Error handling in C++

Dawid Pilarski

Current state of error handling

- Error codes description
- Exceptions

Perfect error handling

Improving error handling mechanisms

Type system related error handling improvements

- std::expected
- std::outcome

Embedding new error handling into the language

- 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
}
```

What else can be done to improve the code:

- enums
- error should be an input output argument (passed by reference) to force user handle it

C++11: yet another approach to the error codes

There are 3 types, that C++ 11 added to support `std::error_code`

- `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
  }
  //so stuff
  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 );
```

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

Quality factors of error handling

error codes - quality

exception - quality

error codes vs exceptions

Other languages' error handling mechanisms

C++ strongest attitude - type system

Implementing our own error handling mechanism

our error handling vs exceptions vs error codes

Introducing expected

Usage

Implementation details

Runtime performance

Introducing outcome

Implementation details

Runtime performance

New exception model - idea

d

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.

`std::error`

features of `std::error` includes:

- trivially-relocatable semantics
- size of max 2 pointers
  -

Syntax for new exceptions

- Let's take legacy exception specifications (`throws(typeid ...)`)
- Let's declare a function, that throws exceptions:
  ```
  void foo() throws(std::bad_alloc);
  ```
- Let `throws()` modify the return channel of a function
- now compiler knows, what kind of exceptions can be thrown.
- exceptions now are copied to the callee.

static exceptions specification