Error Handling

SSP-RS — Safe System Programming (in Rust)

Samuel Tardieu Stefano Zacchiroli

2024-10-08

The billion dollar mistake

Null references

A *null reference* of type “reference to type T” is a pointer which does not point onto an object of type T but uses a special value representing the absence of an object of type T. A “reference to type T” and a “null reference to type T” use the same type, denoted as T * in C and C++.

Here is an example of a null reference n:

```c
#include <stddef.h>

int main() {
  char *s = "Hello, world!";
  char *n = NULL;   // n doesn't reference a valid string or char
}
```

A reference to char and a null reference to char both use the same type char *.

Null references as a way to signal an error

In C, some functions return NULL (a pointer with all bits set to 0) to represent a failure:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  char *mem = malloc(100000000000);  // Try to allocate 100GB
  if (mem == NULL) {
    fprintf(stderr, "Allocation of 100GB failed\n");
  }
}
```

It is common for functions returning a pointer to return a NULL reference when they fail.

Since NULL is defined to be (void *) 0, the test can be written more concisely:

```c
if (!mem) { … }
```

Null references as a way to end a list of pointers

In C, `NULL` is also an usual way for marking the end of an array of pointers (as the `NUL` character is used to mark the end of a string):

```c
#include <stdio.h>

int main(int argc, const char *argv[], const char *env[]) {
  (void) argc;  // Prevent warning about an unused variable
  (void) argv;  // Prevent warning about an unused variable
  printf("Environment variables:\n");
  for (const char * const *p = env; *p; p++)
    printf("  - %s\n", *p);
  printf("End of environment variables\n");
}
```

```
Environment variables:
  - SHELL=/bin/zsh
  - EDITOR=vim
  [...]
  - DISPLAY=:0.0
End of environment variables
```

Null references: Hoare's billion dollar mistake

It looks like null references are great. Why does Tony Hoare, who invented them, call them "my billion dollar mistake"?

Origin of the null reference

In 1965, Hoare added references to ALGOL W, an object oriented language, with type checking performed by the compiler. He "could not resist adding the null reference because it was so easy to implement" (Hoare, *Qcon conference*, London, 2009. See: lecture video).

Since then, "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" (Hoare, *op. cit.*).

Null references: the problem

When a programmer is given an object of type `T *`, they expect it to dereference into an object of type `T`. Since a null reference for this type is also an object of type `T *`, the compiler usually cannot check that `T *` denotes a non-null reference.

Let's look at the following example which displays the first and last names of a person:

```c
 1  #include <stdio.h>
 2
 3  typedef struct {
 4    char *first_name;
 5    char *last_name;
 6  } person_t;
 7
 8  void display_person(const person_t *person) {
 9    printf("%s %s\n", person->first_name, person->last_name);
10  }
```

On line 9, we can see that neither `person` nor `person->first_name` nor `person->last_name` are checked against `NULL`. The function `display_person` assumes that it is given a valid non-null reference containing no null references but nothing in its signature can reflect this.

Protecting against null references

What if we do not want to crash? Should we adopt defensive programming tactics?

```c
#include <stdio.h>

typedef struct {
  char *first_name;
  char *last_name;
} person_t;

void display_person(const person_t *person) {
  if (person && person->first_name && person->last_name)
    printf("%s %s\n", person->first_name, person->last_name);
}
```

Defensive programming allows the execution to proceed without crashing but does not report the error in any way (it *swallows* the error). It may also produce an incorrect output (a missing line in this case).

Defensive programming is rarely consistent

This is a trimmed-down excerpt from a large Java Android application:

```java
public static final class VariablesListAdapter extends […] {
    private VariableList variables;

    public void sortVariables(final Comparator<String> comparator) {
        if (comparator == null) {        // Why would we call this with a null comparator?
            return;
        }
        sortItems((v1, v2) -> comparator.compare(v1.getVar(), v2.getVar()));
        if (variables != null) {         // Ok, so sometimes the variables field can be null
            variables.sortVariables(comparator);
        }
        callCallback();
    }

    public boolean containsVariable(final String var) {
        return variables.contains(var);  // Couldn't the variables field be null?
    }
```

Programmers have to agree on how null references are used (spoiler: they don’t agree).

How can we avoid null references problems?

There are several ways to deal with null references.

Enrich the type system with annotations

Java allows annotating an object type with the `@NonNull` annotation which is later used by IDE and external tools:

- An input parameter or a field annotated with `@NonNull` will be assumed to be non-null.
- A function result or a field annotated with `@NonNull` can only receive something provably (or declared) non-null.

Python also supports tagging parameters and variables with a type hint, and `None` (the equivalent of `null`) must be included explicitly if allowed (`T | None` or `Optional[T]`).

Prevent null references from existing

In Rust, any reference designates an existing, live object. However, there exists a way to denote the absence of an object.

Interlude: the Rust way of dealing with an unexpected situation

The default behaviour for a Rust program entering an error state is to *panic*. The current thread will be stopped, and possibly the whole program if nothing special is done to intercept the panic.

A user can choose to explicitly panic with an error message:

```rust
fn main() {
  let n = std::env::args().len() - 1;
  if n != 3 {
    // We do not know what else to do if we don't have 3 extra arguments. Time to panic.
    panic!("This program must be called with three extra arguments; got {n} arguments instead");
  }
}
```

A crash in an unexpected situation is often better than going on with the execution while being in an unknown state.

Rust `Option` type

The `Option<T>` type represents either the presence of a value of type `T`, or the absence of such a value. It is defined as an enumerated type:

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

- `Option::None` represents an absence of value
- `Option::Some(42)` represents the presence of an integer whose value is 42
- `Option::Some(&s)` represents the presence of a reference on an object `s`

Note

`Option`, `Some`, and `None` are imported into the default namespace and can be used directly in Rust omitting the `Option::` prefix.

Getting the content of an Option

The unwrap() method returns the object contained in an Option, or panics if there is no value:

```rust
// Function returning the name of the user in a Some, or None if unknown
fn current_user() -> Option<String> { … }

fn main() {
  let s: String = current_user().unwrap();  // Panics if current_user() returns None
  println!("The current user is {s}");      // The reference is necessarily valid (a.k.a. not NULL)
} // Memory for the String in s is deallocated here
```

Compare this with the C implementation, where checking for NULL is not mandatory:

```c
char *current_user() { return …; }        // May return NULL if the user cannot be found

int main() {
  char *s = current_user();
  printf("The current user is %s\n", s);  // May display garbage, or crash
} // May leak memory here if the result of current_user() was allocated dynamically
```

In Rust, Option<String> and String are two different, incompatible types.

Checking an `Option` without panicking

Of course it is possible to extract the content of an `Option` without panicking even if it contains `None`, by using pattern matching:

```rust
fn main() {
  match current_user() {   // current_user() returns an Option<String>
    Some(u) => println!("The current user is {u}"),
    None    => println!("I could not determine who the current user is"),
  }
}
```

It is also possible to give a default value to use if the `Option` contains `None`:

```rust
fn main() {
  println!("The current user is {}",
           current_user().unwrap_or(String::from("<unknown user>")));
}
```

In both cases, we were not able to blindly use our `Option<String>` as a `String`.

More about pattern-matching

A `match` construct must cover every possible variant and deconstructs the `Option`:

```rust
1 fn main() {
2   let user: Option<String> = current_user();
3   match user {
4     Some(u) => println!("The current user is {u}"), // Memory for the String in u is freed here
5     None    => println!("I could not determine who the current user is"),
6   }
7   // Variable user is no longer usable here, it has been deconstructed by the match
8 }
```

On line 4, `u` is unified with the string contained in `Some(…)` and now *owns* the `String` that was previously owned by the `Option`. When the `match` clause ends, `u` goes out of scope and the `String` destructor is called to reclaim the heap memory.

Other useful methods on `Option` are `is_some()` and `is_none()`, both returning booleans, or `expect("message")` which acts like `unwrap()` but gives a better context message when panicking.

More precise error codes

NULL/None is not the only way to signal an error

Functions in the C standard library uses the integer value `-1` combined with the global variable `errno` to signal errors in a more precise way:

```c
#include <fcntl.h>

int open(const char *pathname, int flags[, mode_t mode]);
```

The `open` function returns:

- A non-negative integer containing the file descriptor to use with `read()`, `write()`, `close()`, etc. in case of success.
- `-1` in case of error. The global variable `errno` describes the cause of the error:
  - `EACCESS` (13 on a Linux system): access denied
  - `EINVAL` (22): invalid flags were provided
  - `ENOMEM` (12): insufficient kernel memory available
  - …

Again, the same type is used

The same type (`int`) is used to return either a successful answer (the file descriptor) or an error (`-1`). The following code will not trigger any warning at compilation time:

```c
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  int fd = open("/etc/motd", O_RDONLY);
  char buffer[16];
  int n = read(fd, buffer, sizeof buffer - 1);
  buffer[n] = '\0';  // Terminate the string in buffer before printing it
  printf("Beginning of the file: %s\n", buffer);
}
```

This code does not check the return value of `open`, nor the return value of `read`. If `read` returns `-1`, we write a `NUL` character *before* the beginning of `buffer`.

Note: do not confuse `NULL` and `NUL`

`NULL` (null pointer, of type `void *`) != `NUL` (0, of type `char`, for terminating a string)

Is this even detected at runtime?

```c
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  int fd = open("/etc/motd", O_RDONLY);
  char buffer[16];
  int n = read(fd, buffer, sizeof buffer - 1);
  buffer[n] = '\0';  // Terminate the string in buffer before printing it
  printf("Beginning of the file: %s\n", buffer);
}
```

The program compiles cleanly (`-Wall -Wextra`) and executes even though `open()` fails:

```
$ ./t
Beginning of the file:
$ valgrind ./t
==2859820== Warning: invalid file descriptor -1 in syscall read()
==2859820== Conditional jump or move depends on uninitialised value(s)
==2859820==    at 0x4847D09: strlen (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2859820==    by 0x48EA81E: printf (printf.c:33)
==2859820==    by 0x1090BB: main (in ./t)
```

Interlude: errno and multithreaded programs

Originally, errno was a local variable of type int. How has it been adapted to a multi-threading environment without breaking backward compatibility with existing code?

A thread-local variable local_errno holds the error for the current thread (every thread gets its own variable). errno is a macro which transparently dereferences this variable:

```c
#define errno (*__errno_location())    // In errno.h from the standard library

int *__errno_location() {              // Return the address of the current thread's local_errno
  static __thread int local_errno;     // local_errno is a thread-local variable
  return &local_errno;
}
```

It can then be used as the regular errno variable could:

```c
#include <errno.h>
#include <stdio.h>

int main() {
  errno = 3;                           // Translates to (*errno_location()) = 3
  printf("errno = %d\n", errno);       // Will print "errno = 3"
}
```

Exceptions

What are exceptions?

An exception is an error condition which is propagated up the calling stack until it lands in a piece of code willing to handle it. Let’s illustrate this with a Python example.

```python
def x():
    try:
        print("In x() before y()")
        y()
        print("In x() after y()")
    except Exception as e:
        print(f"Caught exception in x: {e}")

def y():
    print("In y() before z()")
    z()
    print("In y() after z()")

def z():
    print("In z()")
    raise Exception("exception raised from z")
    print("At the end of z()")

print("In main program before x()")
x()
print("In main program after x()")
```

prints, while executed:

```
In main program before x()
In x() before y()
In y() before z()
In z()
Caught exception in x: exception raised from z
In main program after x()
```

The exception raised in `z()` has been immediately propagated up to `x()` without executing the end of `z()` or the end of `y()`.

A bit of exceptions terminology

- An exception is said to be *raised* (Python, Ada) or *thrown* (C++, Java, Kotlin), depending on the programming language.
- An exception is said to be *caught* when its propagation is stopped.
- A `finally` piece of code might get executed in any case whether an exception is raised or not (Python, Java, Kotlin).

In the following Python example, the exception is not caught and is propagated up to the top-level which causes the program to stop with an error. However, the `finally` block is executed before the propagation.

```python
try:
    print("Before raise")
    raise Exception("uncaught exception")
    print("After raise")
finally:
    print("In finally block")

print("After try block")   # Not executed
```

```
$ python t.py
Before raise
In finally block
Traceback (most recent call last):
  File "/tmp/t.py", line 3, in <module>
    raise Exception("uncaught exception")
Exception: uncaught exception
```

Finally and RAII

Even when a language does not implement `finally`, the use of RAII (*resource acquisition is initialization*) will run the object destructor before propagating an exception up. Here is a C++ example:

```
1  void my_func() {
2    lock_t lock = wait_for_lock();    // Get lock on critical section
3    […] // Do some things while locked
4    if (problem_detected())
5      throw "this is an exception because something went wrong";  // C++ exceptions can have any type
6    […]
7    lock.unlock(); // Unlock the lock
8    […]             // Do other things that don't require the lock
9  }  // The destructor of "lock" will unlock the resource before propagating the exception up
```

If `problem_detected()` returns `true`, lines 6 to 8 will not get executed. However, the destructor of `lock` will be called and may, if this is the required behaviour, unlock the lock so that other threads can access it.

Note: this is not always a good idea to unlock a lock in RAII mode as the system might be in an inconsistent state. Some implementations (*e.g.,* Rust) implement *lock poisoning*.

Exceptions were not created equal

Depending on the language and the implementation, exceptions may be used for exceptional situations or as regular flow control tools:

- In Python, exceptions are often used as an easy way to return early several levels up and do not necessarily denote an error condition.
- In most languages, exceptions should be reserved to uncommon situations: not raising an exception does not cost much, while raising an exception can take a much longer time.

Why does “not raising an exception” have a cost at all?

- Efficient implementations for exceptions use tables stored along the program code with the code span covered by every exception handler and the type of exception to catch.
- When an exception occurs, the call stack is unwind (getting up one function at a time) until a table matches the current instruction pointer.
- All function frames must have the same recognizable format in order to unwind the call stack. This may require constraining the code generator and optimizer a bit.

Checked and unchecked exceptions

In most languages supporting exceptions, any function is free to raise any exception. However in Java, some exceptions are considered “checked exceptions” (as opposed to “unchecked exceptions”) because the compiler will check that:

- either we handle exceptions when they happen (for example `FileNotFoundException`) with a `try`/`catch` block,
- or we let them be propagated up, but we have to declare it in our function signature.

If we do none of those two things, the program will not compile. Checked exceptions:

- force programmers to document which exception can be raised by each function;
- do not allow exceptions to propagate up indefinitely unless they are declared on the main program signature;
- feel like a burden if all we want to do is have the program crash when such an exception happens.

Unchecked exceptions on the other hand can be raised at any time and propagated as high as necessary in the call stack.

Example of a checked exception

In the following Java example which prints the content of the “/etc/motd” file, any input/output error would cause an exception derived from `IOException` to be raised. Since `IOException` is a checked exception, it has to be declared on `main()` signature as it is not handled locally:

```java
import java.io.*;

class Example {
    public static void main(String[] args)
      throws IOException  // Mandatory, otherwise it will not compile
    {
        BufferedReader file = new BufferedReader(new FileReader("/etc/motd"));
        String line;
        // Print the content of the file, line by line
        while ((line = file.readLine()) != null) {
          System.out.println(line);
        }
        file.close();
    }
}
```

The Rust way of handling errors

How does Rust handle errors?

- Rust uses different types to represent different things. `Option<T>` is not the same type as `T`. Rust does not use integers as C does with `open()` to represent both successful completion and error conditions.
- Rust does not use exceptions, but it forces the programmer to declare which type represents a successful operation and which type represents an error.
- Rust does not let the programmer ignore an error easily.

Introducing Result<T, E>

In addition to `Option<T>`, Rust has a standard `Result<T, E>` ↗ enumerated value:

```
pub enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

A `Result<T, E>` can either be the `Ok` variant with a value of type `T`, or the `Err` variant which represents an error with a value of type `E` describing the error.

As for an `Option`, the `unwrap()` method either returns the content of the `Ok` variant or panics while displaying the content of the `Err` variant. The `expect("message")` method allows giving a better panic message.

Reading a file in Rust

We have to use `unwrap()` on the result of `File::open()` (which returns a `Result<File, std::io::Error>`) to get a `File` object `fd`. Same for `read_to_string()` (which returns a `Result<usize, std::io::Error>`):

```rust
1  use std::{fs::File, io::Read};
2
3  fn main() {
4    let mut fd = File::open("/etc/motd").expect("cannot open file");
5    let mut s = String::new();
6    fd.read_to_string(&mut s).expect("cannot read file");
7    println!("The content of the file is {s}");
8  }
```

The program stops by panicking at line 4 because `"/etc/motd"` was not found on this system:

```
$ target/debug/t
thread 'main' panicked at src/main.rs:4:49
'cannot open file: Os { code: 2, kind: NotFound, message: "No such file or directory" }'
```

Contrary to the C version, it did not try to read the content of the non-existent file.

Returning the error

If we do not want to deal with errors in our function, we can return prematurely when we encounter one:

```rust
use std::{fs::File, io::Read};

fn main() -> Result<(), std::io::Error> {
  let mut fd = match File::open("/etc/motd") {
    Ok(x) => x,
    Err(e) => return Err(e),
  };
  let mut s = String::new();
  if let Err(e) = fd.read_to_string(&mut s) {  // Only execute this code if we have an Err variant
    return Err(e);
  }
  println!("The content of the file is {s}");
  Ok(())
}
```

If `main()` returns an `Err` variant, it gets displayed and a non-zero status code is returned to the shell:

```
$ target/debug/t
Error: Os { code: 2, kind: NotFound, message: "No such file or directory" }
$ echo $?
1
```

Getting more concise

The pattern to extract the value from an `Ok` variant or prematurely returning the error from the `Err` variant is so common that a one-character postfix operator exists in Rust to do that:

```rust
match a {
  Ok(x) => x,
  Err(e) => return Err(e),
}
```

can be simply written as `a?`. Our code can be shortened as:

```rust
use std::{fs::File, io::Read};

fn main() -> Result<(), std::io::Error> {
  let mut fd = File::open("/etc/motd")?;
  let mut s = String::new();
  fd.read_to_string(&mut s)?;
  println!("The content of the file is {s}");
  Ok(())
}
```

Is that all?

No, we can get even more concise by chaining the calls:

```
1  use std::{fs::File, io::Read};
2
3  fn main() -> Result<(), std::io::Error> {
4    let mut s = String::new();
5    File::open("/etc/motd")?.read_to_string(&mut s)?;    // Note the use of two '?' in this line
6    println!("The content of the file is {s}");
7    Ok(())
8  }
```

Neat, eh?

Also note that `File` is a Rust type which has a destructor: when the `File` goes out of scope, its destructor calls `close()` automatically so that the file descriptor is freed and our process can reuse it. Here, assuming `File::open()` has returned a `Ok` variant, this would happen at the end of line 5 because no variable owns the `File` returned by `File::open()` after this line.

What if we forget to check the Result?

The read_to_string() method on a File returns a Result<usize, std::io::Error>. Since we never use the usize, what happens if we do not check the result?

```rust
use std::{fs::File, io::Read};

fn main() -> Result<(), std::io::Error> {
  let mut s = String::new();
  File::open("/etc/motd")?.read_to_string(&mut s);     // The latest '?' has been removed
  println!("The content of the file is {s}");
  Ok(())
}
```

The compiler warns us about it:

```
--> src/main.rs:5:3
5 |   File::open("/etc/motd")?.read_to_string(&mut s);
  |   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  = note: this `Result` may be an `Err` variant, which should be handled
```

Nothing catastrophic will happen if we do not check it as the string s will be empty but still valid. Nevertheless the compiler wants to ensure that we did not just forget to check for an error.

How does the compiler know we didn't check the Result?

The Result type is annotated with a must_use attribute. Because of this, a conversion of a Result<T, E> to () (equivalent of void in C) will trigger the warning:

```
 1  #[must_use = "this `Result` may be an `Err` variant, which should be handled"]
 2  pub enum Result<T, E> { Ok(T), Err(E), }
 3
 4  fn f() -> Result<i32, String> { … }
 5
 6  fn main() {
 7    f();             // Warning, we are converting a Result to () because of the ';'
 8    f().unwrap();    // No warning
 9    let x = f();     // No warning, we store the Result in the x variable
10    x;               // Warning (double warning even, because "x" alone never does anything)
11    let y = f();     // Warning because the y variable itself is unused
12    let _ = f();     // No warning, we explicitly choose to ignore the Result here
13  }
```

Rust makes it hard to inadvertently not check the content of a Result but does not forbid us from doing so, as illustrated on line 12.

How can we return more than one error type?

Using Rust `enum`, it is easy to create a custom error type which encapsulates all possible sources of errors we want to propagate:

```rust
pub enum MyError {
  Io(std::io::Error),                // Encapsulate an I/O error
  Parsing(std::num::ParseIntError),  // Encapsulate an integer to string parsing error
}

fn read_int_from_file(f: &str) -> Result<i32, MyError> {
  let mut file = match std::fs::File::open(f) {
    Ok(x) => x,
    Err(e) => return Err(MyError::Io(e)),
  };
  // Read content of file and convert it to i32? If an I/O
  // error occurs, return an Err with MyError::Io, if the
  // content of the file is not an integer, return an Err
  // with MyError::Parsing and the error inside.
  todo!()
}
```

Converting between error types

Although we haven't studied *traits* yet, here is how it is possible to write a conversion function which let us encapsulate a `std::io::Error` into our own `MyError` type:

```rust
pub enum MyError {
  Io(std::io::Error),                // Encapsulate an I/O error
  Parsing(std::num::ParseIntError),  // Encapsulate an integer to string parsing error
}

impl From<std::io::Error> for MyError {
  fn from(e: std::io::Error) -> MyError {
    MyError::Io(e)
  }
}

fn read_int_from_file(f: &str) -> Result<i32, MyError> {
  let mut file = match std::fs::File::open(f) {
    Ok(x) => x,
    Err(e) => return Err(From::from(e)),   // Make a MyError using our implementation above
  };
  […]
}
```

We can do even better

We have described the `a?` construct as being equivalent to

```
match a {
  Ok(x) => x,
  Err(e) => return Err(e),
}
```

It is a bit more powerful than that, and is in fact akin to:

```
match a {
  Ok(x) => x,
  Err(e) => return Err(From::from(e)),
}
```

It will convert the error into the excepted error type if a `From::from()` method has been implemented.

Converting to oneself

`From<T>` is implemented for any type `T`, so a `MyError` instance can be converted to a `MyError` (by doing nothing).

Rewriting the example

Knowing that the `From` conversion will be used if available, we can rewrite our example:

```rust
use std::io::Read;                   // To be able to use `.read_to_string()` on a file

pub enum MyError {
  Io(std::io::Error),                // Encapsulate an I/O error
  Parsing(std::num::ParseIntError),  // Encapsulate an integer to string parsing error
}

impl From<std::io::Error> for MyError {
  fn from(e: std::io::Error) -> MyError { MyError::Io(e) }
}

impl From<std::num::ParseIntError> for MyError {
  fn from(e: std::num::ParseIntError) -> MyError { MyError::Parsing(e) }
}

fn read_int_from_file(f: &str) -> Result<i32, MyError> {
  let mut s = String::new();
  std::fs::File::open(f)?.read_to_string(&mut s)?;
  Ok(s.parse()?)  // Will automatically select the right return type (i32) for parse()
}
```

Using our function

We can extract the error from the result of our function:

```rust
fn main() {
  match read_int_from_file("/tmp/test.txt") {
    Ok(n)                    => println!("I have read the number {n}"),
    Err(MyError::Io(e))      => println!("I/O error: {e:?}"),
    Err(MyError::Parsing(e)) => println!("Parsing error: {e:?}"),
  }
}
```

or we can use the _ pattern (which matches anything) if we do not care about the details of the error:

```rust
fn main() {
  match read_int_from_file("/tmp/test.txt") {
    Ok(n)  => println!("I have read the number {n}"),
    Err(_) => println!("I was unsuccessful in reading a number from the file"),
  }
}
```

Conclusion

- With its `Option<T>` type, Rust does not let the programmer treat an absent value as if it was present. This is a step-up compared to all languages with null references, whose type are indistinguishable from the type of a valid reference.
- With its `Result<T, E>` type, Rust does not let the programmer ignore errors unless they explicitely intend to do so.
- With `enum`, `From` and `?`, Rust can propagate any error up the call chain by defining new types to encapsulate the various error types, without requiring exceptions.