Error handling in C++

p0709r1 , p0323r3

Dawid Pilarski

1. Current state of error handling
   - Error codes
   - Exceptions
2. `std::expected`
3. New exception model
   - How can new exception handling look like

Plan

1. Current state of error handling
   - Error codes
   - Exceptions
2. `std::expected`
3. New exception model
   - How can new exception handling look like

Error handling and performance

There exist two common strategies for error handling:

- error codes
- exceptions

Error codes - example fopen

```c
/* fopen example */
#include <stdio.h>
int main ()
{
  FILE * pFile;
  pFile = fopen ("myfile.txt","w");
  if (pFile!=NULL)
  {
    //do stuff
  } else {
    //how do I know if everything is fine?

    switch(errno){
    //
    }
  }
  return 0;
}
```

Error code - better approach

```
// declaration
int sqlite3_open( const char *filename,
                  sqlite3 **ppDb /* OUT: SQLite db handle */);

//usage
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
}
```

further improvements:

- enums
- error code taken as reference

C++ 11 approach to the error codes

There are 3 types, that C++ 11 added to support error codes.

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

But don't forget about exceptions

And so there are also exceptions.

How could things look like with exceptions:

```
#include <stdio.h>
int main ()
{
  try {
    FILE* pFile = fopen ("myfile.txt","w");
    //stuff here
  }
  catch(std::exception& e){
    //handle error
  }
  return 0;
}
```

"types" of exceptions

We can divide implementation of exceptions into 2 types:

- table-based implementation
- frame based implementation

"You don't pay for what you don't use"

table based exceptions

optimized for scenarios when usually exceptions are not throwed

frame based exception

optimized for scenarios when exceptions are thrown often

binary size

no matter which implementation is chosen the binary size grows significantly even when exceptions are not used.

Let's stick to error codes or provide dual API

People from standardization commitee tried to do that and failed :)

Example of such failure can be functions from filesystem library

```
directory_iterator& operator++();
directory_iterator& increment( std::error_code& ec );
```

Let's stick to error codes or provide dual API

People from standardization commitee tried to do that and failed :)

Example of such failure can be functions from filesystem library

```
directory_iterator& operator++();
directory_iterator& increment( std::error_code& ec );
```

The increment function even though is meant to return errors through `std::error_code` can return some of the errors through exceptions.

Error codes continued

```
A::A(){          // a constructor here
  /* some initialization happening */
  /* but whoops an error occurs, what now?*/
}
```

Error codes continued

```
A::A(){  //a constructor here
  /* some initialization*/
  /* but whoops an error occurs */

  throw error;
}
```

Error codes continued

```
A::A(){  //a constructor here
  /* some initialization*/
  /* but whoops an error occurs */

  throw error;
}
```

```
A::A(){  //a constructor here
  /* some initialization */
  /* but whoops an error occurs */
}

bool A::IsValid(){
  // was init successful?
}
```

Current exception handling summary

feature	exceptions	error codes
constructors usability	✓	✗
concise code	✓	✗
performance	✗	✓
binary size	✗	✓
safety	✗	✓

Figure: comparison of error handling mechanisms' capabilities

Plan

1. Current state of error handling
   - Error codes
   - Exceptions
2. `std::expected`
3. New exception model
   - How can new exception handling look like

further with error codes

Idea

Let's embed the return error code with expected type

```
template <typename T, typename E>
class expected;
```

Usage example

```
enum class arithmetic_errc {
  divide_by_zero,
  not_integer_division,
  integer_divide_overflows
};
```

Usage example

```
enum class arithmetic_errc {
  divide_by_zero,
  not_integer_division,
  integer_divide_overflows
};
```

```
using errc = arithmetic_errc;
expected<double, errc>
safe_divide(double i, double j){
  if(j==0){
    return
    unexpected(errc::divide_by_zero);
  } else
    return i/j;
}
```

Plan

1. Current state of error handling
   - Error codes
   - Exceptions
2. std::expected
3. New exception model
   - How can new exception handling look like

New exception model - idea

Let's recall the comparison of exceptions and error codes:

feature	exceptions	error codes
constructors usability	✓	✗
concise code	✓	✗
performance	✗	✓
binary size	✗	✓
safety	✗	✓

Learning from mistakes

Conclusion:

- exceptions gives nice code
- error codes provides performance and reliability

Next step:

Let's use exception syntax for error codes-like handling.

Idea

using return channel

Let's make throwing exceptions happen using return channels from functions

using return channel

How to achieve that:

- Function needs to return some kind of a `std::variant<T, E>`
- The `sizeof(E)` in above needs to be known upfront
- Each exception must have same interface
- Compiler must know how to `move` the object

Other things to consider:

- co-existence with dynamic exceptions

std::error

features of `std::error` includes:

- trivially-relocatable semantics
- the `error_category` is able to represent:
  - C++ standard library exceptions
  - POSIX system codes
  - Windows' `NSTATUS`
  - and other common error domains

`std::error` type can be treated as a next-gen `std::error_code` type.

The forgotten throws

Since we no longer need to know the type of the exception, we can notify about function throwing the static exception with `throws` static exception specifier

```
void foo() throws {
  throw arithmetic_error::something;
}
```

more examples

```
string foo() throws {
  //dynamic exception will be translated to static one
  throw std::runtime_error;
}
```

```
try {
  auto result = g();
  cout << "success, result is: " << result;
}
catch(error err) { // catch by value is fine
  cout << "failed, error is: " << err.error();
}
```

more examples

```
int caller2(int i, int j) {
  try {
    return safe_divide(i, j);
  } catch(error e) {
    if (e == arithmetic_errc::divide_by_zero)
      return 0;
    if (e == arithmetic_errc::not_integer_division)
      return i / j; // ignore
    if (e == arithmetic_errc::integer_divide_overflows)
      return INT_MIN;
  }
}
```

The END

Thanks for attention!