std::valstat - metastate return type

Document Number:	**P2192R1**
Date	2020-08-10
Audience	SG18 LEWG Incubator
Author	Dusan B. Jovanovic (dbj@dbj.org)

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. -- C.A.R. Hoare

Table of Contents

Revision history

R1: Marketing blurb taken out. Focused and short proposal. metastate in the front.

R0: "Everything is numbered" style. A lot of story telling and self marketing. Too long.

1. Abstract

This is **not** yet another error handling solution, this is a proposal about lightweight but effective handling of information returned from functions.

This paper proposes applying an programming paradigm as a solution to a large proportion of deeply rooted, both C++ language and community problems[3]. Also, this is a library solution without a language change required. Materialized in standard C++ as one tiny template to be made part of the std lib.

2. Motivation

There are projects and individuals who want or need to deliver a standard C++ code, also faced with strict run-time requirements, very often orthogonal to the non-technical requirements. Primary motivation of this paper is to make those C++ users able to deliver quality code written in standard C++.

Non exhaustive list

1. must not use try / throw / catch
2. can not use <system_error>
3. no special return values
4. no "return arguments"
5. no special globals
6. increased interoperability

Honourable readership is well aware of the reasons behind and issues arising from this list ("C++ fragmentation", etc.). Thus, author will be so bold not to reiterate any of them in order to keep this paper simple and focused, but fully informative.

3. metastate

Paradigm behind this proposal shall be called "metastate". Metastate is combination of the states of value and status returned. Put in the code, metastate idea is really rather simple and easy to comprehend.

```
// legacy
// was there an error?
// is that really an error?
auto [value, error] = legacy_function ();

// metastate
// what has happened? full info please...
// status, not error
// use intrinsic types
auto [value, status] = modern_function ();
```

As meta-language is language of languages, **metastate** is "state of states". In the context of this document it is defined as an boolean AND combination of occupancy states of two instances: Value and State.

```
; kind of a state
occupancy ::=  "empty" | "has value"
```

```
; metastate is AND combination of two occupancies
metastate ::= occupancy(value) AND occupancy(status)
```

Combination of value *and* status occupancies is giving four possible metastates.

Meta State Name	Value occupancy	op	Status occupancy
Info	Has value	AND	Has value
OK	Has value	AND	Empty
Error	Empty	AND	Has value
Empty	Empty	AND	Empty

metastate names are just one word descriptions, they are not implying any kind of required behaviour from both callers or producers. Final metastate meaning is detached from these names. It is defined by adopters, their requirements and their coding standards.

Following is canonical capturing of metastates in standard C++. Here caller tests for all the four possible metastases returned.

```
// there is no 'special' return type
 auto [ value, status ] = metastate_enabled_function () ;

// meta states capturing idiom
// capturing all four possible metastates
// names are just in comments
// there is no 'metastate' type
 if (   value &&   status )  { /* info */ }
 if (   value && ! status )  { /* ok   */ }
 if ( ! value &&   status )  { /* error*/ }
 if ( ! value && ! status )  { /* empty*/ }
```

metastate also serves in coding clean algorithms for complex function call consuming; it is not just a solution for error handling. The aded benefits are immediate applicability and ability in addressing all of the requirements from the Motivation section.

This proposal is not describing an panacea. It's value lies in adaptability. Universal applicability of the metastate paradigm, would be greatly aided by placing one tiny template in the std lib. This proposal requires no changes in the core language.

4. valstat

The logic of metastate producing and consuming is completely application and/or context specific. Thus implementation must not dictate the usage beside supporting the paradigm (of metastate).

valstat is a name of the type, offering the greatest possible degree of freedom for metastate adopters. Implementation is simple, resilient, lightweight and feasible. Almost transparent.

Synopsis

std::valstat<T,S> as a template is an generic interface whose aliases and definitions must allow the easy metastates capturing by examining the state of occupancy of the 'value' and 'status' members.

```
#pragma once
// std lib header: <valstat>
namespace std
{
 template< typename T, typename S>
    struct [[nodiscard]] valstat
 {
    // both types must be able to
    // simply and resiliently
    // exhibit state of occupancy
    // "empty" or "has value"
        using value_type = T ;
        using status_type = S ;

    // metastate is captured by combining
    // state of occupancy of these two

        value_type   value;
        status_type  status;
 };
} // std
```

Valstat instances are vessels for carrying metastates

std::valstat will be assuring the metastate paradigm presence in the standard C++. It does not mandate its usage. It should exist as a recommended, not mandatory implementation. It should be in a separate header *<valstat>*, to allow for complete decoupling from any of the std lib types and headers.

Type requirements

Both value and status type must offer an method that reveals their occupancy state. Presumably in a simple manner. Readily available examples of that behaviour are std::optional type or std::empty family of methods.

4.1. my::valstat

To satisfy that requirement for almost unlimited set of types one common idiom is to use std::optional. In the valstat adopters template alias we declare both value and status as owned by *std::optional*.

```
// adopters namespace
namespace my {
// ready to operate an almost any type
// std::optional permitting
template<typename T, typename S>
using valstat = std::valstat<
         std::optional<T>,
         std::optional<S> >;
} // my
```

Now both producers and consumers have the generic readily applicable valstat template. Most of the time valstat consumers use a structured binding. An ad-hoc example:

```
    // naming implies the roles
    // creating ERROR metastate
     auto [ value, status ] = my::valstat< int, int >{ {}, my::errc };

     /*  using the optional  */
     if (! value && status ) {
         /* ERROR metastate captured
         empty value and non empty status
         status contains the actual errcode */
         cout << my::message( *status );
     }
```

Using fundamental types for both value and status halves of the metastate.

```
    // creating OK metastate
     auto [ value, status ] = my::valstat< int, int >{ 42, {} };

 // capturing the OK metastate
 if ( value && ! status ) {
    /* OK metastate : non empty value and empty status */
    cout << *value ;
 }
```

Above we also see the coding idioms shaping up when dealing with valstat types used for creating and carrying metastates.

5. Usage

There are many equally simple and convincing examples of metastate usage benefits. In order to keep this core proposal short we will observe just one, but illustrative use-case. Appendix A contains few more illustrative examples.

5.1. consumers point of view

valstat carries out (of the function) information to be utilized by callers capturing the metastate. How and why (or why not) is the metastate capturing code shaped, that completely depends on the context of the app, the API logic and many other factors dictated by adopters. Example bellow might be used by adopters operating on some database.

In this illustration, adopters use the metastate to pass back (to the caller) full information, obtained during the database field fetching operation.

There certainly could be, but here is no 'special' over-elaborated return type required. Actually in the code bellow there is no return type needed by the callers, at all. metastate is implied, there is no 'metastate' type.

```
// status type is adopters status code
template<typename T>
  my::valstat<T, my::stat >
     full_field_info
        (database::row row_, std::string_view field_name )
           noexcept ;
// obtaining full information after
// the attempted integer value retrieval
// field name happens to be: "uid"
auto [ value, status ] = full_field_info<int>( db_row, "uid" ) ;
```

Primary objective is enabling callers comprehension of a full information passed out of the function. Of course satisfying the core requirements from the motivation section. In this scenario caller is capturing all four metastates.

```
if (   value &&   status )  { /* metastate captured: info */
   std::cout << "\nSignificant value found: " << *value ;
   std::cout << "\nStatus is: " << my::status_message(*status) ;
  }

if (   value && ! status )  { /* metastate captured: ok   */
   std::cout << "\nRetrieved value: " << *value ;
  }

if ( ! value &&   status )  { /* metastate captured: error */
   // in this scenario status contains an error message
   std::cout << "\nRead error: " <<my::status_message(*status) ;
  }

if ( ! value && ! status )  { /* metastate captured: empty */
   std::cout << "\nField is empty." ;
  }
```

Let us emphasize: Not all possible metastates need to be captured by the caller each and every time. It entirely depends on the API "contract", on the logic of the calling site, on application requirements and such.

5.2. producers point of view

Requirements permitting, API implementers are free to choose if they will use and return them all, one,two or three metastates. Context is adopters API namespace, using some hypothetical database API, not throwing exceptions.

```
template<typename T>
   my::valstat<T, my::stat >
      full_field_info
```

```
          (database::row row_, std::string_view field_name )
             noexcept
{
   database::field_descriptor field = row_.fetch( field_name ) ;

    if ( field.in_error() )
    /* return ERROR metastate halves */
      return { {}, field.last_error_code() };

    // empty field is not error
    // database theory considers as empty, a field having a null value
    if ( field.is_empty() )
    /* return EMPTY metastate halves */
      return { {}, {} };

   T field_value{} ;
   if ( false == field.data( field_value ) )
   /* return ERROR metastate halves */
      return { {},  my::stat::type_cast_failed  };

 /* API contract requires signalling if 'special' value is found */
  if ( special_value( field_value ) )
  /* return INFO metastate halves */
   return { field_value, my::stat::special_value };

   return { field_value, {} }; /* OK metastate halves */
}
```

Obviously one does not even need to use std::valstat to employ the benefits of the metastate paradigm. It entirely depends on the adopters requirements. Using thread safe abstractions, or asynchronous processing is also not stopping the adopters to return the metastates from their API's.

Basically function signalling the metastate is simply returning two fields structure. With all the advantages and disadvantages imposed by the core language rules

6. Summary

Fundamentally, the burden of proof is on the proposers. — B. Stroustrup, [11]

Developing standard C++ code using standard library, but in restricted run-time environments, is what one might call a "situation"[3][4][11]. Author is certain honourable readership knows quite well why is the situation unresolved, and which situation is that. Therefore there is no need for yet another tractate, in this proposal.

Authors primary target is to achieve widespread adoption of the metastate paradigm. metastate is not solving just the "error-signalling problem"[11]. It is an paradigm, instrumental in solving the often hard and orthogonal set of run-time requirements described in the motivation section.

"A paradigm is a standard, perspective, or set of ideas. A paradigm is a way of looking at something ... When you change paradigms, you're changing how you think about something..." vocabulary.com

metastate aims high. And the scope of metastate is rather wide. But this proposal is not C++ language extension, or an "panacea" , "silver bullet", "awesome paradigm" and some such "revolutionary thing".

metastate is just an immediate and effective way of building bridges over one fragmented part of the vast C++ territory. While imposing extremely little on adopters implementations and leaving the non-adopters to "proceed as before".

Obstacles to metastate paradigm adoption are far from just technical. But here is at least an eminently usable attempt to chart the way out.

7. References

- [0] B. Stroustrup (2018) **P0976: The Evils of Paradigms Or Beware of one-solution-fits-all thinking**, https://www.stroustrup.com/P0976-the-evils-of-paradigms.pdf
- [1] Ben Craig, Ben Saks, **Leaving no room for a lower-level language: A C++ Subset**, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1105r1.html#p0709
- [2] Lawrence Crowl, Chris Mysen, **A Class for Status and Optional Value**, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0262r1.html
- [3] Herb Sutter,**Zero-overhead deterministic exceptions**, https://wg21.link/P0709
- [4] Douglas, Niall, **SG14 status_code and standard error object for P0709 Zero-overhead deterministic exceptions**, https://wg21.link/P1028
    - Douglas Niall, **Zero overhead deterministic failure – A unified mechanism for C and C++**, https://wg21.link/P1095
- [5] Gustedt Jens, **Out-of-band bit for exceptional return and errno replacement**, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2361.pdf
    - Douglas Niall / Gustedt Jens, **Function failure annotation** , http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2429.pdf
- [6] Craig Ben, **Error size benchmarking: Redux** , http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1640r1.html
- [7] Vicente J. Botet Escribá, JF Bastien, **Utility class to represent expected object**, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0323r3.pdf
- [8] Shoop Kirk, **Cancellation is not an Error**, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1677r0.pdf
- [9] Wikipedia **Empty String**, https://en.wikipedia.org/wiki/Empty_string
- [10] "Your Dictionary" **Definition of empty**, https://www.yourdictionary.com/empty
- [11] Bjarne Stroustrup **P1947 C++ exceptions and alternatives**, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1947r0.pdf

8. Appendix A

Value of this paradigm is best understood by seeing the code using it. The more the merrier. Here are a few more simple examples illustrating the metastate applicability. All following the initial set of requirements.

metastate as a value carrier

An elegant solution to the "index out of bounds" problem. No exceptions, and no `exit()` in release builds. Using my::valstat as already defined above.

```
// inside some sequence like container
// see the my::valstat above.
// my::errc is std::errc, minus <system_error> included.
 my::valstat< T , my::errc >
     operator [] ( size_t idx_ ) noexcept
    {
        if ( ! ( idx_ < size_ ) )
        /* ERROR metastate */
        return { {}, my::errc::invalid_argument };
        /* OK metastate + value carrier */
        return { data_[idx_] , {} };
    }
```

Metastate does carry the value. But only if metastates are OK or INFO. That pattern alone resolves few difficult and common API design issues.

Uniform solutions for simpler usage

Due to decades of legacy code development and debugging, inevitably its usage and complexity has become a problem. True even for C++ std lib. Somewhat trivial example: To find a char inside a std::string one can use std::find. To find a string inside a std::string, one has to use std::string::find. metastate based solution out of this might be:

```
// value is a position in a sequence being searched
template<class C, class T>
   constexpr  my::valstat< size_t, my::errc >
      my::find(C const & container_ , const T& value) noexcept;
```

using the overloads of above one can imagine a simple variety of implementations but all serving the same find consuming logic at the client side. Using the vector first.

```
    int n1 = 3 ;
    std::vector<int> v{0, 1, 2, 3, 4};
   auto [position,status] = my::find(v, n1);
```

Or using the string

```
    char n1 = '3' ;
     std::string s{"01234"};
    // using std::string overloads
    auto [position,status] = my::find(s, n1);
```

Find a string inside a string

```
    string n2 = "23" ;
    auto [position,status] = my::find(s, n2);
```

metastate presence brings uniformity in algorithms using that API. That in turn makes applications simpler and more resilient. The same find logic is applicable to all containers and sequences alike. (std::string::npos is not required).

```
// always the same find consuming logic
// based on metastate capturing
if ( position )
   {
      // found it
      // position is guaranteed to be inside boundaries
      my::process_the_find( v[*position] ) ;
   } else {
      // not found, status contains the reason
      my::process_the_status( *status ) ;
   }
```

Above is certainly doable without a metastate, but it will require yet another design and some new set of abstractions (remember std::to_chars). And that lowers the interoperability. And raises the complexity.

Decoupling from the legacy

One can imagine using the metastate paradigm for developing simple but ubiquitous layer in front of the legacy API's.

```
// notice here we even do not even use my::valstat
inline std::valstat<FILE*, errno_t >
  modern_fopen(const char* name_, const char* mode_)
noexcept
{
    FILE* fp_{};
    int ec_ = fopen_s(&fp_, name_, mode_);

    if (NULL == fp_)
        // returning the ERROR metastate
        return { {} ,  ec_ };

    // returning the OK metastate
```

```
        return { fp_, {} }; // OK metastate
    }
```

And the metastate usage.

```
FILE * filep{} ;

if (auto [ filep , errc ] = modern_fopen( "non_existent_file" , "w+" ); filep )
{
   fprintf( filep, "OK" ) ;
}  else {
   // ad-hoc code
   auto message = strerror( ( errc ? strerror (errc) : "no status") ) ;
}
```

Above decouples from decades of "special return values" ,errno globals abd POSIX "hash defines". One can imagine the whole layer of metastate enabled functions, in front of the CRT legacy. Again, all doable without std::valstat and following yet another paradigm. Proposal is simple: make std::valstat and metastate, "an official way of doing it".