Error handling in modern C++

Mikel Irazabal

12/11/2019

Content

Content

Introduction

- error: lat. error, -ōris.
- m. Acción desacertada o equivocada. - [RAE online]
- error: “an act that ... fails to achieve what should be done.” — [Merriam-Webster]
- Error: a function that couldn’t do what it advertised (i.e., its preconditions were met, but it could not achieve its successful-return postconditions, and the calling code can recover).
- Bug: state of the program not foreseen by the designer.
- A bug or abstract machine corruption is not an error (both are not programatically recoverable)
- This includes out-of-bound access and null dereference.
- A programmer is needed to recover from such a failure.

Content

Contracts

- Derive from Hoare logic.
- Term coined by Bertrand Meyer and made it available in Eiffel.
- Idea of having assertions on steriods.
- Define Pre-condition (expect), assert and post-condition (ensure) (e.g., invariants were left out)
- Define *audit* (expensive operation) and *axiom* (mostly for static analyzers)

Figure: Sign it!

Contracts

- "With contracts, 90-something% of the typical uses of exceptions in .NET and Java became preconditions. All of the ArgumentNullException, ArgumentOutOfRangeException, and related types and, more importantly, the manual checks and throws were gone." — [Duffy 2016]

Figure: Midori OS

Contracts

```
1
2  void f(int x, int y)
3    [[expects: x>0]]
4    [[expects: y!=0]]
5    [[ensures result: result > x+y]];
6
7
8  bool positive(const int * p) [[expects: p!=nullptr]] {
9    return *p > 0;
10 }
11
12 bool g(int * p) [[expects: positive(p)]];
```

Listing 1: Ex. from document P0542R5

Contracts

- Contracts didn't make it for C++20 :-(. Possibly for C++26.
- In any case it has been proof that restricting the sets that are passed to the functions reduces the number of bugs (e.g., prefer passing by const rather than by *, or don't pass int if the number cannot by negative).

Content

Error Codes

- Available in C.
- Different approaches can be applied.
  - Global variable (e.g., errno)
  - Return value (e.g., open)
  - Singletons (e.g.,)

Figure: From the ancient times...

Error Codes

- Set errno to 0
- Check if the value changed

```
1
2  int main()
3  {
4      // set errno to 0
5      errno = 0;
6      id_t calling_process = 0;
7      int rc = getpriority(PRIO_PROCESS, calling_process);
8      if(errno != 0)
9          printf("Error getting the priority. Message: %s \n",
       strerror(errno) );
10
11     return 0;
12 }
```

Listing 2: errno example 1

Error Codes

- Check for return value and then errno

```
1
2  int main()
3  {
4      const char* file_path= "/tmp/foo.txt";
5      int fd = open(file_path, O_APPEND );
6      if(fd < 0)
7          printf("Error opening the file %s with the message: %s \n",
       file_path, strerror(errno) );
8      return 0;
9  }
10
```

Listing 3: errno example 1

Error Codes

- Check for return value. The used value to indicate an error, may also be a valid value (e.g., 0 value is used to mark an error).

```
1
2  const char *str1 = "42";
3  const char *str2 = "3.14159";
4  const char *str3 = "BcnCppMeetup";
5  const char *str4 = "0";
6
7  int num1 = std::atoi(str1);
8  int num2 = std::atoi(str2);
9  int num3 = std::atoi(str3); \\ Error. num3 = 0.
10 int num4 = std::atoi(str4); \\ Error or valid input?
11
12 std::cout << "std::atoi(\"" << str1 << "\") is " << num1 << '\n';
13 std::cout << "std::atoi(\"" << str2 << "\") is " << num2 << '\n';
14 std::cout << "std::atoi(\"" << str3 << "\") is " << num3 << '\n';
15 std::cout << "std::atoi(\"" << str4 << "\") is " << num4 << '\n';
```

Listing 4: atoi

- Check if the return value is not equal to zero. errno is not changed on failure!

```
1  const size_t SIMD_BIT_ALIGN = 512;
2  const size_t size = 1000;
3  void* ptr;
4  int rc = posix_memalign( &ptr, SIMD_BIT_ALIGN, size);
5  if(rc != 0){
6      switch(rc): // note that errno doesn't change
7          case EINVAL:{
8          printf("The alignment argument was not a power of two, or
       was not a multiple of sizeof(void *)\n");
9          break;
10         }
11         case ENOMEM:{
12         printf("There was insufficient memory to fulfill the
       allocation request\n" );
13         break;
14         }
15         default:
16             assert(0 != 0 && Flow cannot get here..)
```

Listing 5: posix_memalign example

Error codes

- Possible memory leaks...

```
1
2  int main()
3  {
4      int* p = malloc(1000*sizeof(int));
5      if(p == NULL){}
6        printf("Error allocating memory");
7        return 0;
8      }
9
10      // very important code
11
12     free(p); // can we guarantee that all the paths pass through
       the free?
13     return 0;
14 }
```

Listing 6: malloc example

Error codes

- Maybe a valid idiom in C for resource management, use RAII in C++

```
1     int* p = malloc(1000*sizeof(int));
2     if(p == NULL){
3       goto exit;
4     }
5     if(!foo())
6         goto free_func;
7     if(!bar())
8         goto free_func;
9     if(!baz())
10        goto free_func;
11
12    free(p);
13    return 1;
14free_func:
15     free(p);
16exit:
17    return 0;
```

Listing 7: malloc example

Error Codes

- And what about singletons for managing errors!!!!
- Composition of errors is not trivial

```
int main()
{
    GLenum rc = glGetError();
    if(rc != GL_NO_ERROR){
        switch (rc):
            case GL_INVALID_ENUM: {
                printf("Invalid enum passed!");
                break;
            }
            /*
            .
            */
    }
     // more important code
    return 0;
}
```

Listing 8: OpenGL example

Error Codes

- Implementation
  - errno and perror are thread safe according to the C11 standard
  - strerror is not thread safe. Use strerror_r as alternative
  - Out-parameters enforce a memory layout which is not optimizer friendly, basically the return value that it is stored in a register unusable for normal flow code
- Conceptual
  - A lot of noise added
  - Separating error path from normal flow is difficult
  - Tedious to maintain consistently such an approach
  - Postponing error-handling requires the programmer to take into account the error-code through the call-graph.
  - Basically if an error is detected, a task cancellation should happen.
  - Visible flow

Error Codes

- There was at least one continuous bug in the previous slides, can someone point it out?

Figure: Ouchhhhhhhh!

Error Codes

- Yes, printf also returns a value to inform the caller about the sucess or failure... Easy to forget checking for all the return values.
- C++17 added an attribute to generate a warning if the caller ignores the return value. [[nodiscard]]
- If a function declared nodiscard or a function returning an enumeration or class declared nodiscard by value is called from a discarded-value expression other than a cast to void, the compiler is encouraged to issue a warning.

Error Codes

- Since C++11 we can integrate the error codes into the language.
- It is composed of an integer and a pointer to the domain where the error happened.

```
1
2 struct error_code
3 {
4    private:
5      int                  _M_value // error number;
6      const error_category*   _M_cat // domain where the error was
     originated;
7 }
```

Listing 9: std::error_code

- std::error_category is intended to be used with the typical OOP.

```
1  class error_category
2  {
3      public:
4        constexpr error_category() noexcept = default;
5        virtual ~error_category();
6        error_category(const error_category&) = delete;
7        error_category& operator=(const error_category&) = delete;
8
9        // category name
10       virtual const char* name() const noexcept = 0;
11       // error message
12       virtual string  message(int) const = 0;
13       // maps error_code to error_condition
14       virtual error_condition default_error_condition(int __i)
     const noexcept;
15
16        virtual bool equivalent(int __i, const error_condition& __cond
     ) const noexcept;
17 }
```

Listing 10: std::error_category

Error Codes

- Implementation

```
1
2   // Don't do this!! It exists std::errc !!
3
4   enum class socket_err_c {
5     no_error = 0,    // 0 should not represent an error
6     invalid_addr = 1,
7     unknown_proto = 2,
8     invalid_flags = 3,
9     process_fd_limit = 4,
10    system_fd_limit = 5,
11    insufficient_memory = 6,
12    proto_not_supported = 7,
13  };
14  // Yes, we have to pollute the std namespace
15  namespace std
16  {
17    template <> struct is_error_code_enum<socket_err_c> : true_type
18    { };
19  }
```

Listing 11: std::error_category

Error Codes

```
1  class socket_err_c_category : public std::error_category  {
2   public:
3     // Return a short descriptive name for the category
4     const char *name() const noexcept final { return "Socket
   error defined by you!"; }
5     std::string message(int c) const final
6     {
7       switch (static_cast<socket_err_c>(c)) {
8       case socket_err_c::no_error:
9         return "conversion successful";
10      case socket_err_c::invalid_addr:
11        return "converting empty string";
12      case socket_err_c::unknown_proto:
13        return "unknown proto";
14      case socket_err_c::invalid_flags:
15        return "invalid flags";
16      case socket_err_c:: process_fd_limit:
17        return "process file descriptors limit reached";
18      default:
19        return std::to_string(c) + std::string(" code unknown");
20      }
21    }
22 };
```

Error Codes

```
1
2  // Declare a singleton
3   socket_err_c_category& get_socket_err_c_category()
4   {
5     static socket_err_c_category ec;
6     return ec;
7   }
8   // Overload the global make_error_code() free function with our
      custom enum.
9   inline std::error_code make_error_code(socket_err_c ec)
10  {
11    return {static_cast<int>(ec), get_socket_err_c_category()};
12 }
```

Listing 13: std::error_category

Error Codes

```
1
2    // function call
3   std::error_code open_socket()
4   {
5     // try to open the socket without success...
6     return make_error_code(socket_err_c::unknown_proto) ;
7   }
8
9   int main()
10  {
11    auto ec = open_socket();
12    assert(ec == socket_err_c::no_error);
13    return 0;
14  }
```

Listing 14: std::error_category

Error Codes

- It was adopted from the boost::error_code
- It is supposed to be used at some point by the Networking library
- Nowadays used aT the std::filesystem
  - bool remove(const std::filesystem::path p);
  - bool remove(const std::filesystem::path p, std::error_code ec) noexcept;

Content

Exceptions or interrupts

- Exceptions may happen in different domains
  - Hardware exceptions: IEEE 754 floating point exception ex. dividing by zero
  - OS signals. The OS may inform processes via IPC about different behaviours. A handler is registered and the OS can call it. ex. *int*\$0*x*80, SIGKILL.
  - Software exceptions: The term exception is typically used in a specific sense to denote a data structure storing information about an exceptional condition

Figure: Exceptions are exceptional!

Exceptions

- In the 80s and with the hope of improving the complexity of canceling errors, a new paradigm arrived in C++ : exceptions
- It has to be noted that C++ can only detect failures through exceptions at implicitly declared member functions (e.g., constructor, destructor...).
- One of the first papers, was written in 89 by Andrew Koenig and Bjarne Stroustrup and they describe how exceptions should be implemented. They discard the resumption ability: *termination is preferred over resumption; this is not a matter of opinion but a matter of years of experience. Resumption is seductive, but not valid. Bjarne Stroustrup*
- A failure can happen in the library but the library doesn't know how to handle it, but the customer of that library does. Ex. bad_alloc

Exceptions

```
int  foo(int  size) {
void  *ptr  =  malloc(size);
if  (ptr  ==  0)
  throw  NoMoreMemory();
}
int  bar(int  elem) {
  Resource  res;
  auto  result  =  foo(2  *  (res+elem) );
  /*  Do  something  else  */
}
int  main() {
  try  {
      for  (auto i = 0;  i < 100; ++i)
          bar(i);
  }catch  (const NoMoreMemory& excep)  {
      std::cerr  <<  Out  of  memory  " << '/n';
  }  catch  (...)  {/*  Report  other  problems.  */}}
}
```

Listing 15: Exceptions usage

Basic guarantee, Strong guarantee and noexcept guarantee

- Good at centralizing error handling mechanism
- They propagate the error up through the stack. Intermediate functions do not have to add code to manage the exception and destructors are called as expected.
- The exception is identified by a specific entity and not by a special return value, and therefore, exceptions liberate the programmer from the hassle of the failure call-graph. Exception code path can be clearly identified.
- There are two basic implementations of exceptions
  - List based (gcc's flag –RTS=sjlj in Ada)
  - Table based (gcc's flag –RTS=zcx in Ada)

List based exceptions

- Typically, based on setjmp/longjmp functions
- The setjmp function saves the actual context in a jmp_buf structure
- The longjmp is used to perform a non-local goto.
- The try/catch block is replaced with setjmp call and the throw with longjmp.
- A linked-list of jmp_buf structure represents the deepness in the stack.
- Problem: Destruction of the objects. A linked-list with the objects to be destroyed needs to be maintained.

List based exceptions

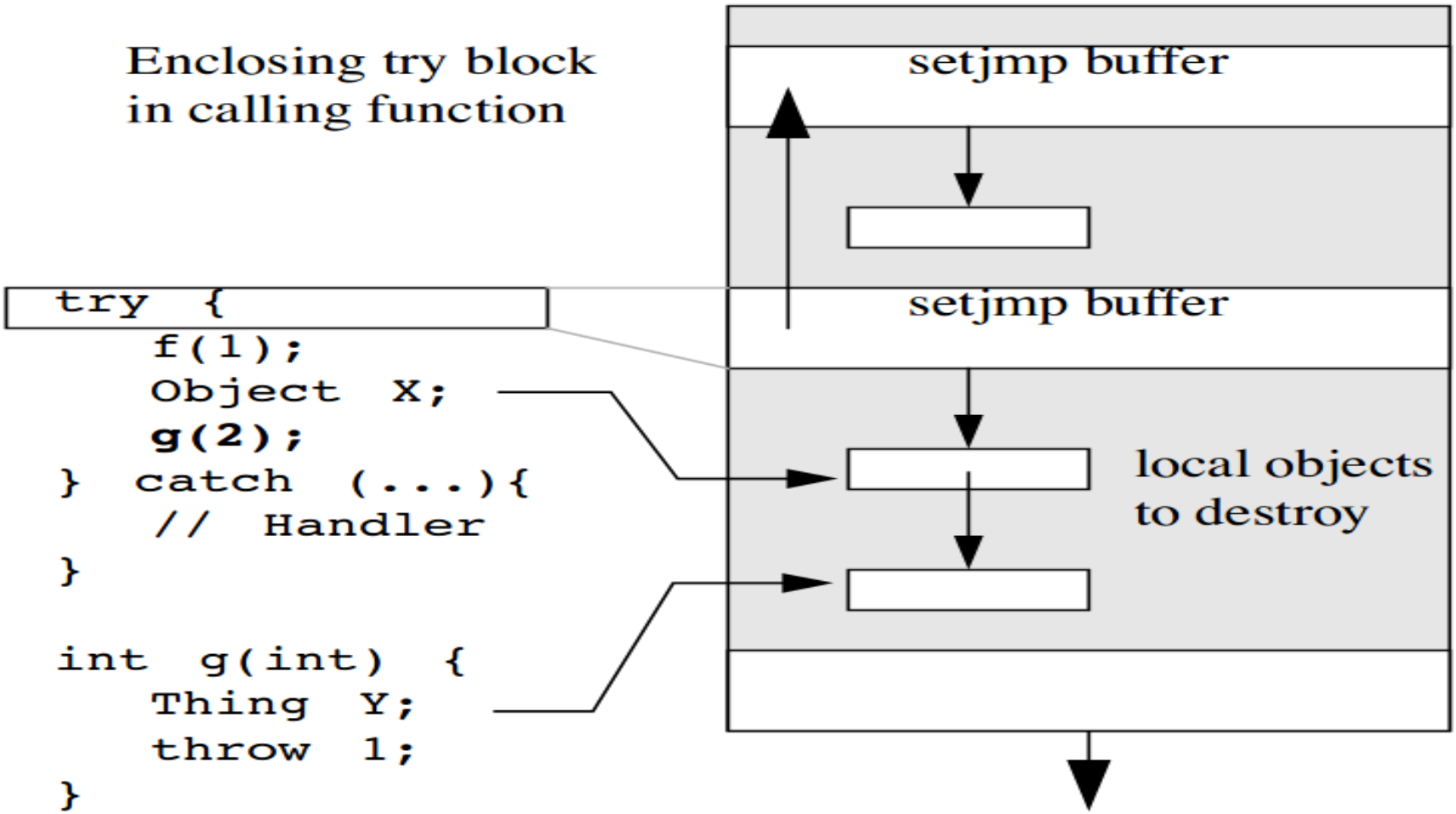

Figure: Setlongjmp exceptions implementation.

List based exceptions

- Additionally, performance drawbacks are significant
  - The setjmp function must be called at every beginning of all the functions, and the jmp_buf structure must be maintained.
  - Maintain a list of the object that we may need to destroy
  - Save all the variables values outside the try block. This can be a significance overhead, in architectures with large number of registers. Ex. RISC processors.

Table-driven exceptions

- A table is used to map the value of the program counter (PC) at the point where the exception is thrown.
- An action table is used to perform various operations (e.g., invoking the destructors, adjusting the stack or matching the exception thrown with the exception handler, returning the address of the handler).
- E.g., Call the destructor for object X at stack offset Y.
- Control is transferred to the handler once the PC is refreshed.

Table-driven exceptions

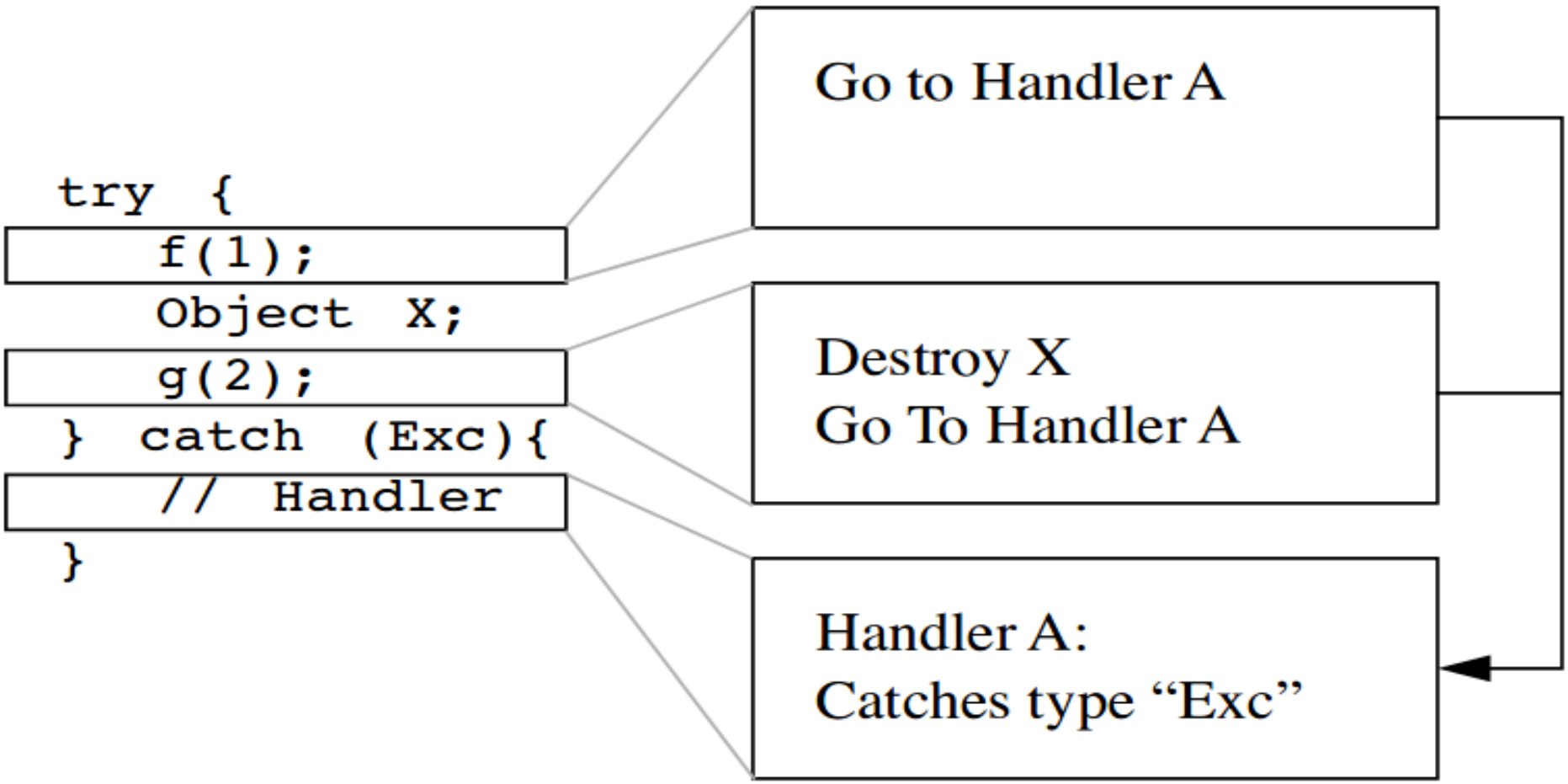

Figure: Table-driven implementation.

Table-driven exceptions

- The compilers translates the throw statement into the calls __cxa_allocate_exception and __cxa_throw, that allocates the exception and starts the unwinding process
- For every catch statement an entry in the exception table and a cleanup table is created.
- When the exception is thrown, the unwinder goes through the exception table (through a function called personality), until it finds a landing pad.
- If no landing path is found for the exception, std::terminate is called.
- If a landing path is found, the unwinder now starts again on the top of the stack.

Table-driven exceptions

- The unwinder goes through the stack a second time and it will call the personality routine to perform a cleanup.
- The personality routine will call the cleanup table. This will call the destructors of the objects allocated.
- The unwinder jumps in the stack into the proper catch statement.
- The memory acquired for the exceptions will be called.

Table-driven exceptions

- Performance penalties
  - The runtime still needs to restore the values of the variables that where declared outside the try block.
  - The destructors of the objects needs to be called.
  - The tables have a non-negligible size. They can be stored in the initialized read-only area.
  - No additional cost for the normal path.

Exceptions

```
1  extern int bar(int)
2  //noexcept
3  ;
4
5  int foo() noexcept
6  {
7      int i = 42;
8      int j = 21;
9      return bar(i+j);
10 }
```

Listing 16: Clang 9.0 C++17 -O3

Exceptions

```
1  foo(): # @foo()
2    push rax
3    mov edi, 63
4    call bar(int)
5    pop rcx
6    ret
7    mov rdi, rax
8    call __clang_call_terminate
9  __clang_call_terminate: # @__clang_call_terminate
10   push rax
11   call __cxa_begin_catch
12   call std::terminate()
```

Listing 17: Clang 9.0 C++17 -O3

Exceptions

```
1  extern int bar(int)
2  noexcept
3  ;
4
5  int foo() noexcept
6  {
7      int i = 42;
8      int j = 21;
9      return bar(i+j);
10 }
```

Listing 18: Clang 9.0 C++17 -O3

Exceptions

```
1 foo(): # @foo()
2   mov edi, 63
3   jmp bar(int) # TAILCALL
```

Listing 19: Clang 9.0 C++17 -O3

Exceptions don't have a zero cost!!!

Figure: Exceptions don't have zero cost!

- As seen exceptions don't have a zero cost abstraction. They do have a cost in space and in time, even if they are not used!
- If the code is not using exceptions, make sure that the -fno-exceptions is enabled in the compiler.

Excpetions: the call graph is difficult to follow...

- How many execution paths are in the following code?

```
1
2  String EvaluateSalaryAndReturnName( Employee e )
3    {
4      if( e.Title() == "CEO" || e.Salary() > 100000 )
5      {
6        cout << e.First() << " " << e.Last()
7             << " is overpaid" << endl;
8      }
9      return e.First() + " " + e.Last();
10   }
11
12
13
```

Listing 20: From Herb Sutter's Exceptional C++

Exceptions: Basic guarantee, Strong guarantee and noexcept guarantee

- Why we cannot get and object in one operation from a stack? Ex. stack top() and pop() functions.
  - $void\ std :: stack < T, Container >:: pop()$
  - $reference\ std :: stack < T, Container >:: top();$
  - $const\_reference\ std :: stack < T, Container >:: top()const;$
- In node based containers, the *extract* member function has been added ex. no more find erase needed in map. It is also the only way to take a move-only object.

Exceptions

```
1  int main() {
2      std::map<int, std::string> m{{1, "Knuth"}, {2, "Hoare"}, {3,
     "Stroustrup"}};
3      for(const auto& n : m)
4          std::cout <<  n.first << " " << n.second << std::endl;
5      auto nh = m.extract(2);
6      nh.key() = 4;
7      m.insert(std::move(nh));
8      for(const auto& n : m)
9          std::cout <<  n.first << " " << n.second << std::endl;
10 }
```

Listing 21: Extract from a container

Exceptions: Conclusion

- They are general approach to handle errors
- They are designed with centralized handling in mind.
- They can easily add arbitrary amount of information.
- "Little" cost on normal path.
- Hard to write exception code correctly, specially libraries
- Based on type based, and therefore on RTTI =¿ non-deterministic in time, requires dynamic allocation.
- Considerable cost in size
- Banned in many coding style standards (e.g., Google)
- "I can't recommend exceptions for hard real time; doing so is a research problem, which I expect to be solved within the decade [Stroustrup 2004]".
- Invisible flow

Content

Expected: Genesis

Figure: Alexander Alexandrescu.

- In 2012, Alexandrescu gave a talk - Systematic Error Handling in C++ - at C++ and Beyond.
- There he discussed the advantages of a Algebraic data type.

Expected: How do other languages manage errors?

- Rust: It has two type of errors. Recoverable errors and non-recoverable errors.
- $Result < T, E >$ for recoverable errors and the panic! macro that stops execution when the program encounters an unrecoverable error.

Expected: Rust

```rust

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

fn main() {
    let f = std::fs::File::open("hello.txt");
    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("Problem opening the file: {:?}", error)
        },
    };
}
```

Listing 22: Pattern matching in Rust

Expected: Rust error handling

```
1
2  use std::io;
3  use std::io::Read;
4  use std::fs::File;
5
6  fn read_username_from_file() -> Result<String, io::Error> {
7      let mut s = String::new();
8
9      File::open("hello.txt")?.read_to_string(&mut s)?;
10
11     Ok(s)
12 }
```

Listing 23: Shortcut for errors in Rust

Expected: Do we have something similar in C++?

- No. Yes. Maybe. Nowadays in C++ just std::optional exists.
- A std::expected class is expected to come in C++23 (first draft in 2014 N4015. Last P0323 in 2019)
- It is a Algebraic data type that returns either the expected value or the error.
- Boost.Outcome is a very similar class designed for such a purpose. Best available option if something like std::expected is needed today.

Expected from P0323

```
1
2  template<class T, class E>
3  class expected {
4  public:
5      using value_type = T;
6      using error_type = E;
7      using unexpected_type = unexpected<E>;
8
9  // 4.6, observers
10     constexpr const T* operator->() const;
11     constexpr T* operator->();
12     constexpr const T& operator*() const&;
13     constexpr T& operator*() &;
14     constexpr const T&& operator*() const&&;
15     constexpr T&& operator*() &&;
16     constexpr explicit operator bool() const noexcept;
17     constexpr bool has_value() const noexcept;
18     constexpr const T& value() const&;
19     constexpr T& value() &;
20     constexpr const T&& value() const&&;
21     constexpr T&& value() &&;
22
23 };
```

Expected

```
1
2       constexpr const E& error() const&;
3       constexpr E& error() &;
4       constexpr const E&& error() const&&;
5       constexpr E&& error() &&;
6       template<class U>
7           constexpr T value_or(U&&) const&;
8       template<class U>
9           constexpr T value_or(U&&) &&;
10
11  private:
12      bool has_val; // exposition only
13      union
14      {
15          value_type val; // exposition only
16          unexpected_type unex; // exposition only
17      };
18  };
```

Listing 25: Expected class from P0323

Expected: arithmetic function

```
1
2  enum class arithmetic_errc{
3  divide_by_zero = 1,
4  not_integer_division = 2,
5  integer_divide_overflows = 3,
6  }
7
8  expected<int, arithmetic_errc> safe_divide(int i, int j){
9  if(j == 0)
10     return unexpected(aritmethic_errc::divide_by_zero);
11 if(i == std::numeric_limits<int>::min() && j == -1)
12     return unexpected(aritmethic_errc::integer_divide_overflows);
13 if(i%j != 0)
14     return unexpected(arithmetic_errc::not_integer_division);
15 return i /j;
16 }
```

Listing 26: Ex. from P0323

Expected

```
1
2 expected<double, arithmetic_errc> f( double i, int j, int k){
3     auto q = safe_divide(j,k);
4     if(!q) return unexpected(q.error());
5     return i + q.value(); // i + *q;
6 }
```

Listing 27: Error handling with expected..

Excepted: Conclusion

- Locally to handle errors
- They can easily add arbitrary amount of information (No two exceptions can run on the same time)
- Still tedious to write it correctly
- Cheap, as the value is passed on the stack.
- Deterministics.
- Visible flow.

Expected: Monads

```
1
2     std::expected<image,fail_reason> get_cute_cat (const image& img
      ) {
3     auto cropped = crop_to_cat(img);
4     if (!cropped) {
5       return cropped;
6     }
7
8     auto with_tie = add_bow_tie(*cropped);
9     if (!with_tie) {
10      return with_tie;
11    }
12
13    auto with_sparkles = make_eyes_sparkle(*with_tie);
14    if (!with_sparkles) {
15       return with_sparkles;
16    }
17
18    return add_rainbow(make_smaller(*with_sparkles));
19 }
```

Listing 28: Monadic interfaces. From Simon Brand implementation

Expected: Monads

```
1
2     tl::expected<image,fail_reason> get_cute_cat (const image& img)
       {
3      return crop_to_cat(img)
4             .and_then(add_bow_tie)
5             .and_then(make_eyes_sparkle)
6             .map(make_smaller)
7             .map(add_rainbow);
8  }
```

Listing 29: Monadic interfaces

Expected: Monads

- map: carries out some operation on the stored object if there is one.
- $tl :: expected < std :: size\_t, std :: error_code > s = exp_string.map(std :: string :: size)$;
- map_error: carries out some operation on the unexpected object if there is one.
- my_error_code translate_error (std::error_code);
- $tl :: expected < int, my\_error\_code > s = exp\_int.map\_error(translate\_error)$;
- and_then: like map, but for operations which return a tl::expected.
- $tl :: expected < ast, fail_reason > parse(const\ std :: strings)$;
- $tl :: expected < ast, fail\_reason > exp\_ast = exp\_string.and\_then(parse)$;
- or_else: calls some function if there is no value stored.
- $exp.or\_else([]throw\ std :: runtime_error"ohno";)$;

Content

Throwing by value

- Herb Sutter came with another idea...
- Zero-overhead deterministic exceptions: Throwing values: P0709
- "A big appeal to Go using error codes is as a rebellion against the overly complex languages in today's landscape. We have lost a lot of what makes C so elegant – that you can usually look at any line of code and guess what machine code it translates into.... You're in a statically typed programming language, and the dynamic nature of exceptions is precisely the reason they suck." — [Duffy 2016]
- Problem: "Filesystem library functions often provide two overloads, one that throws an exception to report file system errors, and another that sets an error_code [N3239]".
- Different error handling methods are proliferating.
- "... if STL switched to making bugs logic errors, domain errors be contracts... and we could make bad_alloc fail fast, we could use the standard STL unmodified throughout Windows." — Pavel Curtis, private communication between Herb Sutter

Throwing by value

- "This proposal aims to marry the best of exceptions and error codes: to allow a function to declare that it throws values of a statically known type" P0709
- Throwing such values behaves as-if the function returned unionR;E;+bool
- On success the function returns the normal return value R and on error the function returns the error value type E, both in the same return channel including using the same registers. The discriminant can use an unused CPU flag or a register.
- Basically it proposes to bake into the language std::expected...

Figure: C++ language baker

Throwing by value

- Throwing std::error_codes in the stack through the new keyword *throws*

```
1
2  string f() throws {
3  if (flip_a_coin())
4      throw arithmetic_error::something;
5  return std::string(xyzzy) + std::string(plover);
6  // any dynamic exception is translated to error
7  }
8  string g() throws { return f() +     plugh    ; }
9  // any dynamic exception is translated to error
10 int main() {
11     try {
12         auto result = g();
13         cout <<   success   , result is:      << result;
14     }
15     catch(error err) {
16     // catch by value is fine
17         cout <<   failed   , error is:      << err.error();
18     }
19 }
```

Listing 30: Ex. from P0323

Throwing by value

```
1
2  expected<int, errc> safe_divide(int i, int j) {
3  if (j == 0)
4      return unexpected(arithmetic_errc::divide_by_zero);
5  if (i == INT_MIN && j == -1)
6      return unexpected(arithmetic_errc::integer_divide_overflows);
7  if (i % j != 0)
8      return unexpected(arithmetic_errc::not_integer_division);
9  return i / j;
10 }
11
12 expected<double, errc> caller(double i, double j, double k) {
13     auto q = safe_divide(j, k);
14     if (q) return i + *q;
15     else return q;
16 }
```

Listing 31: Ex. from P0323R3

Throwing by value

```
1
2  int safe_divide(int i, int j) throws {
3  if (j == 0)
4      throw arithmetic_errc::divide_by_zero;
5  if (i == INT_MIN && j == -1)
6      throw arithmetic_errc::integer_divide_overflows;
7  if (i % j != 0)
8      throw arithmetic_errc::not_integer_division;
9  return i / j;
10 }
11
12 double caller(double i, double j, double k) throws {
13 return i + safe_divide(j, k);
14 }
```

Listing 32: Ex. from P0709R0

Throwing by value: Heap exhaustion debate

- In favour
  - It is an explicit fallible request in source code, unlike stack exhaustion
  - Some (possibly lots) of code is correct today for heap exhaustion even without coding for it specially. Calling new asking for too much data.
- Against
  - Testing is much more difficult. Specially for heap exhaustion.
  - Recovery requires special care. Many programs that believe that they can recover, they are buggy.
  - Treating heap exhaustion the same as all other errors causes a overhead in the cases that don't need it.
  - Many programs nowadays fail on heap allocation failure (e.g., google)

Figure: Pros and Cons

Throwing by value

- In any case this proposal does not have a target year (e.g., C++23, C++26...)

Figure: Definitely not in C++20..

Content

Conclusion

- There co-exists nowadays 3 methods for error handling : error codes, exceptions and outcome/expected algebraic data types.
- A possible 4th method may be baked into the language
- If a new project is started, a outcome/expected approach is the most recommendable.

Content

Questions?

Any question?