


# Exceptions, Maybe


Michael Feathers  
R7K Research & Conveyance

**NULL**



I call it my billion-dollar  
mistake. It was the invention  
of the null reference in 1965.

Tony Hoare


 quotefancy

```
if(ip != NULL)
    printf("%d\n", *ip);
```

```
if(ip != NULL)
    printf("%d\n", *ip);
```



```
if(ip != NULL)
    printf("%d\n", *ip);
```



```
int *ip = NULL;
```



```
if(ip != NULL)
    printf("%d\n", *ip);
```

- Actual Cause
- Point of Detection
- Point of Reaction



```
if(ip != NULL)
    printf("%d\n", *ip);
```



# A History of Mechanisms

# Error Return Values

```
1 /* strtok example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str[] = "- This, a sample string.";
8     char * pch;
9     printf ("Splitting string \"%s\" into tokens:\n",str);
10    pch = strtok (str, " ,.-");
11    while (pch != NULL)
12    {
13        printf ("%s\n",pch);
14        pch = strtok (NULL, " ,.-");
15    }
16    return 0;
17 }
```



# The Sentinel Problem

```
sqrt(x);
```

# The Sentinel Problem

```
sqrt(x);
```

*What can the error return be?*

# The Sentinel Problem

```
sqrt(x);
```

*What can the error return be?*

**33 of course!**

# errno

```
#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 */
```

---

**But.. what if we don't check errno?**



But.. what if we don't check errno?

*We'll make you!*

```
#include <iostream>
using namespace std;

int main()
{
    int x = -1;

    // Some code
    cout << "Before try \n";
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }

    cout << "After catch (Will be executed) \n";
    return 0;
}
```

---

This is a slightly modified version of an article that appeared in the November-December 1994 issue of the [C++ Report](#).

---

## Exception Handling: A False Sense of Security ☒

by [Tom Cargill](#) ☒

I suspect that most members of the C++ community vastly underestimate the skills needed to program with exceptions and therefore underestimate the true costs of their use. The popular belief is that exceptions provide a straightforward mechanism for adding reliable error handling to our programs. On the contrary, I see exceptions as a mechanism that may cause more ills than it cures. Without extraordinary care, the addition of exceptions to most software is likely to diminish overall reliability and impede the software development process. ☒

This "extraordinary care" demanded by exceptions originates in the subtle interactions among language features that can arise in exception handling. Counter-intuitively, the hard part of coding exceptions is not the explicit throws and catches. The really hard part of using exceptions is to write all the intervening code in such a way that an arbitrary exception can propagate from its throw site to its handler, arriving safely and without damaging other parts of the program along the way. ☒

The next operation on  $\tau$  we examine is the copy construction of the  $\tau$  object returned from `pop`:  $\square$

```
template <class T>
T Stack<T>::pop()
{
    if( top < 0 )
        throw "pop on empty stack";
    return v[top--];           // throw
}
```

What happens if the copy construction of this object throws an exception? The `pop` operation fails because the object at the top of the stack cannot be copied (not because the stack is empty). Clearly, the caller does not receive a  $\tau$  object. But what should happen to the state of the stack object on which a `pop` operation fails in this way? A simple policy would be that if an operation on a stack throws an exception, the state of the stack is unchanged. A caller that removes the exception's cause can then repeat the `pop` operation, perhaps successfully.  $\square$

## An Invitation ☐

Regular readers of this column might now expect to see a presentation of my version of `stack`. In this case, I have no code to offer, at least at present. Although I can see how to correct many of the faults in Reed's `stack`, I am not confident that I can produce an exception-correct version. Quite simply, I don't think that I understand all the exception-related interactions against which `stack` must defend itself. Rather, I invite Reed (or anyone else) to publish an exception-correct version of `stack`. This task involves more than just addressing the faults I have enumerated here, because I have chosen not to identify all the problems that I found in `stack`. This omission is intended to encourage others to think exhaustively about the issues, and perhaps uncover situations that I have missed. If I did offer all of my analysis, while there is no guarantee of its completeness, it might discourage others from looking further. I don't know for sure how many bugs must be corrected in `stack` to make it exception-correct. ☐

Exceptions are half of an error handling  
mechanism

```
return :: a -> Maybe a  
return x = Just x
```

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
(>>=) m g = case m of  
           Nothing -> Nothing  
           Just x   -> g x
```

# Monadic Error Handling



```
safeLog :: (Floating a, Ord a) => a -> Maybe a
safeLog x
  | x > 0      = Just (log x)
  | otherwise = Nothing
```

```
> safeLog 1000
Just 6.907755278982137
> safeLog -1000
Nothing
```

```
Optional<FileInputStream> fis =  
    names.stream().filter(name -> !isProcessedYet(name))  
        .findFirst()  
        .map(name -> new FileInputStream(name));
```

`main :: IO ()`



```
def flatMap[B](f: A => OptionT[F, B])(implicit F: Monad[F]): OptionT[F, B] =  
  OptionT(run.flatMap(  
    _.fold(Option.empty[B].point[F])(  
      f andThen ((fa: OptionT[F, B]) => fa.run))))
```

```
module CountEntriesT (listDirectory, countEntries) where

import CountEntries (listDirectory)
import System.Directory (doesDirectoryExist)
import System.FilePath ((</>))
import Control.Monad (forM_, when)
import Control.Monad.Trans (liftIO)
import Control.Monad.Writer (WriterT, tell)

countEntries :: FilePath -> WriterT [(FilePath, Int)] IO ()
countEntries path = do
  contents <- liftIO . listDirectory $ path
  tell [(path, length contents)]
  forM_ contents $ \name -> do
    let newName = path </> name
        isDir <- liftIO . doesDirectoryExist $ newName
    when isDir $ countEntries newName
```

Error handling is managing the path between detection and reaction in design.

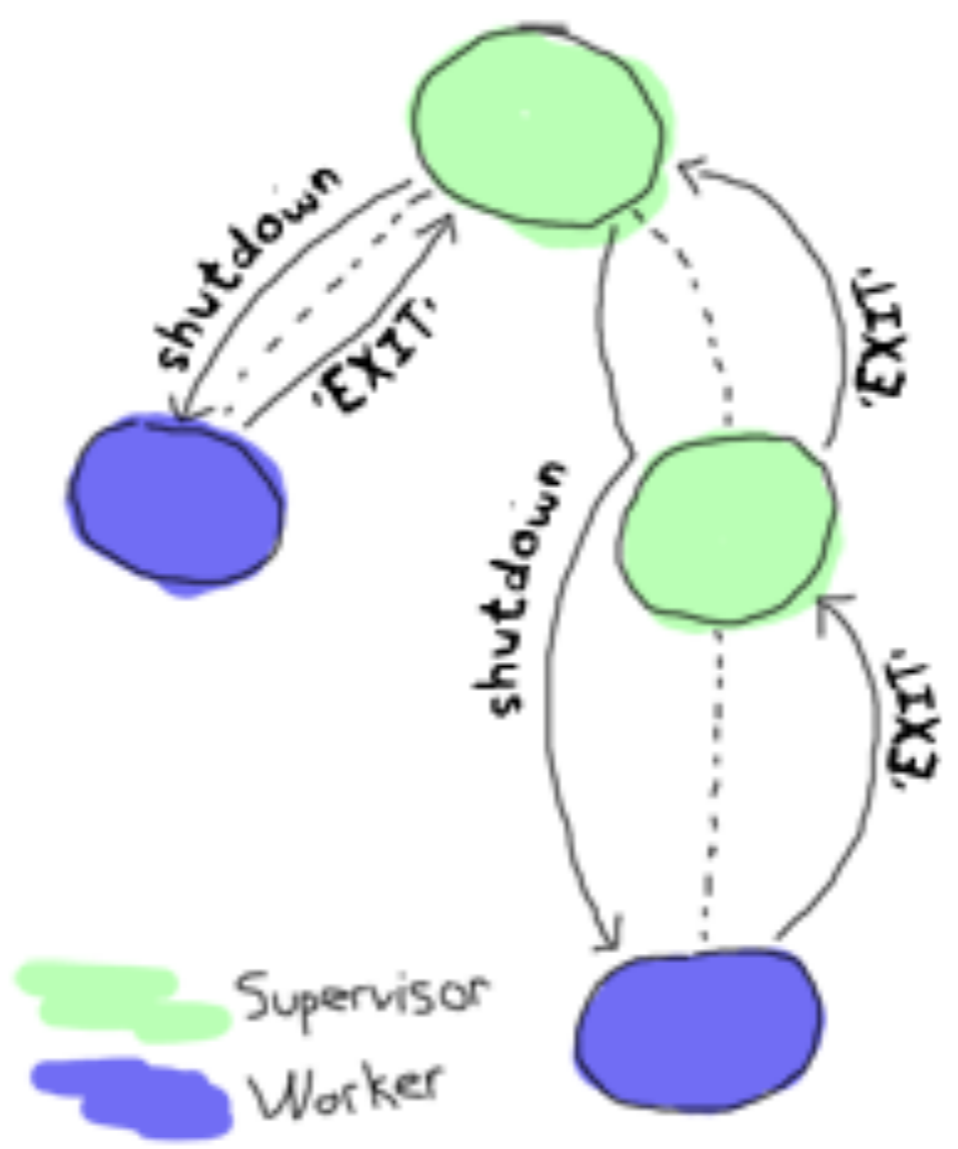
Error handling is managing the path between detection and reaction in design.



ERLANG

**KEEP  
CALM  
AND  
LET IT  
CRASH**



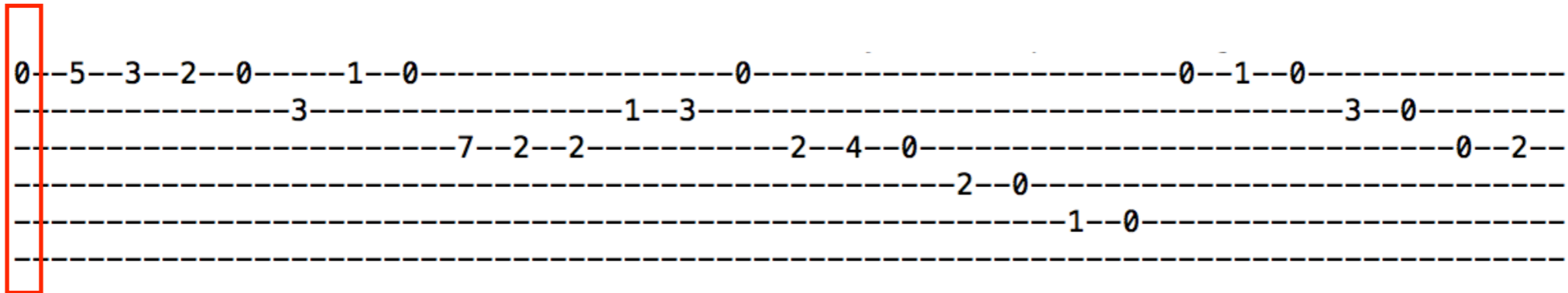


*Noticeable Error Handling is a Symptom of Bad Design*

0--5--3--2--0-----1--0-----0-----0--1--0-----  
-----3-----1--3-----3--0-----  
-----7--2--2-----2--4--0-----0--2--  
-----2--0-----  
-----1--0-----  
-----

0--5--3--2--0-----1--0-----0-----0--1--0-----  
-----3-----1--3-----3--0-----  
-----7--2--2-----2--4--0-----0--2--  
-----2--0-----  
-----1--0-----  
-----

1 0  
1 5  
1 3  
1 2  
1 0  
2 3  
1 1  
1 0  
3 7  
3 2  
3 2  
2 1  
2 3  
1 0  
3 2  
3 4  
3 0



1 0

1 5

1 3

1 2

1 0

2 3

1 1

1 0

3 7

3 2

3 2

2 1

1 0  
1 5  
1 3  
1 2  
1 0  
2 3  
1 1  
1 0  
3 7  
3 2  
3 2  
2 1  
2 3  
1 0  
3 2  
3 4  
3 0  
4 2  
4 0  
5 1  
5 0  
1 0  
1 1  
1 0  
2 3  
2 0  
3 0  
3 2

0. Command line argument for the filename may be missing
1. Unable to open an input file
2. File is empty
3. File contains empty lines
4. Our input file is not a text file
5. A line has more than two numbers
6. A line has less than two numbers
7. A line has fields that can not be parsed as numbers
8. The string number is less than one or more than six
9. The fret number is less than zero or more than twenty-four

```
STRING_COUNT = 6

def tab_column string, fret
  ["---"          ] * (string - 1) +
  [fret.ljust(3, '-')] +
  ["---"          ] * (STRING_COUNT - string)
end

unless File.exist? ARGV[0]
  abort "Unable to open #{ARGV[0]}"
end

File.open(ARGV[0], "r") do |f|
  puts f.each_line
    .map(&:split)
    .map {|string, fret| tab_column(string.to_i, fret) }
    .transpose
    .map(&:join)
    .join($/)
end
```

## Arithmetic Encoding

```
STRING_COUNT = 6
```

```
def tab_column string, fret
  ["---"          ] * (string - 1) +
  [fret.ljust(3, '-')] +
  ["---"          ] * (STRING_COUNT - string)
end
```

```
unless File.exist? ARGV[0]
  abort "Unable to open #{ARGV[0]}"
end
```

```
File.open(ARGV[0], "r") do |f|
  puts f.each_line
    .map(&:split)
    .map {|string, fret| tab_column(string.to_i, fret) }
    .transpose
    .map(&:join)
    .join($/)
end
```



```
STRING_COUNT = 6
```

```
def tab_column string, fret
  ["---"          ] * (string - 1) +
  [fret.ljust(3, '-')] +
  ["---"          ] * (STRING_COUNT - string)
end
```

```
unless File.exist? ARGV[0]
  abort "Unable to open #{ARGV[0]}"
end
```

← Hmmm..

```
File.open(ARGV[0], "r") do |f|
  puts f.each_line
    .map(&:split)
    .map {|string, fret| tab_column(string.to_i, fret) }
    .transpose
    .map(&:join)
    .join($/)
end
```

```
STRING_COUNT = 6
```

```
def tab_column string, fret
  ["---"          ] * (string - 1) +
  [fret.ljust(3, '-')] +
  ["---"          ] * (STRING_COUNT - string)
end
```

```
puts ARGF.each_line
  .map(&:split)
  .map {|string, fret| tab_column(string.to_i, fret) }
  .transpose
  .map(&:join)
  .join($/)
```

- X 0. Command line argument for the filename may be missing
- X 1. Unable to open an input file
- X 2. File is empty
- 3. File contains empty lines
- 4. Our input file is not a text file
- 5. A line has more than two numbers
- 6. A line has less than two numbers
- 7. A line has fields that can not be parsed as numbers
- 8. The string number is less than one or more than six
- 9. The fret number is less than zero or more than twenty-four

1  
1 5  
1 3  
1 2  
1  
2 3

# Domain Extension

```
#include<cmath.h>  
double sqrt (double x );
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

- X 0. Command line argument for the filename may be missing
- X 1. Unable to open an input file
- X 2. File is empty
- 3. File contains empty lines
- 4. Our input file is not a text file
- 5. A line has more than two numbers
- 6. A line has less than two numbers
- 7. A line has fields that can not be parsed as numbers
- 8. The string number is less than one or more than six
- 9. The fret number is less than zero or more than twenty-four



- X 0. Command line argument for the filename may be missing
- X 1. Unable to open an input file
- X 2. File is empty
- 3. File contains empty lines
- X 4. Our input file is not a text file
- 5. A line has more than two numbers
- 6. A line has less than two numbers
- 7. A line has fields that can not be parsed as numbers
- 8. The string number is less than one or more than six
- 9. The fret number is less than zero or more than twenty-four

```

STRING_COUNT = 6

def tab_column string, fret
  ["---"          ] * (string - 1) +
  [fret.ljust(3, '-')] +
  ["---"          ] * (STRING_COUNT - string)
end

lines = ARGF.each_line
      .select {|l| l =~ /\S/ }
      .map(&:split)

check("each line should have two fields") do |line_fields|
  line_fields.count == 2
end

check("all fields should be integers") do |string, fret|
  converts_to_int(string) && converts_to_int(fret)
end

check("strings should be in the range 1..6") do |string, _|
  string >= 1 && string <= 6
end

puts lines.each_line
      .map {|string, fret| tab_column(string.to_i, saturate(fret.to_i, (0..99))) }
      .transpose
      .map(&:join)
      .join($/)

```

0. Command line argument for the filename may be missing
1. Unable to open an input file
2. File is empty

**COMPLETED**

7. A line has notes that can not be parsed as numbers
8. The string number is less than one or more than six
9. The fret number is less than zero or more than twenty-four

```
STRING_COUNT = 6
```

```
def tab_column string, fret
  ["---"          ] * (string - 1) +
  [fret.ljust(3, '-')] +
  ["---"          ] * (STRING_COUNT - string)
end
```

```
lines = ARGF.each_line
      .select {|l| l =~ /\S/ }
      .map(&:split)
```

```
check("each line should have two fields") do |line_fields|
  line_fields.count == 2
end
```

```
check("all fields should be integers") do |string, fret|
  converts_to_int(string) && converts_to_int(fret)
end
```

```
check("strings should be in the range 1..6") do |string, _|
  string >= 1 && string <= 6
end
```

```
puts lines.map {|string, fret| tab_column(string.to_i, saturate(fret.to_i, (0..99))) }
      .transpose
      .map(&:join)
      .join($/)
```

```
STRING_COUNT = 6
```

```
def tab_column string, fret
  ["---"          ] * (string - 1) +
  [fret.ljust(3, '-')] +
  ["---"          ] * (STRING_COUNT - string)
end
```

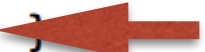
```
lines = ARGF.each_line
      .select {|l| l =~ /\S/ }
      .map(&:split)
```

```
check("each line should have two fields") do |line_fields|
  line_fields.count == 2
end
```

```
check("all fields should be integers") do |string, fret|
  converts_to_int(string) && converts_to_int(fret)
end
```

```
check("strings should be in the range 1..6") do |string, _|
  string >= 1 && string <= 6
end
```

```
puts lines.map {|string, fret| tab_column(string.to_i, saturate(fret.to_i, (0..99))) }
      .transpose
      .map(&:join)
      .join($/)
```



```
STRING_COUNT = 6
```

```
def tab_column string, fret
  ["---"          ] * (string - 1) +
  [fret.ljust(3, '-')] +
  ["---"          ] * (STRING_COUNT - string)
end
```

```
lines = ARGF.each_line
      .select {|l| l =~ /\S/ }
      .map(&:split)
```

```
check("each line should have two fields") do |line_fields|
  line_fields.count == 2
end
```

```
check("all fields should be integers") do |string, fret|
  converts_to_int(string) && converts_to_int(fret)
end
```

```
check("strings should be in the range 1..6") do |string, _|
  string >= 1 && string <= 6
end
```

```
puts lines.map {|string, fret| tab_column(string.to_i, saturate(fret.to_i, (0..99))) }
      .transpose
      .map(&:join)
      .join($/)
```

