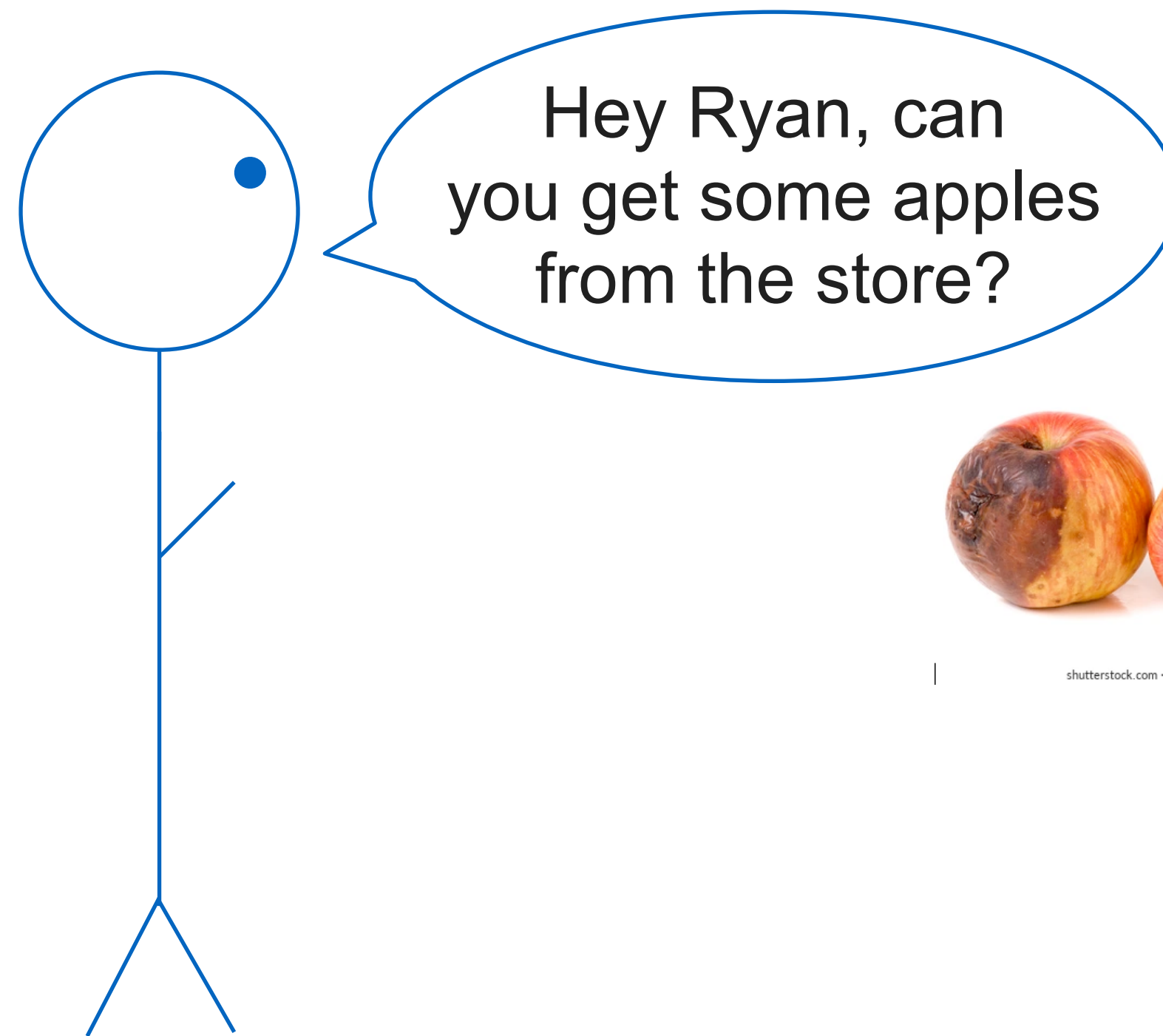
julio:  
`struct apple_pie *make_pie() {  
 get_apples();  
 bake_ingredients();  
}`

ryan:  
`struct apple *get_apples()`

# Error handling in C



int main()

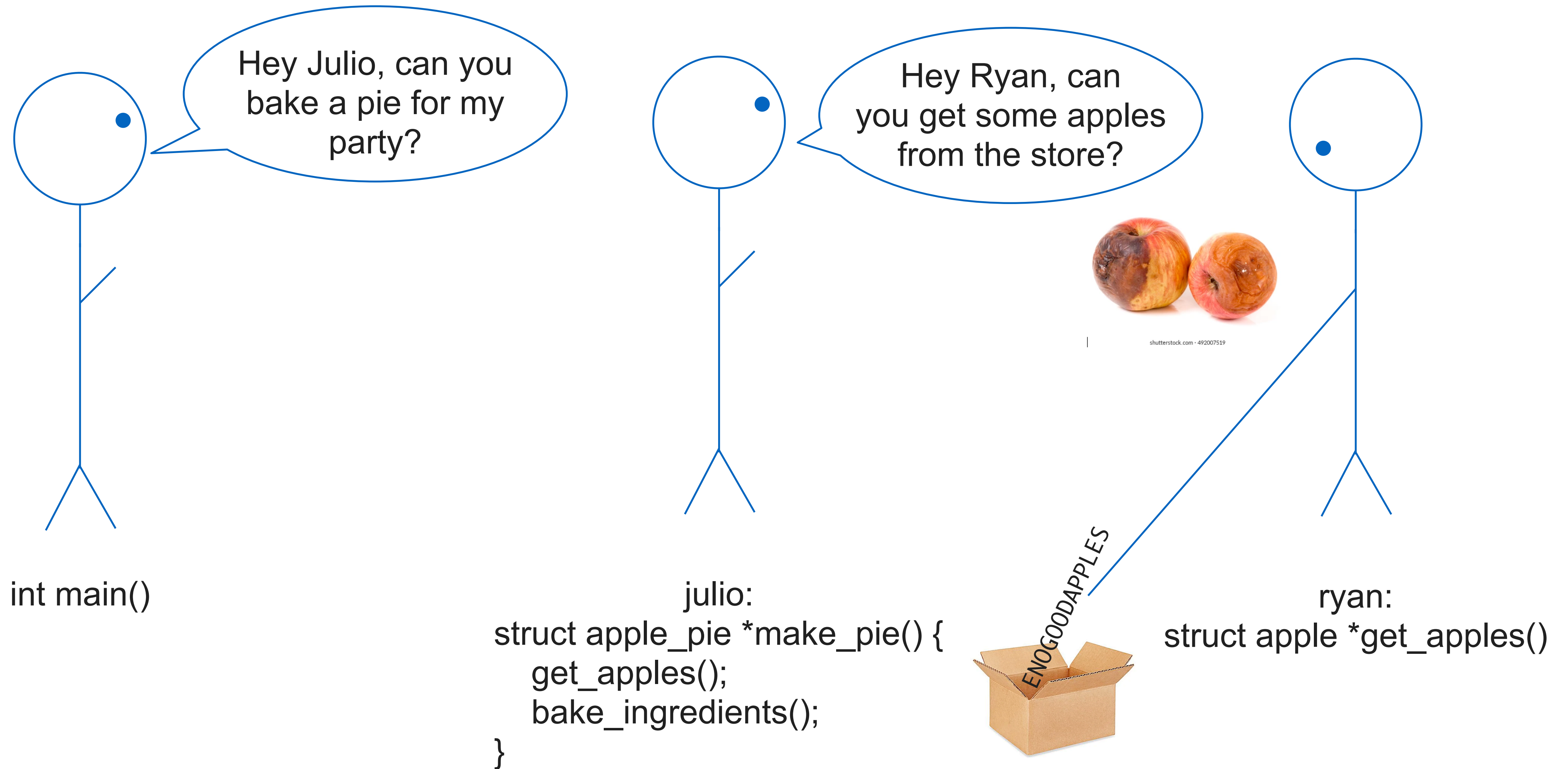


ryan:  
struct apple \*get\_apples()

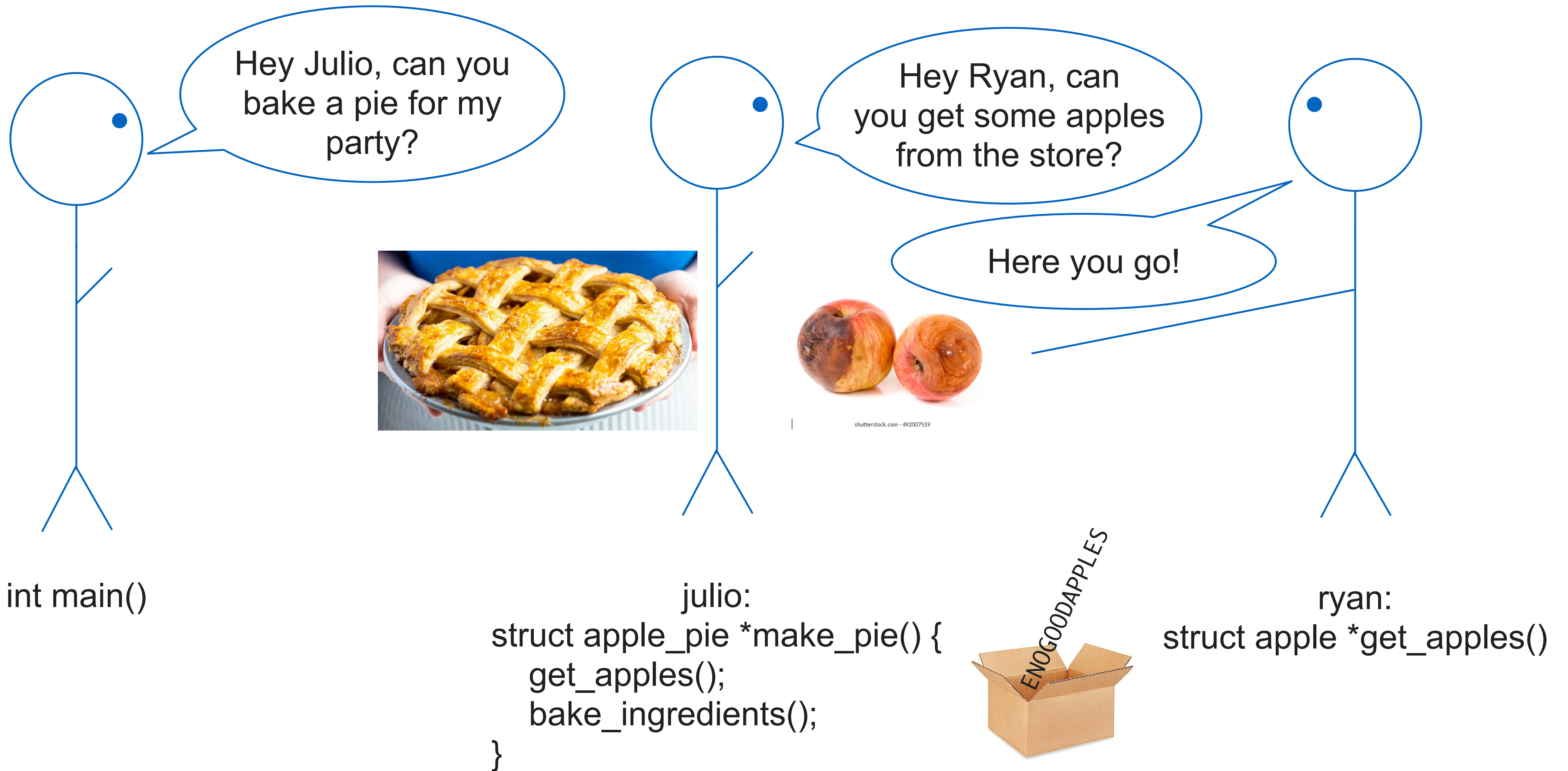
julio:  
struct apple\_pie \*make\_pie() {  
 get\_apples();  
 bake\_ingredients();  
}



# Error handling in C



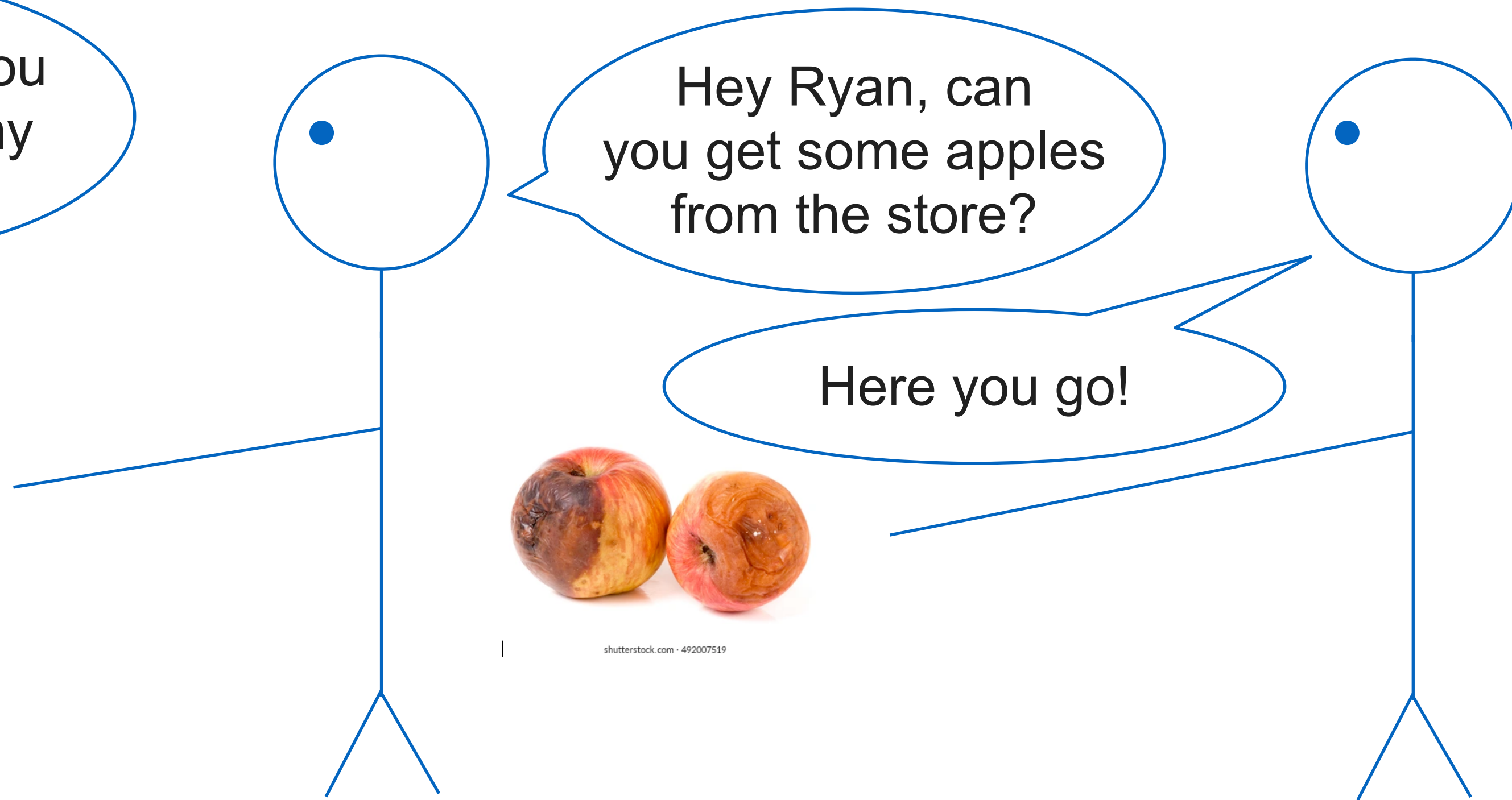
# Error handling in C



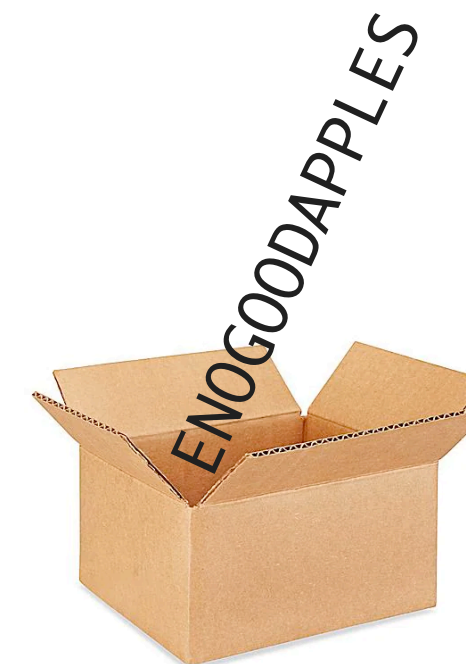
# Error handling in C



```
int main()
```



```
    julio:  
    struct apple_pie *make_pie() {  
        get_apples();  
        bake_ingredients();  
    }
```



```
    ryan:  
    struct apple *get_apples()
```

# Broken code from earlier

```
struct message {  
    size_t length;  
    size_t field1_offset;  
    size_t field2_offset;  
    size_t field3_offset;  
    char payload[];  
}
```

```
// Processing part of a received message:  
void *local_copy = malloc(message->length);  
// Validate field1_offset is within the buffer (no buffer overflows!)  
assert(message->field1_offset + FIELD_1_LENGTH <= message->length);  
// Copy the field and process it  
memcpy(local_copy + message->field1_offset,  
        message->payload + message->field1_offset,  
        FIELD_1_LENGTH);  
process_field_1(local_copy + message->field1_offset);
```

Missing error check! 💣

# CVE-2015-8812

- Critical Linux kernel vulnerability: by sending a malformed network packet, a remote attacker could execute arbitrary code in the kernel
- A set of kernel networking functions were returning -1 for error, 0 for success, but also other values for “warnings”
  - Returned NET\_XMIT\_CN (defined to be 2) when congestion was detected
- Code calling these functions saw nonzero return code and assumed there was a network error
- Freed memory that was still being used for the network. Use-after-free + double free!



# The fix

```
--- a/drivers/infiniband/hw/cxgb3/iwch_cm.c
+++ b/drivers/infiniband/hw/cxgb3/iwch_cm.c
@@ -149,7 +149,7 @@ static int iwch_l2t_send(struct t3cdev *tdev, struct sk_buff *skb, struct
l2t_en
    error = l2t_send(tdev, skb, l2e);
    if (error < 0)
        kfree_skb(skb);
-   return error;
+   return error < 0 ? error : 0;
}
```

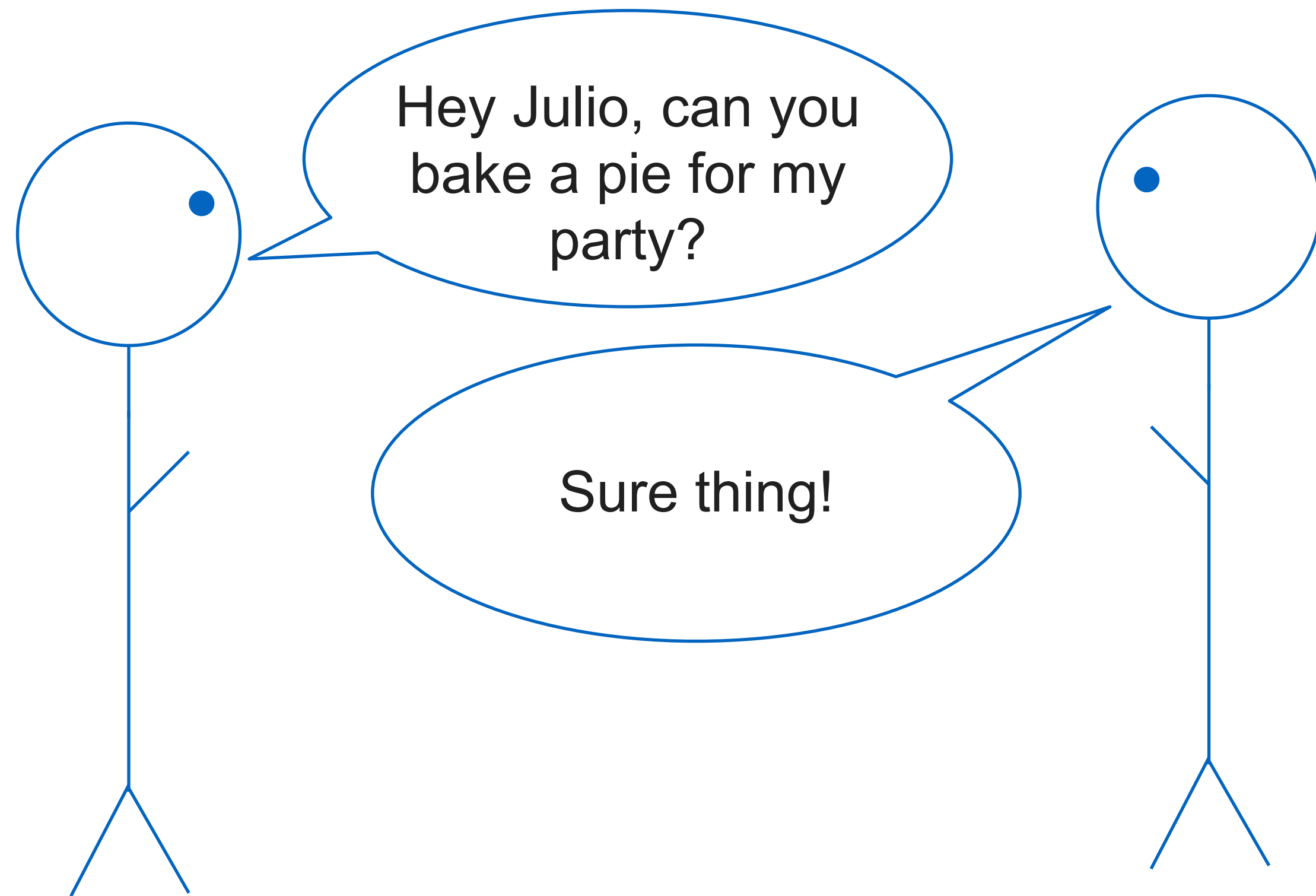


# Proper C error checking is ugly

- Programmer must remember whether the function they are calling might return an error
  - Places immense burden on the programmer to remember how each specific function works
- After every function call that might return an error, must check whether an error occurred and handle it **correctly**
  - This isn't good enough (why not?):

```
void *buf = malloc();  
if (buf == NULL) {  
    perror("error allocating memory");  
}  
memcpy(buf + offset, src, size);
```
- Handling specific errors using `errno` can produce an error-prone mess of `if` statements
  - [Sometimes function documentation does not even properly document what errors might be returned](#)

# Error handling in most languages: Exceptions

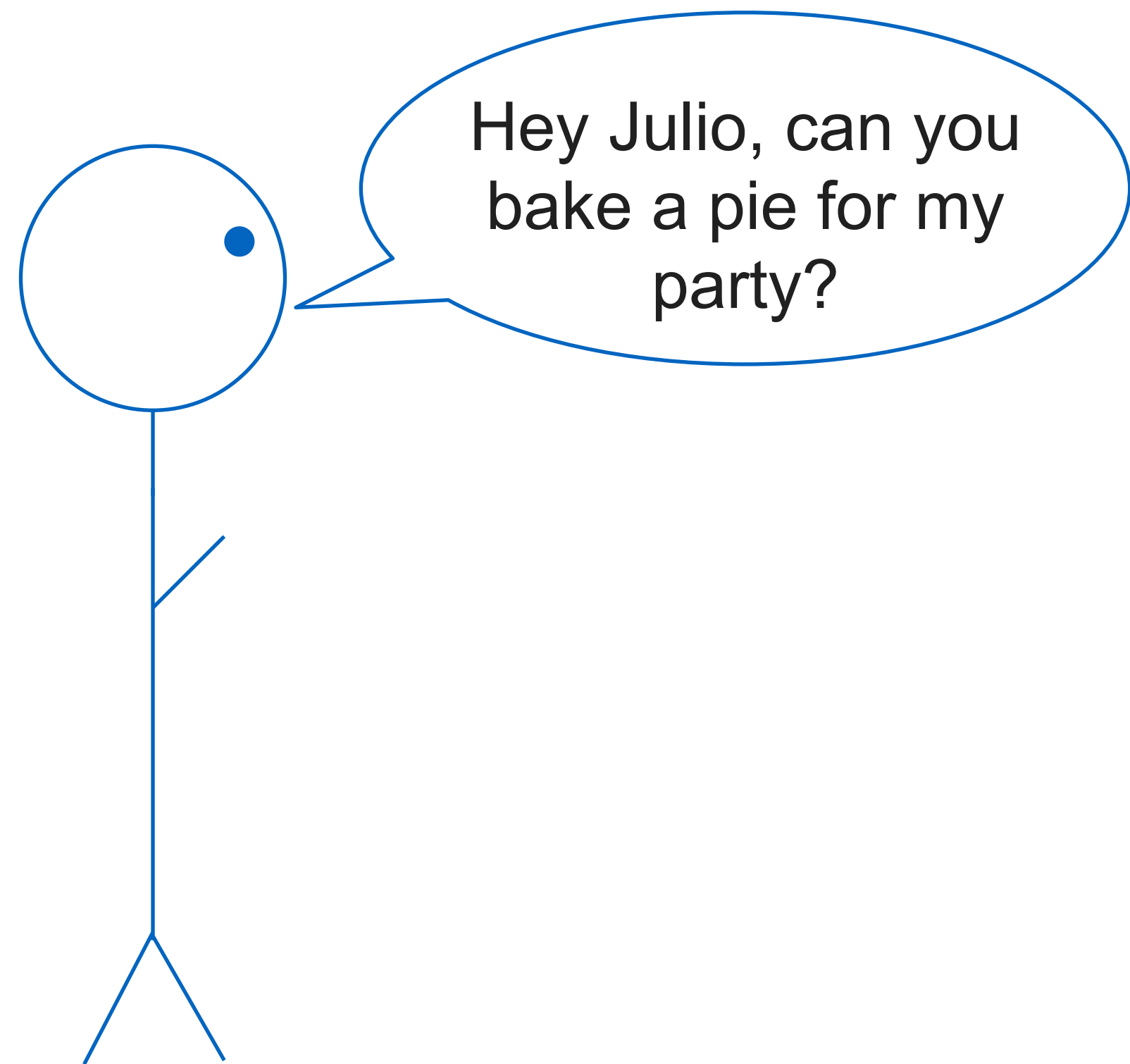


`int main()`

`julio:`  
`struct apple_pie *make_pie() {`  
 `get_apples();`  
 `bake_ingredients();`  
`}`

`ryan:`  
`struct apple *get_apples()`

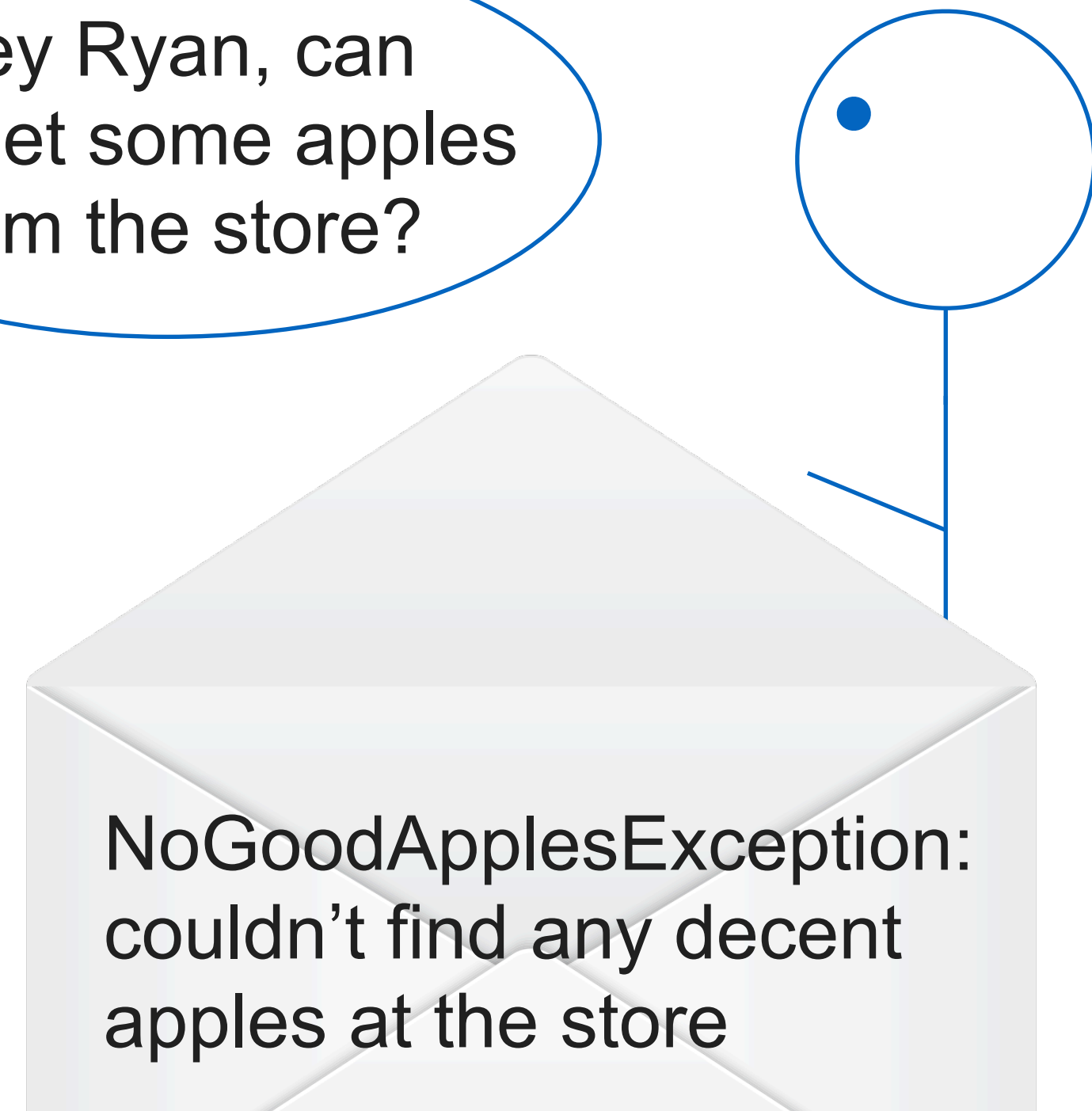
# Error handling in most languages: Exceptions



`int main()`

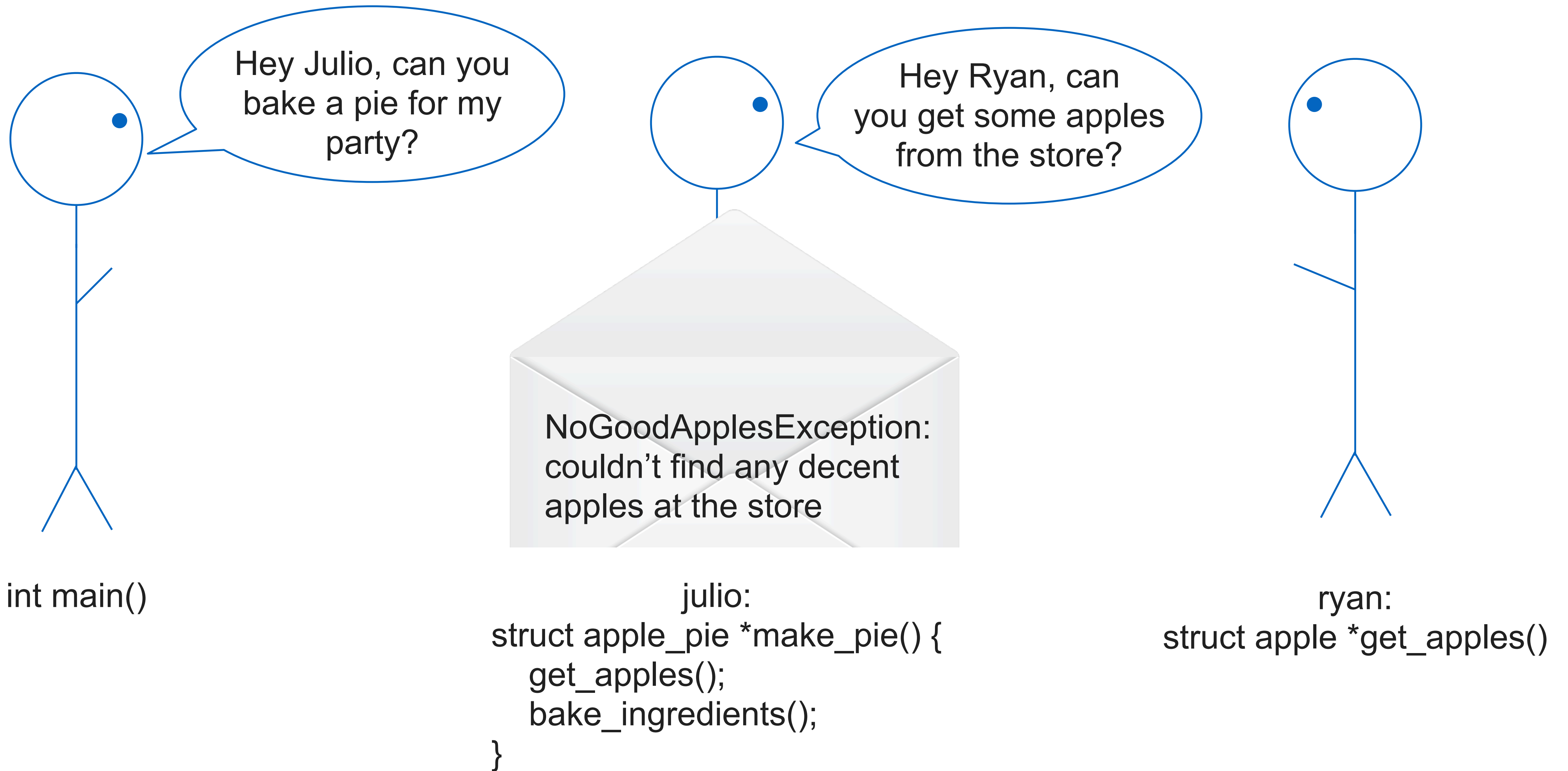


julio:  
`struct apple_pie *make_pie() {  
 get_apples();  
 bake_ingredients();  
}`

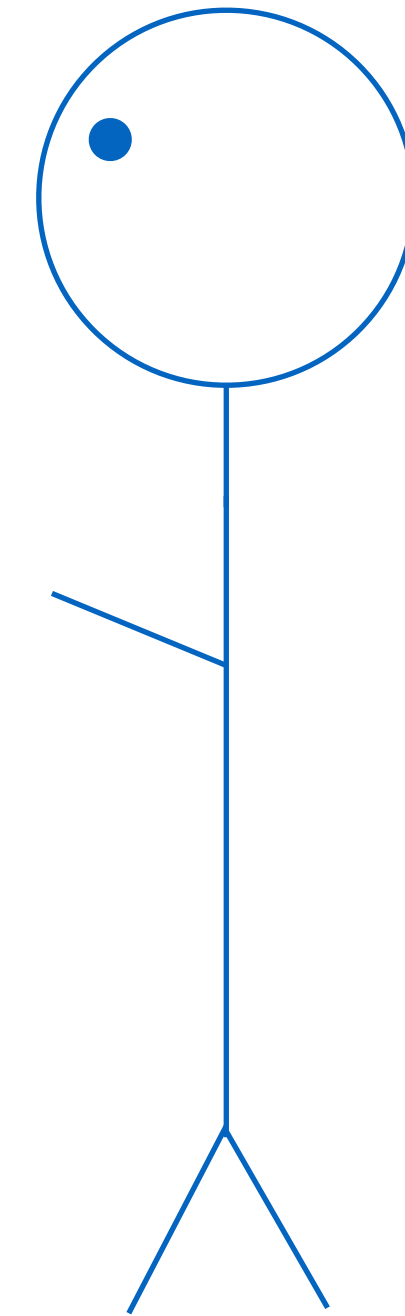
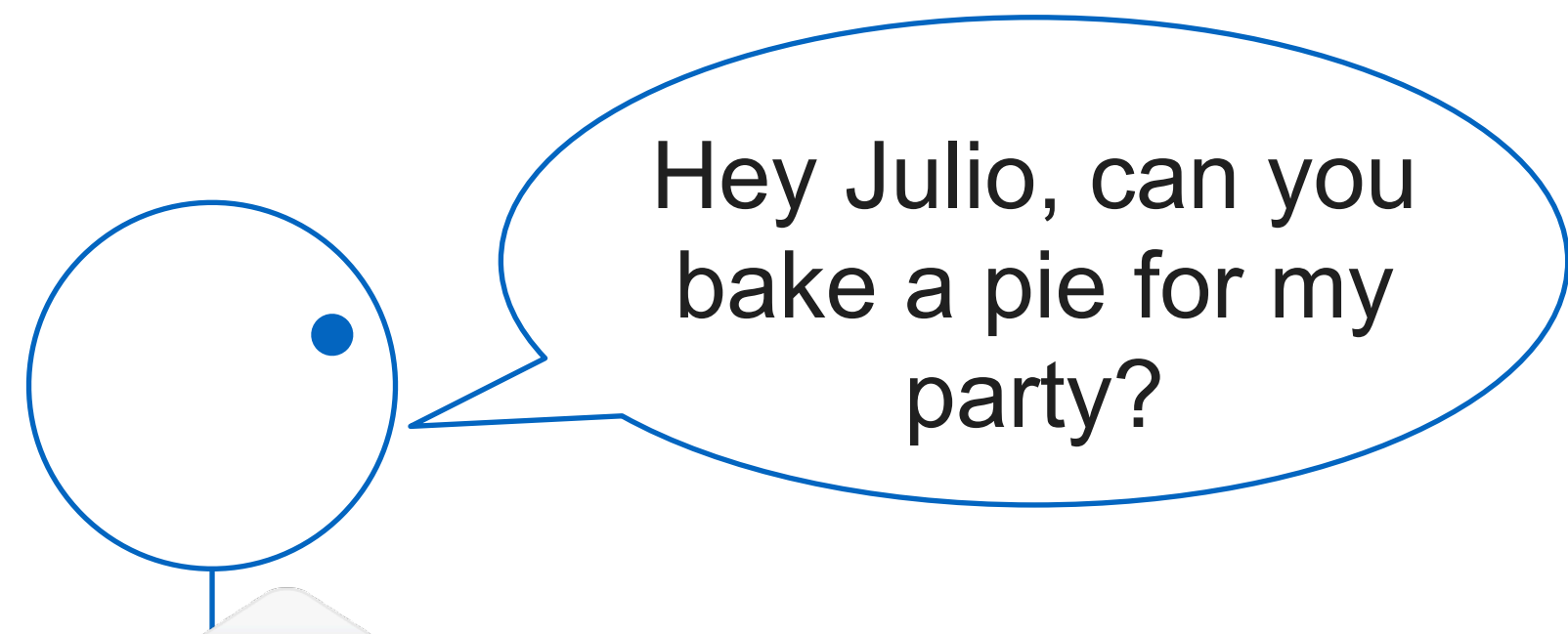


ryan:  
`struct apple *get_apples()`

# Error handling in most languages: Exceptions



# Error handling in most languages: Exceptions



NoGoodApplesException:  
couldn't find any decent  
apples at the store

int main()



julio:  
struct apple\_pie \*make\_pie() {  
 get\_apples();  
 bake\_ingredients();  
}

ryan:  
struct apple \*get\_apples()

# Massive improvements over C-style error handling

- You don't have to write error propagation code *every* time you call a function that might produce an error
  - Exceptions propagate up the stack automatically until they are handled by a try/catch
- Errors will not go unnoticed
  - Worst case scenario, they'll propagate up to main() and crash the program
  - Sounds bad, but a crash is much better than the program continuing to run in an undefined state

# Except Exceptions

- Why might exceptions not be so hot?
  - Failure modes become hard to reason about: any function can throw any exception at any time
    - Code might fail because of an exception that was thrown by a totally unrelated function twelve function calls away
    - Hard to spot where errors may occur
      - What if you call a helper function in a destructor that ends up throwing an exception?
    - Even harder to manage in evolving codebases as new errors are added
  - Can cause resource leaks and other unexpected behavior
    - Exceptions are forbidden in many codebases for this reason



# Exceptions without RAII = sad times

```
void process_input() {
    char *buf = malloc(128);

    // read input from user:
    fgets(buf, 128, stdin);
    // do more processing on input:
    some_helper(input);

    free(buf);
}
```

Looks good to me?

```
int main() {
    while (true) {
        try {
            process_input();
        } catch (BadInputError) {
            cerr << "That wasn't valid, try again" << endl;
        }
    }
}

void some_helper(string input) {
    if (input == "uh oh") {
        throw BadInputError("I don't like that");
    }
}
```

# Towards better error handling: Enums

- An *enum* (enumeration) is a type that can contain one of several *variants*

```
enum TrafficLightColor { Type TrafficLightColor
    Red,
    Yellow,
    Green,
}
```

← Variants

```
let current_state: TrafficLightColor = TrafficLightColor::Green;
```

- A *match* expression is like a switch statement in C/C++/Java, except all possible variants must be covered

```
fn drive(light_state: TrafficLightColor) {
    match light_state {
        TrafficLightColor::Green => println!("zoom zoom!"),
        TrafficLightColor::Red =>
            println!("sitting like a boulder!"),
    }
}
```

```
error[E0004]: non-exhaustive patterns: `Yellow` not covered
--> src/lib.rs:8:11
```

```
1 | / enum TrafficLightColor {
2 | |     Red,
3 | |     Yellow,
4 | |     ----- not covered
5 | |     Green,
6 | | }
7 | | _- `TrafficLightColor` defined here
...
8 |     match light_state {
9 |         ^^^^^^^^^^^^^ pattern `Yellow` not covered
```

= help: ensure that all possible cases are being handled, possibly by adding wildcards or more match arms

= note: the matched value is of type `TrafficLightColor`

# Towards better error handling: Enums

- An *enum* (enumeration) is a type that can contain one of several *variants*

```
enum TrafficLightColor { Type TrafficLightColor
  Red,
  Yellow, ← Variants
  Green,
}
```

```
let current_state: TrafficLightColor = TrafficLightColor::Green;
```

- A *match* expression is like a switch statement in C/C++/Java, except all possible variants must be covered

```
fn drive(light_state: TrafficLightColor) {
  match light_state {
    TrafficLightColor::Green => println!("zoom zoom!"),
    TrafficLightColor::Yellow => println!("slowing down..."),
    TrafficLightColor::Red => println!("sitting like a boulder!"),
  }
}
```

- The compiler will warn you if there's a possibility you missed!

# Towards better error handling: Enums

- Can use a default binding to catch all other cases if there's only a few you're interested in:

```
match light_state {  
  TrafficLightColor::Green => println!("zoom zoom!"),  
  _ => println!("do not pass go"), ← Default binding  
}
```

# Towards better error handling: Enums

- Unlike enums in most common languages, Rust enums can store arbitrary data!

```
enum Location {  
    Coordinates(f32, f32),  
    Address(String),  
    Unknown,  
}
```

- You can extract data from variants using a match expression:

```
fn print_location(loc: Location) {  
    match loc {  
        Location::Coordinates(lat, long) => {  
            println!("Person is at ({} , {})", lat, long);  
        },  
        Location::Address(addr) => {  
            println!("Person is at {}", addr);  
        },  
        Location::Unknown => println!("Location unknown!"),  
    }  
}
```

```
print_location(  
    Location::Address(  
        "353 Jane Stanford Way".to_string()));
```

# Error handling in Rust

- What if we use enums to clearly represent successful returns / errors?
  - If the functions run successfully, return `Ok(whatever return value)`
  - If an error happens, return `Err(some error object)`
- ```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

(this Result type is part of the Rust standard library, no need to define it yourself)

# Usage of Result

Success return type      Error return type

```
fn gen_num_sometimes() -> Result<u32, &'static str> {  
    if get_random_num() > 10 {  
        Ok(get_random_num())  
    } else {  
        Err("Spontaneous failure!")  
    }  
}
```

```
fn main() {  
    match gen_num_sometimes() {  
        Ok(num) => println!("Got number: {}", num),  
        Err(message) => println!("Operation failed: {}", message),  
    }  
}
```

# Comparison to C errors

- We had two main issues with C error handling:
  - It's too easy to miss errors
  - Proper error handling is too verbose (need too much extra code to propagate errors)
- This fixes the first problem: it's now obvious from the function signature which functions can return errors, and the compiler will verify that you do something with a returned error
- Second problem is still an issue!



# Comparison to C errors

- Error handling is still too verbose:

```
fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut s = String::new();

    let result = File::open(filename);

    let mut f = match result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

# Meet the ? operator

- Suppose we have `helper_function() -> Result<T, E>`
- `let val: T = helper_function()?` means:
  - If `helper_function` returns `Ok(some value)`, set `val = that value`
  - If helper function returns `Err(some error)`, stop and return/propagate that error

```
fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut s = String::new();

    let mut f = match File::open(filename) {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => return Err(e),
    }
}

fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut s = String::new();

    let mut f = File::open(filename)?;

    f.read_to_string(&mut s)?;
    Ok(s)
}
```

# Meet the ? operator

- Suppose we have `helper_function() -> Result<T, E>`
- `let val: T = helper_function()?` means:
  - If `helper_function` returns `Ok(some value)`, set `val = that value`
  - If helper function returns `Err(some error)`, stop and return/propagate that error

```
fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut s = String::new();

    let mut f = match File::open(filename) {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => return Err(e),
    }
}
```

Even shorter:

```
fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut s = String::new();
    File::open(filename)?.read_to_string(&mut s)?;
    Ok(s)
}
```

# Meet the ? operator

- Why doesn't this code compile?

```
fn read_file(filename: &str) -> String {  
    let mut contents = String::new();  
    File::open(filename)?.read_to_string(&mut contents)?;  
    contents  
}
```

- Note that the ? operator is for *propagating* errors, and this function returns `String` (i.e. it cannot return an error)

# Panics

- What about errors that we don't wish to propagate/handle?
  - Could be a serious, unrecoverable error
  - Could be an error that we don't anticipate ever happening and don't want to put the effort into handling
- The `panic!` macro crashes a program immediately with an error message

```
if sad_times() {  
    panic!("Sad times!");  
}
```
- Sometimes, a library function may give us an `Err` that we really don't care to handle in any graceful way
  - E.g. if we are a terminal program and we fail to read input from the terminal, there isn't really anything graceful to do
  - `Result::unwrap()` and `Result::expect()` allow us to extract the returned value from an `Ok()` result, panicking if we got an `Err`

# unwrap() and expect()

```
// Panic if opening a file fails:  
let mut file = File::open(filename).unwrap();  
// Same thing, but print a more descriptive error message when panicking:  
let mut file = File::open(filename).expect("Failed to open file");  
  
// Panic if reading from stdin fails:  
let mut input = String::new();  
io::stdin().read_to_string(&mut input).expect("Failed to read from stdin");
```

# Handling nulls

***“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.”***

- Tony Hoare





Common Vulnerabilities and Exposures

[CVE List](#)

[CNAs](#)

[WGs](#)

[Board](#)

[About](#)

[News & Blog](#)



Go to for:

[CVSS Scores](#)

[CPE Info](#)

[Advanced Search](#)

[Search CVE List](#)

[Download CVE](#)

[Data Feeds](#)

[Request CVE IDs](#)

[Update a CVE Entry](#)

TOTAL CVE Entries: **133847**

HOME > CVE > SEARCH RESULTS

## Search Results

There are **1627** CVE entries that match your search.

| Name                          | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">CVE-2020-9759</a> | An issue was discovered in WeeChat before 2.7.1 (0.4.0 to 2.7 are affected). A malformed message 352 (who) can cause a NULL pointer dereference in the callback function, resulting in a crash.                                                                                                                                                                                                                                                                  |
| <a href="#">CVE-2020-9385</a> | A NULL Pointer Dereference exists in libzint in Zint 2.7.1 because multiple + characters are mishandled in add_on in upcean.c, when called from eanx in upcean.c during EAN barcode generation.                                                                                                                                                                                                                                                                  |
| <a href="#">CVE-2020-9327</a> | In SQLite 3.31.1, isAuxiliaryVtabOperator allows attackers to trigger a NULL pointer dereference and segmentation fault because of generated column optimizations.                                                                                                                                                                                                                                                                                               |
| <a href="#">CVE-2020-8859</a> | This vulnerability allows remote attackers to create a denial-of-service condition on affected installations of ELOG Electronic Logbook 3.1.4-283534d. Authentication is not required to exploit this vulnerability. The specific flaw exists within the processing of HTTP parameters. A crafted request can trigger the dereference of a null pointer. An attacker can leverage this vulnerability to create a denial-of-service condition. Was ZDI-CAN-10115. |
| <a href="#">CVE-2020-8448</a> | In OSSEC-HIDS 2.7 through 3.5.0, the server component responsible for log analysis (ossec-analysisd) is vulnerable to a denial of service (NULL pointer dereference) via crafted messages written directly to the analysisd UNIX domain socket by a local user.                                                                                                                                                                                                  |
| <a href="#">CVE-2020-8011</a> | CA Unified Infrastructure Management (Nimsoft/UIM) 9.20 and below contains a null pointer dereference vulnerability in the robot (controller) component. A remote attacker can crash the Controller service.                                                                                                                                                                                                                                                     |
| <a href="#">CVE-2020-8002</a> | A NULL pointer dereference in vrend_renderer.c in virglrenderer through 0.8.1 allows attackers to cause a denial of service via commands that attempt to launch a grid without previously providing a Compute Shader (CS).                                                                                                                                                                                                                                       |
| <a href="#">CVE-2020-7105</a> | async.c and dict.c in libhiredis.a in hiredis through 0.14.0 allow a NULL pointer dereference because malloc return values are unchecked.                                                                                                                                                                                                                                                                                                                        |
| <a href="#">CVE-2020-7062</a> | In PHP versions 7.2.x below 7.2.28, 7.3.x below 7.3.15 and 7.4.x below 7.4.3, when using file upload functionality, if upload progress tracking is enabled, but session.upload_progress.cleanup is set to 0 (disabled), and the file upload fails, the upload procedure would try to clean up data that does not exist and encounter null pointer dereference, which would likely lead to a crash.                                                               |
| <a href="#">CVE-2020-6795</a> | When processing a message that contains multiple S/MIME signatures, a bug in the MIME processing code caused a null pointer dereference, leading to an unexploitable crash. This vulnerability affects Thunderbird < 68.5.                                                                                                                                                                                                                                       |
| <a href="#">CVE-2020-6631</a> | An issue was discovered in GPAC version 0.8.0. There is a NULL pointer dereference in the function gf_m2ts_stream_process_pmt() in media_tools/m2ts_mux.c.                                                                                                                                                                                                                                                                                                       |

[NULL pointer dereferences](#)

# Nulls -> not null damage

- Most null pointer dereferences simply cause crashes (denial of service)... but not all
- CVE-2009-2694 in Pidgin messenger: <https://www.cvedetails.com/cve/CVE-2009-2694/>
- `msn_slplink_message_find()` retrieves previously-received parts of a message
- Special types of messages (“acknowledgement messages”) don’t have any message contents. `message->buffer` is set to NULL
- When trying to re-assemble received data, `msn_slplink_process_msg()` calls `msn_slplink_message_find()` and then runs `memcpy(slplink->buffer + offset, data, len);`
- `slplink->buffer` is null, so the attacker-supplied offset can be used to control what memory gets overwritten
- Similar vulnerability in Adobe Acrobat Pro: <https://www.zerodayinitiative.com/advisories/ZDI-19-871/>

# A sticky situation

- Why are NULLs so dangerous?
  - They place a huge burden on the programmer: any time you have a pointer, you need to think, *is it possible for this to be NULL?*
  - Static analyzers can't warn about all the possible NULLs without being riddled with false positives
- What should we do about it?

```
enum Option<T> {  
    None,  
    Some(T),  
}
```

(also from in the standard library)

```
fn feeling_lucky() -> Option<String> {  
    if get_random_num() > 10 {  
        Some(String::from("I'm feeling lucky!"))  
    } else {  
        None  
    }  
}
```

```
fn feeling_lucky() -> Option<String> {
    if get_random_num() > 10 {
        Some(String::from("I'm feeling lucky!"))
    } else {
        None
    }
}
```

```
match feeling_lucky() {
    Some(message) => {
        println!("Got message: {}", message);
    },
    None => {
        println!("No message returned :-/");
    },
}
```

```
fn feeling_lucky() -> Option<String> {
    if get_random_num() > 10 {
        Some(String::from("I'm feeling lucky!"))
    } else {
        None
    }
}
```

```
// Check if is_none/is_some():
```

```
if feeling_lucky().is_none() {
    println!("Not feeling lucky :(");
}
```

```
// unwrap/expect work here too:
```

```
let message = feeling_lucky().unwrap();
let message = feeling_lucky().expect("feeling_lucky failed us!");
```

```
// you can also provide a default value in case None was returned:
```

```
let message = feeling_lucky().unwrap_or("Not lucky :(".to_string());
```

```
// ? operator also works in functions that return Option:
```

```
let expanded_message = feeling_lucky()? + " Are you?";
```