P0762R0: Concerns about `expected<T, E>` from the Boost.Outcome peer review

Document #:	P0762R0
Date:	2017-10-15
Project:	Programming Language C++ Library Evolution Working Group

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

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 considers the author's v2 design which he believes to meet the consensus opinion of the Boost peer review. Based on those, I then make three specific recommendations for changes to Expected in my personal priority ordering of importance.

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. His inspiration and feedback has greatly influenced Outcome's design, and I think it safe to say that Expected has seen a few changes too in return.

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 not ideal, and it 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

Many people both in and outside of WG21 have asked for a detailed list of design similarities and differences between Expected and Outcome.

Outcome v2, like most Boost libraries, provides lots of facilities and objects, including a complete solution to how the Filesystem TS `error_code` overloads do not return the paths related to the failure like the throwing overloads do. But we need not consider that for this paper: Outcome v2's most similar object to Expected is called `unchecked<T, E>`, which is a template alias to a simplified `result<T, E>`.

Unlike Expected, `unchecked<T, E>` has all-narrow observers, and thus throws no exceptions at all. It is undefined behaviour to observe a state which is not present.

I know a lot of people would like much more detail on Outcome than just that above, but I do not think it appropriate to write such detail to WG21 until the v2 design has passed a second round of Boost peer review[1]. Suffice it to say that Outcome's `unchecked<T, E>` is a very close substitute to the `expected<T, E>` in [P0323]. This is because both designs have moved significantly closer to one another even since the Toronto meeting this summer, in part thanks to the Outcome v1 peer review.

The differences listed below therefore represent what I feel the Boost peer review majority disagreed with Expected's design. We will find out in the second peer review, currently expected for January 2018, if I called those 800+ contributions to the first review right or not.

1.1 Similarities

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

[1] The really keen may find the reference documentation at https://ned14.github.io/outcome/standardese/doc_result.html#standardese-outcome_v2_xxx::result%3CR,S,NoValuePolicy%3E of use.

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 a `T` which is neither copyable nor movable.
12. Both permit implicit or explicit construction from dissimilar instances where both `T` and `E` are implicitly or explicitly constructible.

Both Unchecked 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 my reckoning of a majority of Boost peer review contributors.

1.2 Dissimilarities

These are the dissimilarities:

1. Unchecked (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. Unchecked's `.value()` and `.error()` have **narrow** contracts. It is *undefined behaviour* to use them when no value or error respectively is present. The conditions where incorrect usage of them is UB is told to the compiler, whose static analyser will spot the logic error at compile time and whose undefined behaviour sanitiser will spot the logic error at runtime.

   Expected has a wide `.value()` which throws a logic error type exception, but all the other observers are narrow contract same as Unchecked.

3. Unchecked does not provide `operator*()` nor `operator->()` value observers as it does not model `optional<T>` like Expected does[2].

[2] In over three years of using Expected-like objects in my code, I have not found myself using them like filled|unfilled 'slots' in the same way as I write Optional-based code. This I believe is because if an Expected-returning function

4. Unchecked 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 such as monadic programming.

5. Unchecked 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. Unchecked is standard layout if both `T` and `E` are standard layout. Unchecked is 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.

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

8. Unchecked requires type `E` to be default constructible, and it defaults `E` to `std::error_code`.

   Expected permits `E` to not be default constructible, and does not default it to anything.

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

```
unchecked<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);
```

returns an error, that is something I usually want to deal with as soon as possible. One thus tends to not keep unwrapped returned values hanging around like one does with Optional. This is why Outcome does **not** model Optional, instead it models Variant. There was a surprisingly high amount of consensus during the Boost peer review that this is the right design decision, but to be fair, there is not much love of monadic programming on boost-dev, and monadic programming is the primary use case for an Optional-modelling Expected design.

```
}
```

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:

```
unchecked<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, it's superfluous to needs as the intention of the programmer is unambiguous.

2 Discussion of concerns on design differences in priority order

In case it is not yet clear, Outcome is designed to do one thing and one thing well: return value-or-error from functions.

Expected has been designed to do that as well, but also to act as a non-valued 'slot' same as Optional does, and for constructs of programming logic to be built on top of Expected in the same way as one builds on top of Optional. You could do this with Outcome too, but it would be quite tedious as all the implicit constructors would self-disable, forcing you to always write explicit construction.

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.

I'm going to stake out my position on this in order to explain why I think Expected's observers need to be all-narrow, not wide-narrow-narrow-narrow.

2.1.1 Logic errors are not recoverable errors!

Before the Outcome peer review, I hadn't really thought much as to the foolishness of throwing logic errors as non-fatal exception throws. Throws of `std::logic_error` after all permeate the C++ standard library, and I think back in the 1990s that was a reasonable attitude given the low capability of testing tooling available at the time.

But logic errors are not like normal errors! If your code realises that a logic error has occurred, then that means the programmer has written incorrect code. That in turn means that the programmer's intent for a valid program state has been lost, which in turn means that the program is now in an undefined state.

A better name for a program being in an undefined state is **memory corruption**. You now know **for a fact** that your program cannot safely continue. The only sensible thing to now do is to fatal exit the process.

Now, if throws of `std::logic_error` cannot be prevented from fatal exiting the program, I would have less of a problem with Expected's wide `.value()`. The program has irrecoverably lost a valid state, so we attempt to save out what we can, perhaps launching a new instance of ourselves to recover data, and die.

But of course in C++ until now, throws of `std::logic_error` are treated as ordinary, every day, recoverable errors. And that's stupid in an era where software is supposed to be reliable and predictable. It means that programmer laziness defaults to allowing known memory corruption to continue to accumulate.

2.1.2 Wide observers make analysis tooling useless

Allowing throws of `std::logic_error` to be recoverable made sense back when testing tooling might want to iterate an API call, making sure logic errors were thrown with the wrong inputs.

But testing tooling has come along vastly since the 1990s. We now have very high quality, free of cost, static analysers which can tell us at compile time if a piece of code is obviously incorrect. The opportunity costs of persisting with wide observers in new standard library additions are much higher than they were in the 1990s.

The problem with wide observers is that perhaps the programmer *intends* for the logic error exception to be thrown i.e. it is being used as control flow. The static analyser cannot then issue a warning because it cannot prove that the programmer is not intending for runtime exceptions to be thrown. One thus gets a non-obvious, hard to test corner case behaviour which the programmer probably did not fully think through. We thus get lower quality programming and C++ code.

But what about logic errors static analysis cannot prove? Again, tooling is vastly better now, even since the C++ 14 standard. We now have the undefined behaviour sanitiser which traps any occurrence of UB at runtime. Its overhead is sufficiently low that some applications ship production

release binaries with it enabled. So nowadays *you no longer need to be manually checking for logic errors*, the analysis tooling does it better than you can.

2.1.3 Wide observers raise development costs and slow down code

When you make it possible for a simple vocabulary type like Expected to throw exceptions, you force all code using that simple type to be written to be exception safe. Writing exception safe code is harder than writing code without exception throw paths running through it. **Testing** exception safe code properly is very considerably more expensive than testing code where exception throws cannot traverse it. And compilers must generate slower or more bloaty code when compiling code where exception throws can traverse.

Some wide observers add significant value to the programmer by saving time and boilerplate and visual complexity in the source code. But this is **never** the case for logic errors, these by definition should never be thrown, yet the programmer and the compiler must assume that they could be.

And that's plain stupid. Here we are burdening all C++ programmers with making useless their UB detection tooling and raising their development costs for literally **zero added value**.

Some may say 'ah but users of Expected ought to only use the narrow observers and you get everything you just described'. True. But if this is true, then what the hell is the point at all of a `.value()` which throws logic errors?

Just get rid of it. Make `.value()` narrow, just like `operator*()`, and eliminate `bad_expected_access<E>`. Save everybody time and hassle across the board. If user really, really wants a logic error exception thrown from misuse of the observers, they can implement that on their own by inheriting from Expected with almost no effort. So why hard code something with clear *negative added value* into the C++ standard?

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[3].

[3] My hope on this is that C++ Modules should eliminate this particular disadvantage for Outcome in the near future.

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, which 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 constructor design[4].

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.

My recommendation is that Expected's proposal text adds a footnote recommending that implementations 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 transport it with `E = std::exception_ptr` instead?

2.3 Standard layout propagation

It has not been historically felt important for standard layout-ness to be propagated intact into wrappers of things 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

[4]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. Smart constructors are expensive on compile time.

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;

  // 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[5] needed!

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. No doubt the standard library implementers will disagree with this idea in the strongest terms.

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-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 Outcome's Unchecked has not. However Unchecked'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[6]:

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

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

[6] 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.

Specifically the Boost peer review came up with these constraints for any of the implicit constructors to be enabled[7]:

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 Unchecked:

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.

This is the case because Unchecked (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 as **safe** 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*.

I suspect that Expected is too close to the LWG now to consider thinking again about the wisdom of standardising something so unfocused and ambiguous in use case, and nor probably would I actually want it so[8]. Sometimes it's better to ship imperfection sooner rather than perfection later as with `std::variant<...>`. We just need to get on with it, and paper over the cracks later.

Still, if WG21 finds this paper persuasive, it seems logical to me that **two** Expected-like types be standardised. One is focused exclusively on function returns as part of a *lightweight error handling system* (see *P0779R0 Proposing operator try*, also in this mailing). The other is focused on monadic and slot based programming like how you use Optional.

3 Conclusion and Recommendations for Expected

I don't want anything in this paper to prevent [P0323] Expected from landing in the C++ 20 standard. I need it sooner rather than later, and so do a ton of other people.

[7] 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.

[8] For example, I am blocked from submitting `https://ned14.github.io/afio/` as the proposed File I/O TS until Expected has moved to the LWG.

Equally, I think it would benefit greatly if 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, my most unlikely to be accepted recommendation is that Expected preserve standard layoutness of its types `T` and `E` with guaranteed layout, and a great boon for improving C++ interoperation with C-speaking programming languages. I know from std-proposals that this stands a snowball-in-hell's chance. Which is a shame.

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 if you accept the unfocused intended use cases for Expected. But if one were to standardise two types each focused on their specific remit, I think the C++ standard library would be the better for it.

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