Rust Error Handling

PRESENTED BY:

Dr. Jim Fawcett

Teaching Professor Emeritus of Computer Engineering at Syracuse University

MODERATED BY:

Steve Warzala

2020-06-15

Cyber Security & Information Systems Information Analysis Center

CSIAC is sponsored by the Defense Technical Information Center (DTIC) under Contract #FA8075-17-D-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DTIC.

info@csiac.org
https://www.csiac.org

Rust Error Handling

Important Part of Safe Systems Programming

Jim Fawcett

https://JimFawcett.github.io

What is Rust?

- A modern programming language that emphasizes:
  - Compiler verified freedom from undefined behavior.
  - Support for writing data race free code in multi-threaded environments.
  - Performance comparable to C and C++.
- Rust builds on experience with C and C++ to provide a system programming language that:
  - Closes vulnerabilities by construction.
  - Does not pay run-time performance penalty for safety.
  - Enables abstractions needed to build large maintainable code bases.
- Rust compiles to native code and provides clever mechanisms to ensure freedom from dangling references and resource leaks.

Safe and Productive System Programming

- Rust incorporates a number of interesting ideas that support modern system programming:
  - Compiler-checked **single ownership** policy with transfer and borrowing operations.
  - Heap storage only through **smart pointers** that use scope-based data management, so no resource leaks.
  - **Traits** that define contracts for static and dynamic polymorphism, using generics and trait inheritance, essential for building flexible code that adapts to changing requirements.
  - Error handling: functions return **Result<T, E> { Ok(t:T), Err(e:E) }** enumerations.
  - **Dependency management** through metadata supported transitive builds.
  - **Effective tools** for building (cargo), checking code quality (clippy), formatting (rustfmt), and documentation (rustdoc).

Error Handling

- This presentation focuses on Rust error handling.
  - Role of panics in preventing undefined behavior
  - Returning results from functions that may fail
  - Matching return enumerations with appropriate operations
  - Error event bubbling up call chain
- We will demonstrate these with discussions and code. You can find all code presented here in this Error Handling Code Repository.
- You will find more details about ownership, objects, generics, and the Rust build process in a series of podcasts that are being published by CSIAC and also made available here.
- More details about Rust are provided in a Rust Story from my github site.

What's Unique about Rust Error Handling?

- Rust identifies functions that may fail by returning Result<T, E>
- Rust encourages developers to handle every case were errors may occur.

```
30      print!("\n  Please enter some text: ");
31      //let _ = std::io::stdout().flush();
32      std::io::stdout().flush();
33      let rslt = std::io::stdin().read_line(&mut s);
```

```
cargo -q run
warning: unused `std::result::Result` that must be used
  --> src\main.rs:32:5
   |
32 |     std::io::stdout().flush();
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(unused_must_use)]` on by default
   = note: this `Result` may be an `Err` variant, which should be handled
```

- You have to opt out if you don't need to handle an error case.

```
30      print!("\n  Please enter some text: ");
31      let _ = std::io::stdout().flush();
32      //std::io::stdout().flush();
33      let rslt = std::io::stdin().read_line(&mut s);
```

- Rust has support for bubbling errors up the call chain, creating new custom errors, and returning errors from main.

Errors

- Indexing out of bounds
- Divide by zero
- Integer overflow
- Console and file I/O failures to open or read/write
- Initializing String from non-utf8 byte array
- System and User-defined errors
  - Users supply unexpected or malicious inputs
  - Server not available
  - Unexpected content format

Rust Panics

- A panic is a program exit that attempts to unwind the stack, dropping each object residing in the stack.
  - Panics can be trapped and handled to avoid program exit
- Should a panic occur while unwinding the stack from an earlier panic the program will immediately abort.
  - Multiple panic aborts cannot be trapped, so stopping in this case is inevitable
- Panics are intended program actions that avoid undefined behavior due to program errors.
  - Indexing out of bounds
- So panics are the lowest level of error handling mechanisms.

Avoiding Undefined Behavior with Panic

C++ Code

```
47      std::cout << "\n  Demo of Undefined Behavior - out of bounds index";
48      std::cout << "\n ----------------------------------------------
49
50      int array[3]{ 1, 2, 3 };
51      std::cout << "\n  ";
52      for (size_t i = 0; i <= 3; ++i) {
53          std::cout << array[i] << " ";
54      }
55      std::cout << std::endl;
```

Unowned memory can be accessed. Process ends normally.

```
 Demo of Undefined Behavior - out of bounds index
 ------------------------------------------------
 1 2 3 -858993460

C:\github\JimFawcett\RustModels\Video_1_Introduction\UndefinedB
ehavior\Debug\UndefinedBehavior.exe (process 44608) exited with
 code 0.
Press any key to close this window . . .
```

Rust Code

```
1   use std::io::*;
2
3   fn main() {
4       let array = [1, 2, 3];
5       print!("\n  ");
6       for i in 0..4 {
7           let _ = std::io::stdout().flush();
8           print!("{} ", array[i]);
9       }
10      println!("\n  That's all Folks!\n");
11  }
12
```

```
C:\github\JimFawcett\RustErrorHandling\IndexOutOfBounds>
cargo -q run

  1 2 3 thread 'main' panicked at 'index out of bounds: the l
en is 3 but the index is 3', src\main.rs:8:23
note: run with `RUST_BACKTRACE=1` environment variable to dis
play a backtrace
C:\github\JimFawcett\RustErrorHandling\IndexOutOfBounds>
```

Panic terminates before memory can be accessed

Avoiding Panics

- If your program affects user health, wealth, or safety, then don’t panic.
  - Thank you, Douglas Adams
- Abruptly terminating operation of a Boeing 797 flight navigation system is not a good idea.
- The route to panic-free behavior is handling results of all functions that may fail.
  - Rust makes that obvious, using the return type Result<T, E> for functions that may fail.
  - Rust vigorously reminds you to add any missing error handling for those functions – see slide #2.

Rust Error Handling Types

- Enum Result<T,E> { Ok(T), Err(E) }
  - #[must_use]
  - std crate import
- pub fn is_ok(&self) -> bool
- pub fn is_Err(&self) -> bool
- pub fn unwrap(self) -> T
  - panics if not Ok
- Pub fn unwrap_err(self) -> E
  - Panics if not Err

Error Types

- Std Error
  - type Result<T> = Result<T,std::io::Error>;
  - std::io crate import
- Custom Error
  - Use std::io::{Error, ErrorKind};
  - Let custom_error =
    Error::new(ErrorKind::Other, some_useful_value);

Using Result<T, E> with is_ok()

```
36      print!("\n  --- testing output string ---");
37      let _valid = vec![0x61, 0x62, 0x63];
38      let _invalid = vec![0xED, 0xA0, 0x80];
39      let arg = _invalid;
40      /*-- to see both cases try _valid and _invalid --*/
41      let _result;
42      let cvt_str_rslt = String::from_utf8(arg);
43      if cvt_str_rslt.is_ok() {
44          let s:String = cvt_str_rslt.unwrap();
45          let bytes = s.as_bytes();
46          std::io::stdout().write_all(b"\n  writing: ")?;
47          std::io::stdout().write_all(&bytes)?;
48          _result = Ok(());
49      }
50      else {
51          let error = cvt_str_rslt.unwrap_err();
52          print!("\n  {}", error);
53          _result = Err(std::io::Error::new(ErrorKind::Other, "error"));
54      }
```

Illustrates accepting Result<String, FromUtf8Error>, Testing, with Result::is_ok(), and returning a new Result type: Result<(), CustomError>

Trapping Panics

- If you need to use code that doesn’t reliably avoid panics you may attempt to trap them:
  - `trap_panic(unreliable_function, description_str) -> Result;`
  - Definition in next slide

- Traps are not guaranteed to succeed.
  - A panic unwinds the call stack, returning resources to the process with drop.
  - If a drop also panics, the system will immediately abort.
  - If that happens before leaving trap_panic the trap will fail.

Trapping Panics

```rust
/*-- traps panic, execution continues --*/
#[allow(dead_code)]
fn trap_panic(f:fn(), name:&str) -> std::io::Result<()> {
    let default_hook = panic::take_hook();
    set_panic_hook();
    let rslt = panic::catch_unwind(|| { f() });
    panic::set_hook(default_hook);
    match rslt {
        Ok(()) => {
            Ok(())
        }
        Err(_) => {
            let arg = format!("{:?} panic", name);
            let error = std::io::Error::new(ErrorKind::Other, arg);
            Err(error)
        }
    }
}
/*-----------------------------------------------
   traps panic, execution continues
   - takes function with return value
   - supply input arguments with closure
     - see example at end of main
*/
#[allow(dead_code)]
fn trap_panic_return<F: FnOnce() -> R + UnwindSafe, R>(f:F, name:&str) -> std::io::Result<R>
        where R:Debug + Clone {
    let default_hook = panic::take_hook();
    set_panic_hook();
    let rslt = panic::catch_unwind(|| -> R { f() });
    panic::set_hook(default_hook);
    match &rslt {
        Ok(r) => {
            Ok(r.clone())
        },
        Err(_) => {
            let arg = format!("{:?} panic", name);
            let error = std::io::Error::new(ErrorKind::Other, arg);
            Err(error)
        }
    }
}
/*-- elides default panic message --*/
```

```rust
fn main() {
    print!("\n  {}","-- testing panics --");
    let _ = std::io::stdout().flush();
    // do_panic();
    // let r = trap_panic(do_panic, "do_panic()");
    // show_result(r);
    // index_out_of_bounds();
    let r = trap_panic(index_out_of_bounds, "index_out_of_bounds()");
    show_result(r);
    // divide_by_zero();
    // let r = trap_panic(divide_by_zero, "divide_by_zero()");
    // show_result(r);
    // integer_overflow();
    // let r = trap_panic(integer_overflow, "integer_overflow");
    // show_result(r);
    // initialize_str_with_non_utf8();
    // let fp = initialize_str_with_non_utf8;
    // let r = trap_panic(fp, "initialize_str_with_non_utf8");
    // show_result(r);

    // ------------------------------------------
    // trap panic for string to int conversion
    // uses return computed value
    // ------------------------------------------
    let s = String::from("-3");
    // let s = String::from("-3.5");
    let l = || -> i32 { convert_string_to_int(&s) };
    let name = "convert_string_to_int";
    print!("\n  ");
    let rslt = trap_panic_return(l, name);
    if rslt.is_ok() {
        print!("\n  {:?}\n  returned {}", name, rslt.unwrap());
    }
    else {
        print!("\n  {}", rslt.unwrap_err());
    }
    //-------------------------------------------
    println!("\n\n  That's all Folks!\n");
}
```

Error Handling that Avoids Panics

- Each function that can fail should return a std::result::Result<T, E>
  - `fn f<T, E>() -> Result<T, E> { /* code that can fail */ }`
  - std library functions do this and so should user-defined functions
- Result is an enumeration
  - `Enum Result<T, E> { Ok(T), Err(E), }`
  - Returned Result instance is either `Ok(t:T)` or `Err(e:E)`
    - t is the computed value of f() or unit, (), if no such value is computed
    - e is the instance of error encountered, either from Error enumeration or user-defined
- Testing Result
  - `let rslt = f();`
  - `if rslt.is_ok() { let t:T = rslt.unwrap();  /* do something with t */ }`
  - `if rslt.is_err() { let e:E = rslt.unwrap_err();  /* do something with e */ }`

Evaluating Result by Matching

- `let rslt = f();`
- ```
  match rslt {
       Ok(t) => { /* do something with t */ },
       Err(e) => { /* do something with e */ },
  }
  ```
- match is required to define actions for both possible results
- “if let” uses matching operator =
- ```
  if let Ok(t) = rslt {
       /* do something with t = rslt.unwrap() */
  }
  else {
       /* do something with e = rslt.unwrap_err() */
  }
  ```

Demonstration code using match and if let

```
51      /*-- uses match --*/
52      let rslt = always_fails();
53      print!("\n\n  using match:");
54      match rslt {
55          Ok(()) => print!("\n  function always_fails succeeded!\n"),
56          Err(error) => {
57              print!("\n  function always_fails failed");
58              print!("\n    - error message: {:?}\n", error.msg)
59          }
60      }
61      let _ = std::io::stdout().flush();
62
63      /*-- uses if let --*/
64      let _rslt = always_fails();
65      print!("\n  using if let:");
66      /*--  "=" is match operator, not assignment --*/
67      if let Ok(()) = _rslt {
68          print!("\n  function always_fails succeeded");
69      }
70      else {
71          let error:CustomError = _rslt.unwrap_err();
72          print!("\n  function always_fails failed with message:\n  {:#?}", error.msg);
73      }
74      let _ = std::io::stdout().flush();
```

match requires testing both cases, Ok and Err

if let doesn't require handling both cases, but the code may do so, as shown

Bubbling Errors up Call Chain

- `fn g<T, E>() -> Result<T, E>`
- `fn f<T, E>() -> Result<T, E> {`

```
    // code elided
    let t:T = g()?
    // code using t elided
}
```

- If g() returns an error the try operator ? returns from f(), passing out the Result object, Err(e:E).
- Otherwise, the ? operator unwraps the result, t:T and binds to t.

Bubbling Errors up the Call Chain

```
28  #[allow(dead_code)]
29  fn always_fails() -> std::result::Result<(),CustomError> {
30      let error = CustomError::new("failure test");
31      Err(error)  // return error
32  }
33  #[allow(dead_code)]
34  fn always_succeeds() -> std::result::Result<(),CustomError> {
35      Ok(())      // return unit result
36  }
```

```
76      /*-- uses try operator ? to bubble up error --*/
77      print!("\n\n  using try operator ?\n");
78      always_fails()?;
79
80      println!("\n\n  That's all
81      Ok(())
82  }
```

f<T>() -> Result<T, E>

if Result<T, E> contains Ok(t:T) after evaluating f()
then f()? Evaluates as t = f().unwrap();
if Result<T, E> contains Err(error)
then f()? Returns Result<T, E> to caller

Examples of Common Error Handling

- Console I/O
- File I/O
- TCP communication processing
- Inter-process communication with pipes

- We will briefly discuss the first two in this presentation

Console I/O

- std::io::stdin() -> Stdin
- Stdin functions:
  - fn read_line(&self, buf:&mut String) -> Result<usize>
  - fn read_to_string(&mut self, buf:&mut String) -> Result<usize>
  - Many more here: https://doc.rust-lang.org/1.4.0/std/io/struct.Stdin.html
- std::io::stdout() -> Stdout
- Stdout functions
  - fn write(&mut self, buf: &[u8]) -> Result<usize>
  - fn write_all(&mut self, buf: &[u8]) -> Result<usize>
  - fn flush(&mut self) -> Result<()>
  - Many more here: https://doc.rust-lang.org/1.4.0/std/io/struct.Stdout.html

Console I/O – std::io::stdin()

```
26
27      /*-- reading from stdin --*/
28      let mut s=String::new();
29      use std::io::*;
30      print!("\n  Please enter some text: ");
31      let _ = std::io::stdout().flush();
32      let rslt = std::io::stdin().read_line(&mut s);
33      match rslt {
34          Ok(bytes) => {
35              strip_newline(&mut s);
36              print!("\n  you typed {:?} using {} bytes\n", s, bytes);
37          },
38          Err(error) => print!("\n  your input failed with error: {:?}\n", error),
39      }
40
```

Console I/O – std::io::stdout()

```
42      print!("\n  --- testing output string ---");
43      let _valid = vec![0x61, 0x62, 0x63];
44      let _invalid = vec![0xED, 0xA0, 0x80];
45      let arg = _invalid;
46      /*-- to see both cases try _valid and _invalid --*/
47      let _result;
48      let cvt_str_rslt = String::from_utf8(arg);
49      if cvt_str_rslt.is_ok() {
50          let s:String = cvt_str_rslt.unwrap();
51          let bytes = s.as_bytes();
52          std::io::stdout().write_all(b"\n  writing: ")?;
53          std::io::stdout().write_all(&bytes)?;
54          _result = Ok(());
55      }
56      else {
57          let error = cvt_str_rslt.unwrap_err();
58          print!("\n  {}", error);
59          _result = Err(std::io::Error::new(ErrorKind::Other, "console write error"));
60      }
```

arg = _valid

```
--- testing output string ---
writing: abc
```

arg = _invalid

```
--- testing output string ---
invalid utf-8 sequence of 1 bytes
from index 0
```

```
61      ////////////////////////////////////////////////////////
62      // Using _invalid in code below panics at write_all,
63      // never returns Result.
64      //--------------------------------------------------------
65      print!("\n\n  --- testing write result ---\n");
66      let _valid = &[0x61, 0x62, 0x63];    // valid utf-8 byte sequence
67      let _invalid = &[0xED, 0xA0, 0x80];  // invalid utf-8 byte sequence
68      std::io::stdout().write(b"\n  writing: ")?;
69      let arg = _valid;
70      // setting arg = _invalid
71      // results in untrappable panic, e.g., panic while
72      // panicing
73      ////////////////////////////////////////////////////////
74      // The code below traps panics in Rust code, but
75      // apparently not when calling into foreign code,
76      // like Windows console.
77      //--------------------------------------------------------
78      // let _result = panic::catch_unwind(
79      //     || -> std::io::Result<()> {
80      //         {
81      //             std::io::stdout().write_all(arg)
82      //         }
83      //     }
84      // );
85      let _result = std::io::stdout().write_all(arg);
86      if _result.is_err() {
87          let error = _result.unwrap_err();
88          print!("\n  could not write invalid, error: {:?}", error);
89      }
90      else {
91          print!("\n  wrote {:?}", arg);
92      }
93
```

std::io::stdout()

Stdout() on Windows platform does not work well with non-utf8 characters. If you pass a buffer containing non-utf8 byte sequence(s) the program will panic.

Moreover, that panic cannot be trapped because the stack unwinding process results in a second active panic which always calls an immediate abort.

Note that you can **always avoid this problem** by building a String from the byte sequence, as shown in the previous slide. That does reliably fail with a Result if any of the bytes can't be represented as part of a utf-8 sequence.

If it doesn't fail, you can safely pass the String, as bytes, to the stdout().write or write_all methods.

File I/O

- std::fs::File
- File functions:
  - fn open<P: AsRef<Path>>(path: P) -> Result<File> // opens read-only
  - fn create<P: AsRef<Path>>(path: P) -> Result<File> // opens write-only
  - fn with_options() -> OpenOptions
  - Many more here: https://doc.rust-lang.org/std/fs/struct.File.html

Flexible File Open

```
6
7   #[allow(unused_imports)]
8   use std::fs::{File};
9   use std::io::prelude::*;
10
11  #[allow(dead_code)]
12  struct FileOption;
13  impl FileOption {
14      const CREATE:u8 = 1; const APPEND:u8 = 2;
15      const READ:u8 = 4; const WRITE:u8 = 8;
16  }
17
```

```
18
19  fn open_file(file_name:&str, opt: u8) -> std::io::Result<File> {
20      use std::fs::OpenOptions;
21      let mut f = OpenOptions::new();
22      type FO = FileOption;
23      if opt & FO::WRITE != 0 {
24          f.write(true);
25      }
26      if opt & FO::READ != 0 {
27          f.read(true);
28      }
29      if opt & FO::CREATE != 0 {
30          f.create(true);
31      }
32      if opt & FO::APPEND != 0 {
33          f.append(true);
34      }
35      f.open(file_name)
36  }
37
```

Syntax of the Rust language does not support bit-masking on enums (which you can do in C++). The reason is that Rust enums may have any associated type, not just integers (like C++). This code illustrates one way to accomplish bit masking on options.

File Error Handling

```
37
38  fn main() -> std::io::Result<()> {
39
40      let fn1 = "file1.txt";
41      type FO = FileOption;
42      let rslt = open_file(fn1, FO::WRITE | FO::CREATE | FO::APPEND);
43      if rslt.is_ok() {
44          let mut f1 = rslt.unwrap();
45          f1.write(b"abc")?;
46          print!("\n  open and write {:?} succeeded", fn1);
47      }
48      else {
49          print!("\n  open {:?} failed", fn1)
50      }
51
52      let fn2 = "does_not_exist.txt";
53      let rslt = open_file(fn2, FO::WRITE | FO::APPEND);
54      if rslt.is_ok() {
55          print!("\n  open {:?} no create succeeded", fn2);
56      }
57      else {
58          let error = rslt.unwrap_err();
59          print!("\n  error: {:#?} {:?}", error.kind(), fn2);
60      }
61      // https://blog.yoshuawuyts.com/error-handling-survey/
62
63      println!("\n\n  That's all Folks!\n");
64      Ok(())
65  }
66
```

Two cases are presented here. The first attempts to open a file, and, if it does not exist, will create and open it.

The second case does not attempt to create the file if it does not exist, so will fail if it doesn't exist.

Open errors are managed by examining the open_file function's result. Write failures are handled by bubbling up to the caller – main in this case, so a write error terminates the program with an error message.

Summary

- Rust error handling uses:
  - panics
    - Trapping panics has behavior similar to C++ exception handling
  - std::Result<T,E>
    - Must handle both Ok(t:T) and Err(e:E)
  - Matching
    - Equivalent to manually handling Result, but often less code
  - call-chain error event bubbling
    - Supports chaining calls, e.g., anInstance.f1()?.f2()?.f3()?;
    - Chaining requires each function to return self or &self
- Rust tries to prevent developers from ignoring errors or forgetting to manage them.

References

Link	Contents
Rust Error Handling Code	Demonstration code for this presentation
Rust Book	Covers most of the language clearly
Gentle Introduction to Rust	Well written, fewer topics than Rust Book, very clear
Half Hour to Learn Rust	Stroll throught most of the common constructs
Rust Standard Library	The official documentation for the std library

That’s all Folks!

Thanks for listening/reading