Futuristic Error Handling

Error handling in C++ today and tomorrow

Dawid Pilarski

dawid.pilarski@panicsofware.com

Introduction

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?

Error codes nowadays

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

```
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
}
```

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

```
void foo_bar(int& errc /*...*/){
  errc = foo();
  // ...
  errc = bar();
  // ...
}
```

error translation

```
void foo_bar(foo_bar_errc errc&){
  foo_errc ferrc = foo();
  errc = translate_foo(ferrc);
  // ...
  bar_errc berrc = bar();
  errc = translate_foo(berrc);
}
```

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
- boiler plate code

C-style error codes summary

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

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

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
- And to fix as many C-style issues as possible

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

```
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
```

Acting upon error

```
std::error_code errcode;
is_regular_file("non_existent_file", errcode);

if(errcode == errc::no_such_file_or_directory){
  // creating a file
}
```

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 (or use existing one)

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 (or use existing one)
- 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 (or use existing one)
- create enum to error code factory function
- (optional) 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 (or use existing one)
- create enum to error code factory function
- (optional) 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 (or use existing one)
- create enum to error code factory function
- (optional) 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 (or use existing one)
- create enum to error code factory function
- (optional) define custom error condition
    - define error condition enum
    - inform the world about new error condition enum
    - make conversion function from new error code 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 (or use existing one)
- create enum to error code factory function
- (optional) define custom error condition
  - define error condition enum
  - inform the world about new error condition enum
  - make conversion function from new error code to error condition
- enjoy!

Error codes -
defining custom error codes

Step 1 - define custom enum with error codes

```cpp
enum class map_access_error : int {
  SUCCESS, // zero means success
  MAP_NOT_INSTALLED,
  LACK_OF_PERMISSION,
  MAP_CORRUPTED,
};
```

Step 2 - inform the world about new error code type

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

Step 3 - custom error category

```
struct map_access_domain : std::error_category {
  const char *name() const noexcept override;
  std::string message(int errc) const override;
};
```

Step 3 - custom error category

```
const char* map_access_domain::name() const noexcept{
  return "Map Access Error";
}
```

Step 3 - custom error category

```
std::string map_access_domain::message(int errc) const{
  switch (static_cast<map_access_error>(errc)){
    case map_access_error::SUCCESS:
      return "SUCCESS";
    case map_access_error::MAP_NOT_INSTALLED:
      return "MAP IS NOT INSTALLED ON THE DEVICE";
    case map_access_error::LACK_OF_PERMISSION:
      return "MISSING PERMISSIONS TO READ THE MAP";
    case map_access_error::MAP_CORRUPTED:
      return "MAP IS CORRUPTED. REINSTALLATION NEEDED";
    default:
      return "ERROR UNKNOWN";
  }
}
```

Step 4 - factory function

```
namespace std{
  template <typename ErrorCode>
  error_code(typename std::enable_if<
                      is_error_code_enum<
                        ErrorCode>
                      ::value, ErrorCode>
                    ::type errcode) noexcept
             : error_code(make_error_code(errcode))
  {}
}
```

Step 4 - factory function

```
std::error_code make_error_code(map_access_error errc){
  return {static_cast<int>(errc), map_access_error_domain};
}
```

Step 5 - custom error condition

```
enum class calculate_route_error : int {
  SUCCESS,
  MAP_ERROR,
  COULD_NOT_FIND_PATH,
  WRONG_ARGUMENTS
};
```

Step 5 - custom error condition

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

Step 5 - custom error condition

```
struct calculate_route_error_domain : std::error_category{
  const char *name() const noexcept override;
  std::string message(int errc) const override;
  bool equivalent(const std::error_code &errc, int condition)
                                              const noexcept override;
};
```

Step 5 - custom error condition

```cpp
bool calculate_route_error_domain::equivalent(
          const std::error_code &errc, int condition)
                                             const noexcept{

  switch (static_cast<calculate_route_error>(condition)){
    case calculate_route_error::SUCCESS:
      if(errc.value() == 0)
        return true;
    case calculate_route_error::MAP_ERROR:
      if(errc.category().name() == map_access_domain().name())
        return true;

    // other cases
  }
  return false;
}
```

Step 6 - Enjoy - real life example

```
std::error_code errcode;
auto route = calculate_route({}, {}, {}, errcode);

if(!errcode)
  return route;

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

if(errcode == calculate_route_error::MAP_ERROR)
  reinstall_map();
else if (errcode == calculate_route_error::COULD_NOT_FIND_PATH)
  inform_user_no_path_found();
else if (errcode == calculate_route_error::WRONG_ARGUMENTS)
  std::terminate();
```

Step 6 - Enjoy - real life example

```
route calculate_route(point a, point b, route_options options,
                      std::error_code& errc){
  auto map_database = database(errc);
  if (errc) return {};

  auto a_handle = map_database.find(a, errc);
  if(errc) return {};
  auto b_handle = map_database.find(b, errc);
  if(errc) return {};

  route result_route = find_path(a_handle, b_handle,
                                 options, errc);
  if(errc) return {};

  return result_route;
}
```

Error codes - summary

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

```cpp
try{
  auto route = calculate_route(/*arguments*/);
} catch(map_error& err){
  // logic
} catch(path_not_found& err){
  // logic
} /* catch(std::invalid_argument){

} */
```

Brief look at the example

```
route calculate_route(point a, point b,
                      route_options options){

  auto map_database = database();

  auto a_handle = map_database.find(a, errc);
  auto b_handle = map_database.find(b, errc);

  route result_route = find_path(a_handle, b_handle, options);

  return result_route;
}
```

Defining custom exception

```
class map_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 reasons 50% of projects have disabled exceptions

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

C++ in Mars rover

zawartość...