D0762R0: Concerns about `expected<T, E>` from the Boost.Outcome peer review

Document #:	D0762R0 draft 2
Date:	2017-10-1
Project:	Programming Language C++ Library Evolution Working Group

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

Todo:
- Awaiting new todo items.

In May 2017 one of the liveliest, longest and most detailed peer reviews in some years was held at the Boost C++ Libraries regarding a proposed Outcome library (https://ned14.github.io/outcome/). Over 800 contributions were made to the review, and discussion continued long after the formal end of the review for something approaching three weeks in total.

The v1 library was rejected, but with a surprising amount of consensus on what a v2 design should look like, and specifically on what `expected<T, E>` should **not** look like.

This report summarises the author's best personal interpretation of those 800+ review contributions. Unlike the official peer review summary report which can be found at https://lists.boost.org/boost-announce/2017/06/0510.php, this review also presents the author's v2 design which he believes to meet the consensus opinion of the Boost peer review.

I should emphasise that Vicente, who is the champion of the Expected proposal (currently [P0323]), was a major contributor to the Boost.Outcome v1 peer review. The concerns raised by this paper are partially points of design disagreement between me and him, partially between Vicente and the Boost peer review where Vicente disagrees with the Boost majority opinion, but also partially points of disagreement between the Boost peer review consensus opinion and WG21's consensus opinion (i.e. Vicente disagrees with WG21 on the same points as Boost did). I don't claim that the concerns raised in this paper are a perfect rendition of the Boost peer review consensus: gauging a single consensus design from 800+ pieces of often opposing feedback is more of an art than a science. But I do think that WG21's present chosen design for Expected is sub-optimal, and could be better.

My hope is that this paper will spur a reconciliation of design for Expected between the Boost and WG21 consensus opinions such that the best possible design moves forward for standardisation.

Contents

1 Summary of design differences **2**

1.1 Similarities 2
1.2 Dissimilarities 3

2 Discussion of concerns on design differences in priority order 5
2.1 All-wide or all-narrow observers 5
2.2 Self-disabling implicit constructors 6
2.3 Standard layout propagation 7

3 Conclusion 7

4 Acknowledgements 7

5 References 7

1 Summary of design differences

Outcome v2's most similar objects to Expected are called `unchecked<T, E>` and `checked<T, E>`, both of which are template aliases to a simplified `result<T, E>`. `checked<T, E>` throws a `bad_result_access` or `bad_result_access_with<E>` in its wide contract observers. `unchecked<T, E>` has all-narrow observers, and thus throws no exceptions at all.

1.1 Similarities

These are the following similarities between `expected<T, E>` and `checked<T, E>`:

1. Both are simple vocabulary types representing either a `T` or an `E`.
2. Both implicitly construct from a `T` or something constructible to same.
3. Both make available their `T` via `.value()` and their `E` via `.error()`.
4. Both provide a strong never-empty guarantee.
5. Both provide a `.has_value()` and an explicit boolean operator test so `if(expected) ...` works.
6. Both provide type sugar for wrapping a `T` or an `E` to give it unambiguous convertibilty.
7. Both are constexpr friendly.
8. Both preserve triviality of copy, move, assignment and destruction of `T` and `E`.
9. Both permit `T = void`.
10. Both permit `const` and `volatile` types which can be very useful sometimes.
11. Both permit implicit or explicit construction from dissimilar instances where both `T` and `E` are implicitly or explicitly constructible.

12. Both throw a C++ exception type, with logic error semantics, when one violates a wide contract observer (e.g. calling `.value()` when there is no value available).

Both Checked and Expected are very simple vocabulary types, and at first glance look to be almost identical. There are however quite a lot of differences, most of which stem from the design choices made by a majority of Boost peer review contributors.

1.2 Dissimilarities

These are the dissimilarities:

1. Checked (and everything in Outcome) replicates `std::variant<...>`'s constructor design instead of `std::optional<T>`'s i.e. we implicitly construct from either a `T` **or** an `E`, or any pattern which *could* construct to a `T` or an `E`, *where it is unambiguous* (if it *could* be ambiguous, all implicit constructors self-disable).

   Expected permits implicit construction from `T` and `unexpected<E>` only.

2. Checked's `.value()` and `.error()` have **wide** contracts, with clearly delineated alternative narrow contract observer functions `.assume_value()` and `.assume_error()`. As mentioned earlier, a different template alias `unchecked<T, E>` has all-`narrow` contract observers.

   Expected has a wide `.value()`, but a narrow `.error()` i.e. access to `.error()` where there is no unexpected value is undefined behaviour (narrow contract).

3. Checked does not provide `operator*()` nor `operator->()` value observers as it does not model `optional<T>` like Expected does.

4. Checked always carries the `[[nodiscard]]` attribute to ensure the programmer checks the returned value.

   Expected currently does not, and doing so may not be appropriate for some choices of type `E` given Expected's wider intended use case as a primitive building block for other constructs.

5. Checked is never default constructible, thus always forcing the user to specify success or failure or `optional<result<T>>` to indicate potential emptiness or not-success-not-failure.

   Expected is default constructible if `T` is default constructible.

6. Checked propagates standard layout-ness of `T` and `E`, and is thus intentionally designed for use from C code as improved interoperation between C++ and C code. We require this specific layout to be guaranteed:

```
struct
{
  T value;
  // flags bit 0 set if value contains a T instance (and E is to be ignored)
  // flags bit 1 set if value does not contain a T (and E is to be observed)
  // flags bit 4 set if errcode is a generic POSIX errno (std::generic_category, std::errc enum)
  unsigned int flags;
  E error;
};
```

Historically the C++ standard tries to avoid dictating implementation specifics, and thus neither does the Expected proposal. I will come back to this point in detail later on why this might need to change.

7. Checked provides a `.has_error()` so people write what they mean descriptively, thus reducing cognitive load on those reading the code.

8. Checked requires type `E` to be default constructible.

9. Checked's clarifying type sugar is called `success<T = void>` and `failure<E>`. This lets you unambiguously return either success or failure from a Checked returning function in a cognitively undemanding expression:

```
checked<std::string> get_home_directory() noexcept
{
  if(const char* x = std::getenv(n))
    return success(x);  // could write return x;
  else
    // implicitly converts to std::error_code
    // could return std::errc::no_such_file_or_directory directly
    return failure(std::errc::no_such_file_or_directory);
}
```

Expected's type sugar is `unexpected<E>` which maps almost identically onto `failure<E>`. Expected has no corollary to `success<T = void>` as it does not permit implicit construction from `E`.

10. As mentioned earlier, Outcome provides the same implicit constructor design as `std::variant<...>` which allows some particularly elegant and succinct usage:

```
checked<HANDLE> open_file(std::filesystem::path path) noexcept
{
  HANDLE h = CreateFile(path.c_str(), GENERIC_READ, 0, NULL, OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL, NULL);

  // std::error_code constructs from { int, const std::error_category & }
  if(INVALID_HANDLE_VALUE == h)
    return { (int) GetLastError(), std::system_category() };
  return h;
}
```

Expected only provides implicit construction for expected inputs, and thus would require one to `return unexpected(GetLastError(), std::system_category());`. As much as that looks to be only an extra 'unexpected' in there, remember that braced initialisation aggregates, and that becomes tedious after a while when returning nested braced initialisations.

2 Discussion of concerns on design differences in priority order

2.1 All-wide or all-narrow observers

Conclusively resolving the debate between wide vs narrow observers, as anyone serving on WG21 for any time is well aware, is unsolvable. Sometimes you want wide contracts (where at runtime incorrect usage is detected and an exception thrown), sometimes you want narrow contracts (where at runtime incorrect usage is not detected, thus allowing tooling like the undefined behaviour sanitiser or valgrind to trap the incorrectness). Wide contracts require code to handle exception throws, whilst narrow contracts allow code to assume no exception throws, usually leading to lower development and testing costs, and more auditable logic. Narrow contracts can sometimes be checked for correctness by static analysis, wide contracts never can be as throwing an exception may be the programmer intended behaviour.

Expected, like Optional, provides mostly narrow observers with the glaring exception of `.value()`, which is wide. I'll freely admit that I consider this design choice to be the **worst** of both worlds, not the best of both worlds. Arthur said on std-proposals that WG21 historically uses English words for wide observers e.g. `.at()` and punctuation for narrow observers e.g. `operator[]()`. If that was indeed the convention, then it needs to change because that rule is neither intuitive nor able to be applied properly to Expected. And besides, we have far more powerful correctness validation tooling available now such that wide contract observers are a bad way of detecting program logic errors given the better alternatives now available.

I appreciate that my experience with Optional is not as extensive as others, but I have faced a codebase in the past using Optional which was written by someone who was not aware that `operator*()` and `operator->()` and `.value()` do not have the same contract. They had used each interchangeably, sometimes apparently relying that `operator*()` will throw on incorrect use, sometimes losing exception safety because they did not account for `.value()` being able to throw.

It's easy to say 'bad programmers write bad code', but I think that's a cop out. Firstly, programmers tend to be lazy, and like to press fewer keys. So they'll tend towards `operator*()` once they realise it's there. Before that they probably used `.value()` because it's got the word 'value' in it, so it stuck out initially. Where we end up is with **incorrectness**, but *the worst possible kind* of incorrectness because it's virtually impossible to detect without someone knowledgeable to audit the code, and it's especially expensive to fix because somebody must go reason about every use case and replace with the appropriate observer (or throw the whole thing away like I did, and rewrite again from scratch with repeated comments sprinkled saying *'do NOT use value(), not once, not ever in this code'* as a warning to future maintainers.

The ship has sailed on Optional. But we have no good reason to repeat this design mistake with Expected.

For simple vocabulary objects like these, a better new convention for WG21 to follow is to provide a 'primary' set of observers which are either all-wide or all-narrow, and then a 'secondary' set of observers with a clearly delineated naming convention which are all exactly the opposite. Like Outcome does with `.value()`/`.assume_value()` and `.error()`/`.assume_error()`. And given how program logic errors are best detected by static analysis, sanitisers etc., the presumption ought to

be 'narrow contracts unless programmer benefiting reason for wide contracts'.

So I would really urge that Expected's `.value()` become narrow like Unchecked's `.value()`, perhaps with free function forms being checking editions i.e. `value(expected)` is wide, `expected.value()` is narrow. That would make all of Expected's primary observers narrow, and thus generate least surprise when this object gets used in the wild by ordinary programmers.

2.2 Self-disabling implicit constructors

Most on WG21 will quite rightly feel that implicit construction is dangerous, and that Expected has made the right choice and Checked has not. However Checked's implicit constructors are not unconstrained, and in fact the Boost peer review went into very considerable depth as to exactly what constraints to impose in order to avoid implicit construction surprises like[1]:

```
// No, we do not initialise a string here!
std::variant<std::string, int, bool> mySetting = "Hello!";
```

Specifically the Boost peer review came up with these constraints for any of the implicit constructors to be enabled[2]:

- `T` is not constructible from `E`.
- `E` is not constructible from `T`.

This may seem to be overly severe. However, consider the overwhelmingly most likely choices for type `E` in users of Checked:

- `std::error_code`.
- `std::exception_ptr`.
- A type for which `std::is_error_code_enum_v<E>` is true or `std::is_error_condition_enum_v<E>` is true.

This is the case because Checked (and all the types in Outcome) is specifically intended for use as a function return type, not as a generic vocabulary primitive type like Expected. We therefore can over-eagerly disable implicit construction because very few choices of type `T` being returned from functions will be constructible into any of the types above, or vice versa. And this has been verified in a limited amount of empirical testing.

None of this implies that Expected has made the wrong design choice for its constructor design. It is provided as food for thought with regard to how alternative motivations lead to alternative designs.

And I ask: what precise problem is WG21 solving with the present Expected design? Because Outcome solves a precise problem: returning success or failure from functions with predictably

[1] It's expensive on compile times, but https://github.com/cbeck88/strict-variant is an example of how this sort of surprise could be avoided.

[2] I added an exception to the peer review recommendations in Outcome v2: We ignore this requirement if `T = bool` and `E` is one of the common error types as otherwise we see over eager implicit construction disabling if `E` is boolean testable, which is the case for the common error types.

low worst case execution times. That leads us, indirectly, to being able to provide safe implicit construction.

2.3 Standard layout propagation

Yet to write

3 Conclusion

Yet to write

4 Acknowledgements

- Almost all of the content, ideas, critiques and discussion points above came originally from the Boost peer review of proposed Boost.Outcome which led to the v2 Outcome design, and the Result object design discussed in this paper. That review was one of the most valuable I've seen at Boost in many years, it ranged both wide and deep over three plus weeks, and as much as it was very hard work, that kind of peer review really is engineering at its highest calibre. Thank you boost-dev.
- Vicente J. Botet Escribá for all the work he has done on making Expected possible, and being such a good sport when I from time to time criticise his Expected design which has been ongoing now for three years. I hope that he finds the above useful whatever happens to Result and Expected as far as standardisation goes.
- Andrzej Krzemienski for his ongoing and extensive contributions to Outcome's design and development.
- Charley Bay for review managing the Outcome peer review.
- Peter Dimov for suggesting that `result<T, E>` could be standards material. I hadn't considered the merit of it before.
- Paul Bristow who proposed the name "Outcome" for the library after a very extended period of bike shedding on boost-dev.
- Michael Park for making available this LaTeX template at https://github.com/mpark/wg21/.

5 References

[P0650] Vicente J. Botet Escribá,
C++ Monadic interface
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0650r0.pdf

[P0323] Vicente J. Botet Escribá, JF Bastien,
A proposal to add a utility class to represent expected object (Revision 5)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0323r3.pdf

[P0262] Lawrence Crowl, Chris Mysen,
A Class for Status and Optional Value
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0262r0.html