Boost.Expected
Google summer of code proposal.

Expected mentor: Vicente J. Botet Escriba

Author: Pierre Talbot

May 3, 2013

Abstract

A good error handle system is hard to design, the first and old return error code by value has been widely used but has serious drawbacks. Most modern languages use now the exception system which is not completely satisfying yet. This proposal is based on the Alexandrescu talk, and it aims to upgrade the return error code mechanism. Furthermore, it proposes a mechanism that will help library designers to make exception-free interface and will let the final user decide whether to throw or not. The main goal is to get this library accepted into Boost.

Contents

1 Introduction

The class template expected$< T, Error >$ proposed here is a type that contains a type T or a type Error in its storage space. The principal purpose of this library is to help the conception of exception free interface. Resuming, it's a class containing an expected T or the error that prevents the creation of the T.

2 Motivation

Back to a pure procedural paradigm, a popular way to signal error to higher layer is to return an error code. However, this mechanism is quite old, and has drawbacks [2]:

1. It monopolizes the return channel.
2. Historically a poor integer describes the error.
3. Unchecked return value can lead to bugs that can be hard to find.

Now, many languages provide the exception mechanism which fixes some of the weaknesses of error code. The exception can be as rich as a class can be, use their own communication channel and provides a stack trace in case of unhandle exception. On the other side, it has also few disadvantages:

1. When thrown, an exception is slow to catch.
2. The exceptional cases are harder to find because exception-based code focus on the normal execution flow [7].
3. Preventing a crash before it happens can be tricky because exceptions are invisible and can be caught in any of the upper layers [7][6].

So expected is a fast error mechanism that can be seen as a modern error return class.

3 Summary of Proposal

The class expected is inspired from the talk "C++ and Beyond 2012: Systematic Error Handling in C++" by Andrei Alexandrescu [1]. The interface has been upgraded by Vicente J. Botet Escriba and myself. The discussion on the Boost mailing list[4] helped a lot too.

3.1 Interface of expected

The class expected takes two template arguments:

```
template<class T, class ExceptionalType=std::exception_ptr>
class expected{};
```

Default construction of expected is private because it must contains an error or a value and shouldn't be empty. The traits class "exceptional_traits" helps to have specific behavior on exceptional cases. Another design could be to have two classes such as expected would only contain a std::exception_ptr and expected_or_error$< T, E >$ would contain a custom error code. However the code/interface of both would be redundant and a single class with a trait class for the few variation points is better here. The following example focuses on the default interface with std::exception_ptr. Some use cases with custom error code are shown in the section 3.2.

We can construct an expected by value, the constructor is not explicit because it's easier to use.

```
expected<int> f()
{
  return 0; // implicit conversion to expected<int>
  // instead of
  // return expected<int>(0);
}
```

We can create expected from value with assignment operator and copy constructor.

```
expected<int> ek = 0;  // assign 0 to ek.
ei = 1;    // assign 1 to ei.
ek = ei;   // assign ei to ek. ek contains 1.
expected<int> el(ek);  // construct el by copy from ek. el contains 1.
```

While the construction by value is implicit, the construction from error is not.

```
expected<int> em(exceptional, std::invalid_argument("bad arg"));  // em
    contains the exception invalid_argument.
ei = make_exceptional_expected<int>(std::invalid_argument("bad arg"));
    // ei contains the exception invalid_argument.
try{
  throw std::invalid_argument("bad arg");
}
catch(...){
  ek = make_exceptional_expected<int>(); // ek contains
      exceptional_traits<expected<int>>::current_exceptional().
}
```

Tag exceptional is required in case we store the error as a value too.

The move semantic is defined accordingly to the move semantic of T and Error, meaning that an expected can be moved if the contained object can move too.

```
expected<int> e = 5; // e contains 5.
expected<int> e2 = move(e); // e2 contains 5. e contains an empty int.

e = make_exceptional_expected(std::runtime_error("bad value")); // e
    contains an error
e2 = move(e); // e2 contains the error and e have an empty error.
```

We can use factory facilities to construct an expected from an error code or even from code. The make_noexcept_expected is specialized for exception error type.

```
expected<int> em = make_noexcept_expected(some_function); // em contains
     the some_function result or the exception thrown.
em = make_noexcept_expected([](){  // em contains the invalid_argument
    exception thrown from the lambda.
 throw std::invalid_argument("bag arg");
});
```

The lifetime of the T and the error is dependent of the lifetime of the expected. When expected is called, the destructor of T or Error is called. We access the value through the value() method or the indirection operator, the former check if the value is contained, and if not, calls bad_access(). The second makes no check and calling *expected on an uninitialized value results in undefined behavior.

```
int i = em.value(); // calls exceptional_traits<expected<int>>::
    bad_access(error).
i = el.value();      // i contains 1.
i = *el;       // i contains 1.
i = *em;       // undefined behavior.
```

We can avoid that value() calls bad_access() by testing if it contains a value before. Furthermore, we can avoid the double check by using the indirection operator.

```
if(em.valid())
  i = *em;
else
  i = 0;   // i contains 0.

if(el)
  i = *el; // i contains 1.
```

```
else
  i = 0;
```

We can emplace value using the emplace tag or the factory function.

```
expected<std::string> en(emplace, "expected string"); // emplace the
    string into expected.
en = make_expected<std::string>("expected_string");   // use the factory
     facility to emplace the string.
```

We provide convenient idiom to handle error and chaining tasks that can fail. The following example shows the connection to a socket[1].

```
expected<int> create_and_connect()
{
  return socket() // call the function socket that create an expected
      handle to a socket.
  .then(connect)  // Pass the expected handle to connect if it contains a
       value, otherwise just return it.
  .on_error(error_resolver); // Use the error_resolver if the result
      expected contains an error, otherwise just return it.
}
```

The "then" method returns the current expected if the function taken in parameter have a void return type. Otherwise it forwards the return type of the function into an expected. By doing so, you can chain treatments and report error handling until you hit the on_error member.

```
expected<int> error_resolver(std::exception_ptr error)
{
  // Try to repare the error.
  if(success_to_repair)
    return repair_value;
  else
    return make_exceptional_expected<int>(error);
}
```

Because the error_resolver still returns an expected, we have chances to repair the error. The "then" member takes function that can be defined as:

```
expected<int> f(int value)
{
  // Do some treatment with value
  if(everything_ok)
    return value;
  else
    return make_exceptional_expected<int>(error);
}
```

But this function can also returns void such as:

```
void f2(int value)
{
  // Do some treatment with value that doesn't not impact the value.
}
```

When calling `e.then(f2).then(f);` the value passed to f and f2 is strictly the same unless f2 accepts it by reference and proceed to changes that can't fail. When chaining the "then" function, we don't break the chaining with void return function. For example:

[1]For the sake of simplicity, arguments to functions are not shown

```
// With the void ability.
e = h().then(f2).then(f).on_error(error_resolver);

// Without the void ability.
e = h();
if(e)
{
  f2(e);
  e.then(f).on_error(error_resolver);
}
else
  e.error_resolver(e);
```

3.2 Custom error code

A common use case is to use custom error code instead of exception.

```
enum MyErrors
{
   PrinterOnFire, DeveloperGoneInsane, ....
};

expected<int, MyErrors> print()
{
  int handle;
  // ...
  if(fire) make_error_expected<int, MyErrors>(PrinterOnFire);
  // ...
  return handle;
}
```

4 Exception safety

The expected type is a wrapper around T so most of the constructors/assignment and swap method throw if some methods of T throw. The containing exception can be thrown in the value() method only. Note that all the methods in the synopsis annotated with "noexcept( )" can throw if the corresponding method of T throw.

5 Synopsis

5.1 "Must" features

This interface shows which features must be in Boost.Expected.

```
namespace boost{

// Construction from exception
struct exceptional_tag {};
constexpr exceptional_tag exceptional = {};

// In-place construction
struct emplace_tag {};
constexpr emplace_tag emplace = {};

template <class T, class ExceptionalType=std::exception_ptr>
class expected
{
public:
  typedef T value_type;
  typedef ExceptionalType error_type;

  // Constructors
  constexpr expected(const T&) noexcept(    );
  constexpr expected(T&&) noexcept(    );
```

```
  expected(const expected&);
  expected(expected&&);
  constexpr expected(exceptional_tag, ExceptionalType const&) noexcept;
  template <class E>
  constexpr expected(exceptional_tag, E const &e) noexcept;
  template <class... Args>
  constexpr explicit expected(emplace_tag, Args&&... args) noexcept(    );

  // Destructor
  ~expected();

  // assignment
  expected& operator=(const expected&);
  expected& operator=(expected&&);
  expected& operator=(const T&) noexcept(    );
  expected& operator=(T&&) noexcept(    );
  template <class... Args> expected& emplace(Args&&...);

  // swap
  void swap(expected&) noexcept(    );

  // observers
  constexpr T const& operator *() const noexcept;
  T& operator *() noexcept;
  constexpr explicit operator bool() const noexcept;
  constexpr T const& value() const;
  T& value();
  constexpr bool valid() const noexcept;

private:
  union {   // exposition only
    error_type excpt;
    value_type value;
  };
  bool has_value; // exposition only
};

// Specialized algorithm
template <class T>
void swap(expected<T>& x, expected<T>& y) noexcept(noexcept(x.swap(y)))

// Factories
template <typename T, typename ErrorType>
expected<T, ErrorType> make_error_expected(ErrorType const&) noexcept;

template <typename T, typename ErrorType, typename E>
expected<T, ErrorType> make_error_expected(E const&) noexcept;

template <typename T>
expected<T> make_exceptional_expected() noexcept;

template <class T, class... Args>
expected<T> make_expected(Args&&...) noexcept(    );

// Factory specialized for the exception_ptr error.
template <typename F>
expected<typename result_of<F()>::type>
make_noexcept_expected(F&& fuct) noexcept;

template <typename T>
expected<T> make_exceptional_expected(std::exception_ptr const&) noexcept
    ;

template <typename T, typename E>
expected<T> make_exceptional_expected(E const&) noexcept;

} // namespace boost
```

5.2 Traits class

```
namespace boost{
// Traits classes

template <typename ExceptionalType>
struct exceptional_traits
{
  typedef ExceptionalType exceptional_type;

  template <class E>
  static exceptional_type make_exceptional(E const&);

  static exceptional_type current_exceptional();

  static void bad_access(const exceptional_type &);
};

// Specialization for exception_ptr
template <>
struct exceptional_traits<std::exception_ptr>
{
  typedef std::exception_ptr exceptional_type;

  template <class E>
  static exceptional_type make_exceptional(E const&);

  static exceptional_type current_exceptional();

  static void bad_access(const exceptional_type &);

  template <class E>
  static bool has_exception(bool, exceptional_type) const noexcept;
};
}// namespace boost
```

5.3 “Should” features

```
namespace boost{

template<class T, class ExceptionalType=std::exception_ptr>
class expected
{
  // ...

  // utilities
  // if F has a void return type.
  template <typename F>
  enable_if<
    is_void<typename result_of<F(const T&)>::type>::value,
    expected<T>
  >::type then(F&&) noexcept;

  // if F has a non-void return type.
  template <typename F>
  enable_if<
    !is_void<typename result_of<F(const T&)>::type>::value,
    expected<typename result_of<F(const T&)>::type>
  >::type then(F&&) noexcept;

  // if ErrorResolver has a void return type.
  template <typename ErrorResolver>
  enable_if<
    is_void<typename result_of<ErrorResolver(const ExceptionalType&)>::
        type>::value,
    expected<T>
  >::type
  on_error(ErrorResolver&);

  // if ErrorResolver has a non-void return type.
  template <typename ErrorResolver>
```

```
  enable_if<
    !is_void<typename result_of<ErrorResolver(const ExceptionalType&)>::
        type>::value,
    typename result_of<ErrorResolver(const ExceptionalType&)>::type
  >::type
  on_error(ErrorResolver&);
};
} // namespace boost
```

5.4 "Could" features

```
namespace boost{

template<class T, class ExceptionalType=std::exception_ptr>
class expected
{
  // ...

  template <typename... F>
  /* unspecified */ then(F&& ...) noexcept;
};

template <class... Expected>
/* unspecified */ if_all(Expected&& ...) noexcept;
} // namespace boost
```

The "then" method can takes more than one function and returns a tuple-like with all the results or the first error that prevents the creation of one of these results. Example:

```
// With then chaining.
e.then(h).then(f, g).then(h2).on_error(error_resolver);

// With exception try-catch.
try{
  h2(f(h()), g(h()));
} catch(...) {
  error_resolver();
}
```

The "if_all" method can takes one or more expected results and returns a similar result. Example:

```
// With the if_all facility

if_all(f(), g()).then(h).on_error(error_resolver);

try {
  h(f(), g());
} catch(...) {
  error_resolver();
}
```

Finally, a method could be add to the trait class:

```
template <typename ExceptionalType>
struct exceptional_traits
{
  // as defined above
  // ...

  template <class E>
  static ExceptionalType catch_exception(); // catch it with
      current_exception.
};
```

This can be useful for the non-exception expected. For example, the library doesn't enable users to build exception free expected$< T, E >$ from code. Because even if we caught the exception, we can't store it, we don't know what to do. This would be the role of the catch_exception method. It could also help for the "then" method. A possible implementation of the "then" method could be:

```
template <typename F>
expected<typename result_of<ErrorResolver(const ExceptionalType&)>::type>
then(F&&) noexcept
{
  typedef typename result_of<ErrorResolver(const ExceptionalType&)>::type
       result_type;
  if(valid())
  {
    try{
      return f(value);
    }
    catch(...)
    {
      // if ExceptionalType is std::exception_ptr we can store this
          exception inside, otherwise we have no choice that rethrow...
      // The following call to catch_exception could help to design free
          exception code, even with error code expected.
      return make_exceptional_expected<result_type>(
  exceptional_traits<ExceptionalType>::catch_exception());
    }
  }
  else
    return make_exceptional_expected<result_type>(error);
}
```

6 Personal Details

- **Name:** Pierre Talbot
- **College/University:** University of Lyon 1 (France)
- **Course/Major:** Computer science
- **Degree Program:** Master degree (first year)
- **Email:** ptalbot [at] mopong [dot] net
- **Homepage:** None

7 Availability

- *How much time do you plan to spend on your GSoC?*

  At least 35 hours per week, but from past experience it will be as much as needed.

- *What are your intended start and end dates?*

  I'd begin just after my final exams, the 24th June. My end date should be the GSoC end date, but I'll work more on week-end the last two weeks because I restart school.

- *What other factors affect your availability (exams, courses, moving, work, etc.)?*

  Maybe a two or three days holiday during July or August.

8 Background Information

- *Please summarize your educational background (degrees earned, courses taken, etc.).*

  I started to study computer science in Belgium for a three-years diploma, I left after my second year to join the University of Lyon 1. The two first years were practical-oriented, I mainly studied:

  - Programming languages (deeply study): C, Java.
  - Programming languages (initiation - 24 hours per course): C++, C#, web languages (html/css, PHP, Javascript).
  - Others: database (Oracle, SQL), data structure, OOP, UML, basic networking.

  The past two years in Lyon have shown me the theoretical side of computer science.

  - Advanced data structure (red-black tree, skip-list, graph).
  - Mathematical logic (applications: SAT solver, prolog, datalog), Turing machines and automaton.
  - Programming languages (advanced): C++ (generic programming), web-oriented Java (JSP, servlet,...).

- *Please summarize your programming background (OSS projects, internships, jobs, etc.).*
  - **January-february 2013**: Research work at the University. I upgraded the combinatorial map[5] library in *CGAL* to be compatible with some Boost.Graph concepts and others. Users should be able to use it with the *Triangulated Surface Mesh Simplification* algorithm[3]. My mentor was Guillaume Damiand.
  - **Summer 2012**: Internship of 4 months at Datakit to work on reversing the Solidworks (CAD software) files. The code was written in C++. I successfully completed the internship with five PMI (entity annotation) reversed and few others code improvements (such as the use of smart pointer). My mentor was Clément Dumas.
  - **Summer 2011**: Google Summer of Code in the Boost organization with the Boost.Check project. Even if the final version was usable, I'm still working on it before a review request. My mentor was Paul Bristow.

- *Please tell us a little about your programming interests. Please tell us why you are interested in contributing to the Boost C++ Libraries.*

  Since I worked on the Boost.Check project I know that I like working on library. I prefer well-design code to quick-and-dirty code. When programming and thinking on what is a good design, I feel like I learn a lot. Otherwise, I didn't found what are my programming interest yet. I just know that, for any projects, I like to think on the design.

- *What is your interest in the project you are proposing?*

  During few weeks I think on how to return a value or the error that preventing the creation of the value. It was for my Boost.Check project, I finally used Boost.Optional without being completely satisfied. When I watched the Expected video by Alexandrescu I knew that it was a missing features. I hope to use this library in Boost.Check.

- *Have you done any previous work in this area before or on similar projects?*

  I read some articles but I'm far from an expert in error handling. Thinking about it during many weeks because of Boost.Check helped to see the design issues of such a class.

- *What are your plans beyond this Summer of Code time frame for your proposed work?.*

  I'll work on it until it accepts into Boost. After this, I will continue the Boost.Check library using Boost.Expected. I can't throw my projects to the trash bin, even if it takes years, I want to complete them.

- *Please rate, from 0 to 5 (0 being no experience, 5 being expert), your knowledge of the following languages, technologies, or tools:*

  - **C++**: 3.5. I know many language features but I still learn new everyday.
  - **C++ Standard Library**: 4. I use it every time when I program in C++.
  - **Boost C++ Libraries**: 3. There are a lot of Boost libraries that I never used, the ones I already used are the BGL, MPL, Asio, Spirit and some utilities (such as bjam and Quickbook).
  - **Subversion**: 3. Knowledge to use it for an every day usage. I already used the Boost SVN.

- *What software development environments are you most familiar with (Visual Studio, Eclipse, KDevelop, etc.)?*

  I mainly develop on Linux with Sublime Text 2 for code writing. I compile in the console. But I also used Visual Studio during my internship and for the GSoC 2011.

- *What software documentation tool are you most familiar with (Doxygen, DocBook, Quickbook, etc.)?*

  I'm familiar with Doxygen and Quickbook, I didn't use DocBook a lot thanks to Quickbook.

9 Proposed Milestones and Schedule

Date	Description
May - June	Further refinements on the design decisions through the Boost mailing list.
24/06 to 01/07	Start of GSoC. Implementation based on the design decisions.
29/06 to 05/07	Discussion of the implementation with my mentor on the mailing list. Correction of the design details.
06/07 to 15/07	Add tests, examples and documentation features.
16/07 to 29/07	Test the code and correct it on the different compilers.
06/07 to 29/07	Discussion and code correction to fit the design decisions.
20/07 to 02/08	"Boostify" the part of the code that are not yet. Learning the Boost technicals to made the code portable.
03/08 to 10/08	Invite developers to use Boost.Expected and ask feedback. Bugs correction. Maintain the documentation and tests.
11/08 to end	Ask a Boost review. Discuss with the community and find solutions to inconvenient issues.

10 Acknowledgement

This proposal wouldn't look like this or even exists without these people:

- Andrei Alexandrescu for his talk about Expected.
- Vicente J. Botet Escriba for all the good feedback and discussions on Expected.
- All the others people from the Boost mailing list.

References

[1] A. Alexandrescu. C++ and beyond 2012 - systematic error handling in c++, 2012. http://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Andrei-Alexandrescu-Systematic-Error-Handling-in-C.

[2] Ned Batchelder. Exceptions vs. status returns, 2003. http://nedbatchelder.com/text/exceptions-vs-status.html.

[3] F. Cacciola. Cgal manual, chapter 53, triangulated surface mesh simplification, 2007. http://www.cgal.org/Manual/latest/doc_html/cgal_manual/Surface_mesh_simplification/Chapter_main.html.

[4] Boost community. Gsoc-2013: Boost.expected, 2013. http://boost.2283326.n4.nabble.com/gsoc-2013-Boost-Expected-td4645271.html.

[5] G. Damiand. Cgal manual, chapter 27, combinatorial maps, 2011. http://www.cgal.org/Manual/latest/doc_html/cgal_manual/Combinatorial_map/Chapter_main.html.

[6] Raymond. Cleaner, more elegant, and harder to recognize, 2005. http://blogs.msdn.com/b/oldnewthing/archive/2005/01/14/352949.aspx.

[7] Joel Spolsky. Exceptions, 2003. http://www.joelonsoftware.com/items/2003/10/13.html.