Futuristic Error Handling

Error handling in C++ today and tomorrow

Dawid Pilarski

dawid.pilarski@panicsofware.com

Why am I here?

Why should we bother with error handling?

Recommendable error handling mechanism

Which error mechanism would you choose?

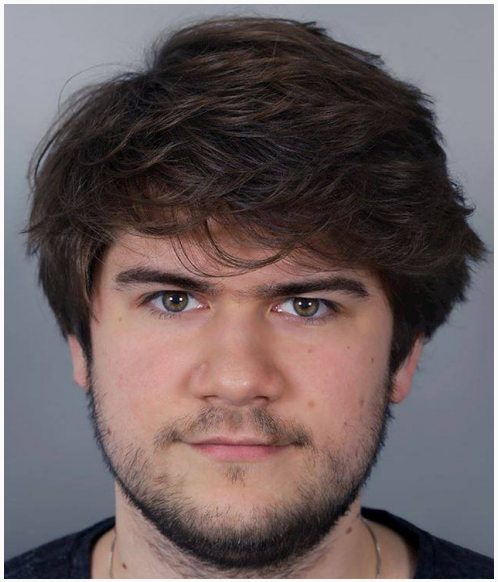
There exist two common strategies for error handling:

- error codes
- exceptions

Who am I?

Dawid Pilarski

- Senior Software Developer in TomTom
- Member of the ISO/JTC1/SC22/WG21
- Member of the PKN KT (programming languages)
- C++ blog writer



Error codes nowadays

What are the error codes?

According to the Wikipedia :

- error code is an enumerated message,

What are the error codes?

According to the Wikipedia :

- error code is an enumerated message,
- that corresponds to the status of a specific software application.

What are the error codes?

According to the Wikipedia :

- error code is an enumerated message,
- that corresponds to the status of a specific software application.
- They are typically used to identify faults, such as those in faulty hardware, software, or incorrect user input

The error codes.

- Old. C-compatible. Comes from assembly time.

The error codes.

- Old. C-compatible. Comes from assembly time.
- Machine friendly.

The error codes.

- Old. C-compatible. Comes from assembly time.
- Machine friendly.
- Super fast.

The error codes.

- Old. C-compatible. Comes from assembly time.
- Machine friendly.
- Super fast.
- Used till today.

Error code example

```
1  int sqlite3_open( const char *filename, sqlite3 **ppDb );
```

Error code example

```c
int sqlite3_open( const char *filename, sqlite3 **ppDb );
```

```c
int open_status = sqlite3_open(/* ... */ );
if(open_status == SQLITE_OK){
  // make use of opened database
} else if( open_status == SQLITE_CANTOPEN_ISDIR ) {
  // handle the error
}
```

Error code example

```c
int sqlite3_open( const char *filename, sqlite3 **ppDb );
```

```c
int open_status = sqlite3_open(/* ... */ );
if(open_status == SQLITE_OK){
  // make use of opened database
} else if( open_status == SQLITE_CANTOPEN_ISDIR ) {
  // handle the error
}
```

Error code example

```c
int sqlite3_open( const char *filename, sqlite3 **ppDb );
```

```c
int open_status = sqlite3_open(/* ... */ );
if(open_status == SQLITE_OK){
  // make use of opened database
} else if( open_status == SQLITE_CANTOPEN_ISDIR ) {
  // handle the error
}
```

Handle the error

How to handle the error correctly?

Handle the error

How to handle the error correctly?

- `std::terminate()`

Handle the error

How to handle the error correctly?

- `std::terminate()`
- take the error callback

Handle the error

How to handle the error correctly?

- `std::terminate()`
- take the error callback
- propagate the error to the caller

Error codes - propagation

Propagation

```
1  void foo_bar(int& errc /*...*/){
2    errc = foo();
3    // ...
4    errc = bar();
5    // ...
6  }
```

Propagation

```
1  void foo_bar(int& errc /*...*/){
2    errc = foo();
3    // ...
4    errc = bar();
5    // ...
6  }
```

Propagation

```
1  void foo_bar(foo_bar_errc errc&){
2    foo_errc ferrc = foo();
3    errc = translate_foo(ferrc);
4    // ...
5    bar_errc berrc = bar();
6    errc = translate_foo(berrc);
7  }
```

Propagation

```
1  void foo_bar(foo_bar_errc errc&){
2    foo_errc ferrc = foo();
3    errc = translate_foo(ferrc);
4    if(errc != foo_errc::SUCCESS){
5            return;
6    }
7    // ...
8  }
```

C-style error codes summary

So we can see serious disadvantages (except for obvious advantages):

- success path same as error path

C-style error codes summary

So we can see serious disadvantages (except for obvious advantages):

- success path same as error path
- cluttering code with translations

C-style error codes summary

So we can see serious disadvantages (except for obvious advantages):

- success path same as error path
- cluttering code with translations
- boring manual error propagation

Error codes - modern approach

Standard library support - what do we need?

- A way to define new error codes
- A way to distinguish domain of the error codes

Standard library support - what we get?

We get three new major types:

- std::error_code

Standard library support - what we get?

We get three new major types:

- std::error_code
- std::error_category

Standard library support - what we get?

We get three new major types:

- std::error_code
- std::error_category
- std::error_condition

std::error_code in action

```cpp
std::error_code errcode;
is_regular_file("non_existent_directory", errcode);

std::cout << errcode << std::endl;
std::cout << errcode.value() << std::endl;
std::cout << errcode.message() << std::endl;
std::cout << errcode.category().name() << std::endl;
```

output

```
$ generic:2
$ 2
$ No such file or directory
$ generic
```

std::error_code in action

```
1  std::error_code errcode;
2  is_regular_file("non_existent_directory", errcode);
3
4  std::cout << errcode << std::endl;
5  std::cout << errcode.value() << std::endl;
6  std::cout << errcode.message() << std::endl;
7  std::cout << errcode.category().name() << std::endl;
```

output

```
$ generic:2
$ 2
$ No such file or directory
$ generic
```

std::error_code in action

```
1  std::error_code errcode;
2  is_regular_file("non_existent_directory", errcode);
3
4  std::cout << errcode << std::endl;
5  std::cout << errcode.value() << std::endl;
6  std::cout << errcode.message() << std::endl;
7  std::cout << errcode.category().name() << std::endl;
```

output

```
$ generic:2
$ 2
$ No such file or directory
$ generic
```

std::error_code in action

```
1  std::error_code errcode;
2  is_regular_file("non_existent_directory", errcode);
3
4  std::cout << errcode << std::endl;
5  std::cout << errcode.value() << std::endl;
6  std::cout << errcode.message() << std::endl;
7  std::cout << errcode.category().name() << std::endl;
```

output

```
$ generic:2
$ 2
$ No such file or directory
$ generic
```

std::error_code in action

```
1  std::error_code errcode;
2  is_regular_file("non_existent_directory", errcode);
3
4  std::cout << errcode << std::endl;
5  std::cout << errcode.value() << std::endl;
6  std::cout << errcode.message() << std::endl;
7  std::cout << errcode.category().name() << std::endl;
```

output

```
$ generic:2
$ 2
$ No such file or directory
$ generic
```

Acting upon error

```
1  std::error_code errcode;
2  is_regular_file("non_existent_file", errcode);
3
4  if(errcode == errc::no_such_file_or_directory){
5    // creating a file
6  }
```

Acting upon error

```
1  std::error_code errcode;
2  is_regular_file("non_existent_file", errcode);
3
4  if(errcode == errc::no_such_file_or_directory){
5    // creating a file
6  }
```

Acting upon error

With enum error codes.

```
1  void foo_bar(foo_bar_errc errc&){
2    foo_errc ferrc = foo();
3    errc = translate_foo(ferrc);
4    if(errc != foo_errc::SUCCESS){
5      return;
6    }
7    // ...
8  }
```

Acting upon error

With std::error_code.

```
1  void foo_bar(std::error_code errc&){
2    errc = foo();
3    if(errc){
4      return;
5    }
6    // ...
7  }
```

Let's define our own error code

Steps to create own error code:

- define custom enum with error codes

Let's define our own error code

Steps to create own error code:

- define custom enum with error codes
- inform, that the enum is an error code

Let's define our own error code

Steps to create own error code:

- define custom enum with error codes
- inform, that the enum is an error code
- create custom error category

Let's define our own error code

Steps to create own error code:

- define custom enum with error codes
- inform, that the enum is an error code
- create custom error category
- create enum to error code factory function

Let's define our own error code

Steps to create own error code:

- define custom enum with error codes
- inform, that the enum is an error code
- create custom error category
- create enum to error code factory function
- define custom error condition

Let's define our own error code

Steps to create own error code:

- define custom enum with error codes
- inform, that the enum is an error code
- create custom error category
- create enum to error code factory function
- define custom error condition
  - define error condition enum

Let's define our own error code

Steps to create own error code:

- define custom enum with error codes
- inform, that the enum is an error code
- create custom error category
- create enum to error code factory function
- define custom error condition
  - define error condition enum
  - inform the world about new error condition enum

Let's define our own error code

Steps to create own error code:

- define custom enum with error codes
- inform, that the enum is an error code
- create custom error category
- create enum to error code factory function
- define custom error condition
  - define error condition enum
  - inform the world about new error condition enum
  - create custom error category for condition enum

Let's define our own error code

Steps to create own error code:

- define custom enum with error codes
- inform, that the enum is an error code
- create custom error category
- create enum to error code factory function
- define custom error condition
  - define error condition enum
  - inform the world about new error condition enum
  - create custom error category for condition enum
  - make conversion function from new error condition enum to error condition

Let's define our own error code

Steps to create own error code:

- define custom enum with error codes
- inform, that the enum is an error code
- create custom error category
- create enum to error code factory function
- define custom error condition
  - define error condition enum
  - inform the world about new error condition enum
  - create custom error category for condition enum
  - make conversion function from new error condition enum to error condition
- enjoy!

Error codes -
defining custom error codes

Step 1 - define custom enum with error codes

```
1  enum class open_file_error {
2    SUCCESS, // zero means success
3    NO_SUCH_FILE_OR_DIRECTORY,
4    FILE_IS_DIRECTORY,
5    LACK_OF_RESOURCES,
6    FILE_BROKEN,
7    NO_PERMISSIONS
8  };
```

Step 1 - define custom enum with error codes

```
1  enum class open_file_error {
2    SUCCESS, // zero means success
3    NO_SUCH_FILE_OR_DIRECTORY,
4    FILE_IS_DIRECTORY,
5    LACK_OF_RESOURCES,
6    FILE_BROKEN,
7    NO_PERMISSIONS
8  };
```

Step 1 - define custom enum with error codes

```
1  enum class open_file_error {
2    SUCCESS, // zero means success
3    NO_SUCH_FILE_OR_DIRECTORY,
4    FILE_IS_DIRECTORY,
5    LACK_OF_RESOURCES,
6    FILE_BROKEN,
7    NO_PERMISSIONS
8  };
```

Step 1 - define custom enum with error codes

```
enum class open_file_error {
  SUCCESS, // zero means success
  NO_SUCH_FILE_OR_DIRECTORY,
  FILE_IS_DIRECTORY,
  LACK_OF_RESOURCES,
  FILE_BROKEN,
  NO_PERMISSIONS
};
```

Step 2 - inform the world about new error code type

```cpp
namespace std{
  template <> struct
  is_error_code_enum<open_file_error> : std::true_type{};
}
```

Step 3 - custom error category

```
1  struct open_file_error_domain : std::error_category {
2    const char *name() const noexcept override;
3    std::string message(int errc) const override;
4  };
```

Step 3 - custom error category

```
1  const char* open_file_error_domain::name() const noexcept{
2    return "Open File Error";
3  }
```

Step 3 - custom error category

```
1  std::string open_file_error_domain::message(int errc) const{
2    if(errc < 0 or errc > 4) return "UNKNOWN ERROR";
3    switch (static_cast<open_file_error>(errc)){
4      case open_file_error::SUCCESS:
5        return "Success.";
6      case open_file_error::NO_SUCH_FILE_OR_DIRECTORY:
7        return "File does not exist.";
8      // other cases
9      case open_file_error::NO_PERMISSIONS:
10       return "Missing permissions to open the file."
11   }
12 }
```

Step 3 - custom error category

```
1  std::string open_file_error_domain::message(int errc) const{
2    if(errc < 0 or errc > 4) return "UNKNOWN ERROR";
3    switch (static_cast<open_file_error>(errc)){
4      case open_file_error::SUCCESS:
5        return "Success.";
6      case open_file_error::NO_SUCH_FILE_OR_DIRECTORY:
7        return "File does not exist.";
8      // other cases
9      case open_file_error::NO_PERMISSIONS:
10       return "Missing permissions to open the file."
11   }
12 }
```

Step 3 - custom error category

```
1  std::string open_file_error_domain::message(int errc) const{
2    if(errc < 0 or errc > 4) return "UNKNOWN ERROR";
3    switch (static_cast<open_file_error>(errc)){
4      case open_file_error::SUCCESS:
5        return "Success.";
6      case open_file_error::NO_SUCH_FILE_OR_DIRECTORY:
7        return "File does not exist.";
8      // other cases
9      case open_file_error::NO_PERMISSIONS:
10       return "Missing permissions to open the file."
11   }
12 }
```

Step 3 - custom error category

```
1  std::string open_file_error_domain::message(int errc) const{
2    if(errc < 0 or errc > 4) return "UNKNOWN ERROR";
3    switch (static_cast<open_file_error>(errc)){
4      case open_file_error::SUCCESS:
5        return "Success.";
6      case open_file_error::NO_SUCH_FILE_OR_DIRECTORY:
7        return "File does not exist.";
8      // other cases
9      case open_file_error::NO_PERMISSIONS:
10       return "Missing permissions to open the file."
11   }
12 }
```

Step 3 - custom error category

```
1  std::string open_file_error_domain::message(int errc) const{
2    if(errc < 0 or errc > 4) return "UNKNOWN ERROR";
3    switch (static_cast<open_file_error>(errc)){
4      case open_file_error::SUCCESS:
5        return "Success.";
6      case open_file_error::NO_SUCH_FILE_OR_DIRECTORY:
7        return "File does not exist.";
8      // other cases
9      case open_file_error::NO_PERMISSIONS:
10       return "Missing permissions to open the file."
11   }
12 }
```

Step 4 - factory function

```
1  namespace std{
2    template <typename ErrorCode>
3    error_code::error_code(typename std::enable_if<
4                                       is_error_code_enum<
5                                           ErrorCode>
6                                       ::value, ErrorCode>
7                                ::type errcode) noexcept
8                  : error_code(make_error_code(errcode))
9    {}
10 }
```

Step 4 - factory function

```
1  namespace std{
2    template <typename ErrorCode>
3    error_code::error_code(typename std::enable_if<
4                                        is_error_code_enum<
5                                            ErrorCode>
6                                        ::value, ErrorCode>
7                               ::type errcode) noexcept
8                : error_code(make_error_code(errcode))
9    {}
10 }
```

Step 4 - factory function

```
1  namespace std{
2    template <typename ErrorCode>
3    error_code::error_code(typename std::enable_if<
4                                       is_error_code_enum<
5                                           ErrorCode>
6                                       ::value, ErrorCode>
7                              ::type errcode) noexcept
8                : error_code(make_error_code(errcode))
9    {}
10 }
```

Step 4 - factory function

```
1  std::error_code make_error_code(open_file_error errc){
2    return {static_cast<int>(errc), open_file_error_domain};
3  }
```

Step 5 - custom error condition

```
1  enum class library_error_condition{
2    SUCCESS,
3    WRONG_ARGUMENT,
4    OS_ERROR,
5    PERMISSIONS_ERROR
6  };
```

Step 5 - custom error condition

```
enum class library_error_condition{
  SUCCESS,
  WRONG_ARGUMENT,
  OS_ERROR,
  PERMISSIONS_ERROR
};
```

```
enum class open_file_error {
  SUCCESS, // zero means success
  NO_SUCH_FILE_OR_DIRECTORY,
  FILE_IS_DIRECTORY,
  LACK_OF_RESOURCES,
  FILE_BROKEN,
  NO_PERMISSIONS
};
```

Step 5 - custom error condition

```
namespace std{
  template <> struct
  is_error_condition_enum<library_error_condition>
                              : std::true_type{};
}
```

Step 5 - custom error condition

```
1  struct library_error_domain : std::error_category{
2    const char *name() const noexcept override;
3    std::string message(int errc) const override;
4    bool equivalent(const std::error_code &errc, int condition)
5                                          const noexcept override;
6  };
```

Step 5 - custom error condition

```
1  bool library_error_domain::equivalent(
2            const std::error_code &errc, int condition)
3                                              const noexcept{
4    switch (static_cast<library_error_condition>(condition)){
5      case library_error::SUCCESS:
6        if(errc == open_file_error::SUCCESS)
7          return true;
8      case library_error::WRONG_ARGUMENT:
9        if(errc ==
10           open_file_error::NO_SUCH_FILE_OR_DIRECTORY or
11         errc ==
12           open_file_error::FILE_IS_DIRECTORY)
13         return true;
14     //other cases
15   }
16   return false;
17 }
```

Step 5 - custom error condition

```
1  bool library_error_domain::equivalent(
2            const std::error_code &errc, int condition)
3                                              const noexcept{
4    switch (static_cast<library_error_condition>(condition)){
5      case library_error::SUCCESS:
6        if(errc == open_file_error::SUCCESS)
7          return true;
8      case library_error::WRONG_ARGUMENT:
9        if(errc ==
10           open_file_error::NO_SUCH_FILE_OR_DIRECTORY or
11         errc ==
12           open_file_error::FILE_IS_DIRECTORY)
13         return true;
14     //other cases
15   }
16   return false;
17 }
```

Step 5 - custom error condition

```
1  bool library_error_domain::equivalent(
2            const std::error_code &errc, int condition)
3                                              const noexcept{
4    switch (static_cast<library_error_condition>(condition)){
5      case library_error::SUCCESS:
6        if(errc == open_file_error::SUCCESS)
7          return true;
8      case library_error::WRONG_ARGUMENT:
9        if(errc ==
10           open_file_error::NO_SUCH_FILE_OR_DIRECTORY or
11         errc ==
12           open_file_error::FILE_IS_DIRECTORY)
13         return true;
14     //other cases
15   }
16   return false;
17 }
```

Step 6 - Enjoy - real life example

```
1  std::error_code errcode;
2  auto settings = read_user_settings("settings.txt", errcode);
3
4  if(!errcode)
5    return settings;
6
7  std::cout << errcode.category().name() << " : " <<
8               errcode.message() << std::endl;
9
10 if(errcode == library_error_condition::PERMISSIONS_ERROR)
11   ask_for_permissions();
12 else if (errcode == library_error_condition::OS_ERROR)
13   std::terminate();
14 else if (errcode == library_error_condition::WRONG_ARGUMENTS)
15   std::terminate();
```

Step 6 - Enjoy - real life example

```
1  std::error_code errcode;
2  auto settings = read_user_settings("settings.txt", errcode);
3
4  if(!errcode)
5    return settings;
6
7  std::cout << errcode.category().name() << " : " <<
8               errcode.message() << std::endl;
9
10 if(errcode == library_error_condition::PERMISSIONS_ERROR)
11   ask_for_permissions();
12 else if (errcode == library_error_condition::OS_ERROR)
13   std::terminate();
14 else if (errcode == library_error_condition::WRONG_ARGUMENTS)
15   std::terminate();
```

Step 6 - Enjoy - real life example

```
1  std::error_code errcode;
2  auto settings = read_user_settings("settings.txt", errcode);
3
4  if(!errcode)
5    return settings;
6
7  std::cout << errcode.category().name() << " : " <<
8               errcode.message() << std::endl;
9
10 if(errcode == library_error_condition::PERMISSIONS_ERROR)
11   ask_for_permissions();
12 else if (errcode == library_error_condition::OS_ERROR)
13   std::terminate();
14 else if (errcode == library_error_condition::WRONG_ARGUMENTS)
15   std::terminate();
```

Step 6 - Enjoy - real life example

```
1  std::error_code errcode;
2  auto settings = read_user_settings("settings.txt", errcode);
3
4  if(!errcode)
5    return settings;
6
7  std::cout << errcode.category().name() << " : " <<
8               errcode.message() << std::endl;
9
10 if(errcode == library_error_condition::PERMISSIONS_ERROR)
11   ask_for_permissions();
12 else if (errcode == library_error_condition::OS_ERROR)
13   std::terminate();
14 else if (errcode == library_error_condition::WRONG_ARGUMENTS)
15   std::terminate();
```

Step 6 - Enjoy - real life example

```
settings read_user_settings(std::string_view filename,
                                        error_code& errc){
  auto file_handle = open(filename, errc);
  if (errc) return {};

  ensure_file_correct(file_handle, errc);
  if(errc) return {};

  return read_settings(file_handle);
}
```

Step 6 - Enjoy - real life example

```
1  settings read_user_settings(std::string_view filename,
2                                          error_code& errc){
3    auto file_handle = open(filename, errc);
4    if (errc) return {};
5
6    ensure_file_correct(file_handle, errc);
7    if(errc) return {};
8
9    return read_settings(file_handle);
10 }
```

Step 6 - Enjoy - real life example

```
settings read_user_settings(std::string_view filename,
                                          error_code& errc){
  auto file_handle = open(filename, errc);
  if (errc) return {};

  ensure_file_correct(file_handle, errc);
  if(errc) return {};

  return read_settings(file_handle);
}
```

Step 6 - Enjoy - real life example

```
settings read_user_settings(std::string_view filename,
                                          error_code& errc){
  auto file_handle = open(filename, errc);
  if (errc) return {};

  ensure_file_correct(file_handle, errc);
  if(errc) return {};

  return read_settings(file_handle);
}
```

Error codes - summary

Error codes summary

Pros

- Performance
  - speed

Error codes summary

Pros

- Performance
  - speed
  - small (occupied memory)

Error codes summary

Pros

- Performance
  - speed
  - small (occupied memory)
  - speed predictability

Error codes summary

Pros

- Performance
  - speed
  - small (occupied memory)
  - speed predictability
  - memory occupation predictability

Error codes summary

Pros

- Performance
  - speed
  - small (occupied memory)
  - speed predictability
  - memory occupation predictability
  - C compatibility

Error codes summary

Pros

- Performance
  - speed
  - small (occupied memory)
  - speed predictability
  - memory occupation predictability
  - C compatibility

Error codes summary

Pros

- Performance
  - speed
  - small (occupied memory)
  - speed predictability
  - memory occupation predictability
  - C compatibility

Cons

- business logic cluttering

Error codes summary

Pros

- Performance
  - speed
  - small (occupied memory)
  - speed predictability
  - memory occupation predictability
  - C compatibility

Cons

- business logic cluttering
- massive amount of boilerplate code

Error codes summary

Pros

- Performance
  - speed
  - small (occupied memory)
  - speed predictability
  - memory occupation predictability
  - C compatibility

Cons

- business logic cluttering
- massive amount of boilerplate code
- template magic in case of std::error_code

Exceptions to the rescue (?)

Brief look at the example

```
try{
  auto settings = read_user_settings("settings.txt");
} catch(permissions_error& err){
  // logic
} /* catch(path_not_found& err){
  // logic
}  catch(std::invalid_argument& ){
  // logic
} */
```

Brief look at the example

```
1  settings read_user_settings(std::string_view filename){
2    auto file_handle = open(filename);
3    ensure_file_correct(file_handle);
4
5    return read_settings(file_handle);
6  }
```

Defining custom exception

```
1  class open_file_error : public std::runtime_error{};
```

Dark side of the exceptions

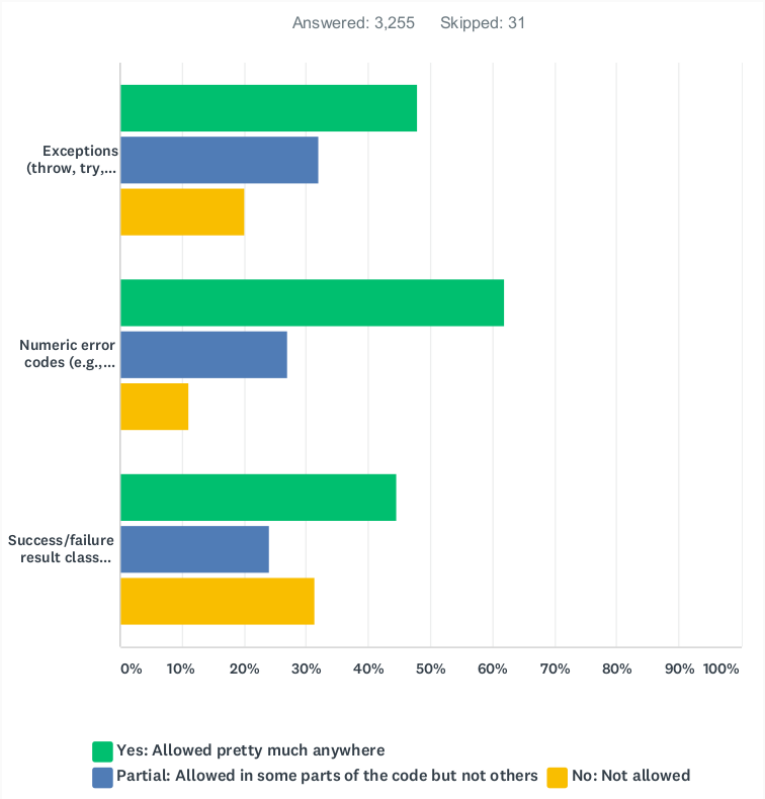
- Still translation of exceptions is needed

Dark side of the exceptions

- Still translation of exceptions is needed
- For performance related reasons about 50% of projects have disabled exceptions CppDevSurvey 2018

Dark side of the exceptions

- Still translation of exceptions is needed
- For performance related reasons about 50% of projects have disabled exceptions CppDevSurvey 2018



C++ - zero overhead rule

What is zero overhead?

- language features can introduce overhead

What is zero overhead?

- language features can introduce overhead
- "you don't pay for what you don't use"

What is zero overhead?

- language features can introduce overhead
- "you don't pay for what you don't use"
- if you use a feature it should be as afficient as handcoded version.

Exceptions not to the rescue

Exceptions break the zero overhead rule.

But why?

Exceptions - how do they work?

Approaches towards implementation

Two major kinds of implementation:

- additional data added to the frame stack

Approaches towards implementation

Two major kinds of implementation:

- additional data added to the frame stack
- additional data added to someplace on the heap

Implementations' consequences

implementation	performance	
	without throwing	with throwing
frame-based	overhead	fast
table-based	almost no overhead	slow

Implementations' consequences

implementation	performance	
	without throwing	with throwing
frame-based	overhead	fast
table-based	almost no overhead	slow

Binary size

No matter what's the strategy for exception handling. The binary will grow even if you do not use exceptions.

Exceptions summary

Pros

- differentiated error and success paths

Exceptions summary

Pros

- differentiated error and success paths
- automagical error propagation

Exceptions summary

Pros

- differentiated error and success paths
- automagical error propagation
- little/no boilerplate code

Exceptions summary

Pros

- differentiated error and success paths
- automagical error propagation
- little/no boilerplate code

Exceptions summary

Pros

- differentiated error and success paths
- automagical error propagation
- little/no boilerplate code

Cons

- Performance
  - slow

Exceptions summary

Pros

- differentiated error and success paths
- automagical error propagation
- little/no boilerplate code

Cons

- Performance
  - slow
  - not deterministic speed

Exceptions summary

Pros

- differentiated error and success paths
- automagical error propagation
- little/no boilerplate code

Cons

- Performance
  - slow
  - not deterministic speed
  - not deterministic storage occupation

Exceptions summary

Pros

- differentiated error and success paths
- automagical error propagation
- little/no boilerplate code

Cons

- Performance
  - slow
  - not deterministic speed
  - not deterministic storage occupation
  - not compatible with C

Exceptions summary

Pros

- differentiated error and success paths
- automagical error propagation
- little/no boilerplate code

Cons

- Performance
  - slow
  - not deterministic speed
  - not deterministic storage occupation
  - not compatible with C
  - not usable in any safety standards (e.g. MISRA)

Possible future of error handling.

Perfect error handling mechanism

feature	exceptions	std::error_code
distinct error path	yes	no
distinct success path	yes	no
unhandled error propagation	yes	no
unhandled error is visible	no	yes
uncluttered business logic	yes	no
RTTI required	yes	no
deterministic space/time occupation	no	yes
time cost == return	no	yes
C compatibility	no	no

Key idea for improvement

Key ideas

- Let's use the return channel to return the std::error

Let's call those *static exceptions*

Key idea for improvement

Key ideas

- Let's use the return channel to return the std::error
- Let the compiler generate boilerplate code for error propagation

Let's call those *static exceptions*

How to use return channel for std::error

We can do that manually using variant:

```
using Result =
      variant<T, std::error>;
Result foo();
```

But this can be nicer with syntax sugar:

```
T foo() throws;
```

Static exceptions example

```
1  string f() throws {
2    if (flip_a_coin()) throw arithmetic_error::something;
3    return "xyzzy"s + "plover";
4  }
5
6  string g() throws { return f() + "plugh"; }
7
8  int main() {
9    try {
10     auto result = g();
11     cout << "success, result is: " << result;
12   } catch(error err) {
13     cout << "failed, error is: " << err.error();
14   }
15 }
```

Static exceptions example

```
1  string f() throws {
2    if (flip_a_coin()) throw arithmetic_error::something;
3    return "xyzzy"s + "plover";
4  }
5
6  string g() throws { return f() + "plugh"; }
7
8  int main() {
9    try {
10     auto result = g();
11     cout << "success, result is: " << result;
12   } catch(error err) {
13     cout << "failed, error is: " << err.error();
14   }
15 }
```

Static exceptions example

```
string f() throws {
  if (flip_a_coin()) throw arithmetic_error::something;
  return "xyzzy"s + "plover";
}

string g() throws { return f() + "plugh"; }

int main() {
  try {
    auto result = g();
    cout << "success, result is: " << result;
  } catch(error err) {
    cout << "failed, error is: " << err.error();
  }
}
```

Static exceptions example

```
1  string f() throws {
2    if (flip_a_coin()) throw arithmetic_error::something;
3    return "xyzzy"s + "plover";
4  }
5
6  string g() throws { return f() + "plugh"; }
7
8  int main() {
9    try {
10     auto result = g();
11     cout << "success, result is: " << result;
12   } catch(error err) {
13     cout << "failed, error is: " << err.error();
14   }
15 }
```

Compiler's support in handling errors

```
1  int f1() throws;
2  int f2() throws;
3
4  int main(){
5  // return f1() + f2(); // error
6  try return f1() + f2();// ok, covers both
7  return try f1()+ f2();
8  }
```

Compiler's support in handling errors

```
1  int f1() throws;
2  int f2() throws;
3
4  int main(){
5  // return f1() + f2(); // error
6  try return f1() + f2();// ok, covers both
7  return try f1()+ f2();
8  }
```

Compiler's support in handling errors

```
1  int f1() throws;
2  int f2() throws;
3
4  int main(){
5  // return f1() + f2(); // error
6  try return f1() + f2();// ok, covers both
7  return try f1()+ f2();
8  }
```

Compiler's support in handling errors

```
1  int f1() throws;
2  int f2() throws;
3
4  int main(){
5  // return f1() + f2(); // error
6  try return f1() + f2();// ok, covers both
7  return try f1()+ f2();
8  }
```

Cool! But what about C compatibility

C compatibility

It is possible, that C will be compatible with static exceptions!

This implies:

- Exceptions could be thrown from C++, passed through C and catched again in C++
- We could handle C++ exceptions in C
- We could handle C exceptions in C++

Short story about C language

```c
_Either(int, std_error) somefunc(int a){
  return a > 5 ? _Expected(a) : _Unexpected(a);
}

// ...

_Either(int, std_error) ret = somefunc(a);
if(ret)
  printf("%d\n", ret.expected);
else
  printf("%f\n", ret.unexpected);
```

Static exceptions summary

feature	static exceptions
distinct error path	yes
distinct success path	yes
unhandled error propagation	yes
unhandled error is visible	yes
uncluttered business logic	yes
RTTI required	no
deterministic space/time occupation	yes
time cost == return	yes
C compatibility	maybe

Possible issue

We will end up having 3 ways to handle error codes:

- dynamic exceptions
- static exceptions
- old style error codes

Bibliography

This presentation wouldn't be possible without:

- Herb Sutter - author of the proposal (code examples, exception features taken from his proposal) - p0709r1
- Andrzej Krzemiński for his blog about error codes and error conditions - Your own error code

Thank you

Thank you for your attention!

Questions?

---

blog: blog.panicsoftware.com

presentation: github.com/dawidpilarski/error_handling_presentation