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


| | |
|---|---|
| Document #: | D0762R0 draft 2 |
| Date: | 2017-10-4 |
| Project: | Programming Language C++<br>Library Evolution Working Group |


| | |
|---|---|
| Reply-to: | Niall Douglas<br><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

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 error contains a generic POSIX errno int (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 Struct-based rather than Union-based storage

One of the more surprising things which emerged from the Boost peer review of Outcome was the large minority who backed struct-based storage rather than union-based storage. This might seem strange to people, so it is worth explaining.

One initially might think that union-based storage would be superior to struct-based storage for these sort of `T` or `E` objects. The obvious advantage is reduction of space consumed, plus it might be pleasing to some that memory representation equals logical representation.

However there are significant costs to union-based storage for this sort of simple object. Outcome was designed to be used in the public interface files of *really* big code bases, ones where every `#include` in an interface header file is benchmarked before and after for effects on build times because it can make *hours* of difference. Outcome, because it has to drag in `<system_error>` which in turn drags in `<string>` and the full STL memory allocation and exception throwing infrastructure, already stands at a disadvantage[1].

However the more pressing concern for me the library designer is codebases which use these objects as the return type for every single function in the codebase, including all the internal ones, must end up instantiating, copying, moving, assigning and destructing them **a lot**. How hard the compiler must work to generate assembler for these objects is therefore of paramount importance to sane build times.

As it happens, I have two libraries which use Outcome very heavily throughout, including in extensive metaprogrammed constructs such as long multi-nested initialiser lists of auto-generated parameter permutations for test cases which contain Outcome types. These are quite hard on the compiler, none of clang, GCC nor MSVC shine when compiling the test suite for these libraries, and precompiling headers makes little difference. But it used to be much worse – three plus years ago I originally started using Boost.Expected, but found its compile time impact unacceptable, and thus became motivated to write Outcome.

Outcome v1 used every semi-legal trick in the book to bring down compile times, most of which the Boost peer review rejected for the hacks that they were. Outcome v2, still needing to be minimum compile time impact, ended up with struct-based storage which enabled simple forwarding implementations of copy, move, assignment, swap and destruction. The only compile-time intensive part is non-copy-move construction where extensive lists of traits must be calculated and a long list of either SFINAE or Concepts evaluated in order to implement `std::variant<...>`'s intelligent

[1] My claim on this is that C++ Modules should eliminate this particular disadvantage for Outcome, thus exposing Outcome's specific compiler loading at some point in the future.

constructor design[2].

Bringing this back to Expected, I feel concern regarding the compiler load that implementing strong never-empty guaranteed union storage must impose on the compiler precisely because for every copy, move, assignment and destruction – unless all the types are trivial – you must get the compiler to execute significantly more work than a struct-based design in order to generate assembler. Early straight after the peer review I knocked together an experimental Outcome based on `std::variant<...>`, and found my compile times ballooned unacceptably by more than fourfold. Of course, Expected is not Variant, nor would need all of Variant to be implemented, so this is not a fair comparison. But I still find it concerning.

Vicente tells me that he is open to making Expected utilise struct-based storage instead of union-based storage where certain conditions are fulfilled e.g. all types are trivial or have standard layout, or maybe some trait is satisfied, or maybe simply `sizeof(E) <= 64`. I would still worry that until C++ Modules land, that just means even more code for the compiler's parser to grok through per compiland, but I can see that precompiling headers would deliver significant benefits in this situation.

To conclude this point, my recommendation is that Expected either adopt all-struct storage, or only utilise union-based storage when circumstances warrant it e.g. `sizeof(E) > 64`. I personally think the former option easier, simpler, and much less problematic than some might think. After all, how often in actual real world code will `sizeof(E) > 64`? And if it does, why not just encourage the programmer to store extended error state via malloc and transported with `E = std::exception_ptr` instead?

### 2.3 Standard layout propagation

It's not been historically felt important for standard layout-ness to be propagated intact e.g. `std::optional<T>` provides no guarantee that if `T` has standard layout, then so will `std::optional<T>`. Similarly, `expected<T, E>` currently makes no guarantee of propagation of this quality either i.e. if both `T` and `E` have standard layout, then so shall `expected<T, E>` and its layout will be (for example):

```
struct
{
  union
  {
    T value;
    E error;
  };

  // I'd actually recommend bit 0 = has value, bit 1 = has error
  // for improved robustness in case of memory corruption or
  // use of uninitialised bytes rather than a simple boolean in bit 0.
  // It also aids C code supplying debuggable Expected's to C++ code.
  unsigned char has_value;

```

[2]You might be interested to learn that my libraries once ported to v2 regressed compile times over v1 Outcome by quite a bit, about 15%, which is enough to be noticeable.

```
  // unknown standard library metadata may follow here i.e.
  // C code cannot assume this C struct represents the whole
  // C++ object
};
```

Standard layout-ness doesn't particularly matter to C++ code usually, but it matters hugely to code which can speak C such as Python, Rust, Microsoft COM and a long list of various programming languages with a C foreign function interface (FFI). These languages generally interoperate poorly with C++, requiring the use of bindings generators and other complex interoperation tooling to arbitrate between C++ and their usually C-based FFI layer.

I find this a missed opportunity. There is zero good reason why the C++ standard cannot specify that if `T` has standard layout, then so will `std::optional<T>` and its layout will be:

```
struct
{
  T value;
  _Bool has_value;
};
```

Then I can write this in C++ 17:

```
static HANDLE h;
extern "C" std::optional<HANDLE> get_handle() noexcept
{
  if(INVALID_HANDLE_VALUE == h)
    return {};
  return h;
}
static_assert(std::is_standard_layout_v<std::optional<HANDLE>>);
```

And in C11 I can write this:

```
struct optional_HANDLE
{
  HANDLE value;
  _Bool has_value;
};

extern struct optional_HANDLE get_handle();

int readdata()
{
  struct optional_HANDLE h;
  h = get_handle();
  if(!h.has_value)
  {
    return EBADF;
  }
  ReadFile(h.value, ...
}
```

And no complex bindings tooling like SWIG[3] needed!

[3]Simplified Wrapper and Interface Generator http://www.swig.org/.

The ship has sailed for Optional, particularly as both `{ bool has_value; T value; }` and `{ T value; bool has_value; }` layouts are in use in the major STLs. But it is not too late for Expected, it can implement layout guarantees without breaking any existing ABIs.

Some have felt that adopting this route starts down a slippery slope, and soon the C++ standard will end up dictating implementation with lots of negative consequences. I'm not saying the standard should do this except where it adds significant value to the C++ userbase to do so. Right now the lack of propagating standard-layoutness where it would be trivially easy for the C++ standard to do so is making life harder for anybody outside C++ to reuse code written in C++, or for C++ users to write a C++ library with a C-compatible external API.

If it isn't trivially easy to standardise layout for something, or if there is any potential negative consequence on standard library implementers, then don't do it! Though do note that standard library implementers can still append custom data of their choosing without breaking the layout guarantee.

But equally if it costs nobody anything, then let's not be anti-social to C and other programming languages just for the sake of it!

### 2.4 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[4]:

```
// 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[5]:

1. `T` is not constructible from `E`.
2. `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:

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

[4] 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.

[5] 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.

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 in those libraries using Outcome extensively that I mentioned earlier.

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 low worst case compile and execution times. That leads us uncontroversially to the chosen design.

# 3 Conclusion

I don't want anything in this paper to prevent [P0323] Expected from landing in the C++ 20 standard. Equally, I think it would benefit greatly if its `.value()` could not throw `bad_expected_access<E>` i.e. it stops being a runtime checked type entirely, and leaves correctness checking up to the excellent free of cost static analysis tooling like clang-tidy and the runtime sanitisers like the undefined behaviour sanitiser instead.

The result would be an even simpler Expected than at present, one more closely matching Outcome's `unchecked<T, E>` which is the simplest type Outcome has, and is the foundation stone upon which the rest of Outcome's layering of increasingly complex types is built.

Furthermore, if Expected used struct-based storage, it would become even simpler again with probably less compiler load, something likely highly important to 'big iron' C++ users when considering whether to use this type in a multi-million line codebase or not.

Finally, if you have accepted these previous two recommendations, then my final recommendation that Expected preserve standard layout-ness of its types `T` and `E` becomes pretty easy to accept, and a great boon for improving C++ interoperation with C-speaking programming languages.

The only design difference where I do not have a specific recommendation for Expected is regarding implicit constructors. I find unconstrained implicit construction to be too dangerous, which is why Outcome disables them where there is any chance that `T` and `E` could get confused. That is overkill for Expected which must reasonably support `expected<int, int>` just as much as dissimilar types. I therefore find the present constructor design for Expected acceptable.

# 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.
- Andrzej Krzemienski for his ongoing and extensive contributions to Outcome's design and development.
- Charley Bay for review managing the Outcome peer review.
- 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