std::valstat - Returns Handling

Document Number:	**P2192R3**
Date	2020-10-13
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

- 1. Abstract
- 2. Motivation
  - 2.1. Standard returns handling is missing
  - 2.2. Requirement categories, across domains, we don't talk about
- 3. metastate
  - 3.1. Field
    - 3.1.1. Occupancy states
  - 3.2. Definition
  - 3.3. What this has to do with the std lib?
- 4. valstat
  - 4.1. Synopsis
  - 4.2. Type requirements
  - 4.3. my::valstat
  - 4.4. bare-bones valstat
- 5. Usage
  - 5.1. Users point of view
  - 5.2. the API point of view
- 6. Conclusions
- 7. References
- 8. Appendix A
  - 8.1. metastate as a solution for known and difficult problems
  - 8.2. Fully functional valstat type, minus std::valstat.
- 9. Appendix B: Requirements Common Across Domains
  - 9.1. Run-Time
  - 9.2. Interoperability
  - 9.3. Energy
- 10. Appendix C: metastate BNF definition

Revision history

R3: Two stages returns handling clarification. Better examples.

R2: More elaborate motivation. Better metastate section. Cleaner Appendix examples. Title changed from "std::valstat - transparent return type" to "std::valstat -Transparent Returns Handling".

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 a proposal about logical, feasible, lightweight and effective handling of information returned from functions, based on an paradigm shift.

This paper proposes an Architecture and Implementation of solutions, to some of deeply rooted ISO C++ issues [3][15]. Implemented for and by C++ adopters, this would be a tiny std library citizen without any language change required.

2. Motivation

2.1. Standard returns handling is missing

As of today, in the std lib, there are few (or more than few) **error** handling paradigms, idioms and return types. Accumulated through decades, from ancient to old. None of them dealing with general function returns in an satisfactory manner. Together they have inevitably contributed to a rising technical debt present inside C++ std lib.

But none of them serves the ISO C++ users, returns handling requirements. Think famous, HTTP codes handling algorithms, as an well known example. Lack of a common and ubiquitous non trivial returns handling recommendation or solutions is raising the level of complexity for all levels of application architectures developed with standard C++.

What is "Returns handling"? Returns handling is the next step in evolution of error handling. It is the true picture of a real life code. Returns handling is already deployed in API consuming algorithms, required to deal with wider scope of issues, opposed to a simple "error or no error", returns processing logic.

2.2. Requirement categories, across domains, we don't talk about

Each and every system programming effort, meets these requirements, yet there is no advice how to approach them. They are external to C++ as a programming language, of course. But directly shaping the logic and style of coding in very large portion of today's projects. Motivation is to develop an paradigm to aid solving these three core requirements categories, on the level of code:

1. Run-Time
2. Interoperability
3. Energy

Please see Appendix B: Requirements Across Domains for a bit more detailed but quick overview.

3. metastate

"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

When developing or calling (ISO C++) API's returning `std::optional`, users are already applying the two stages logic:

- stage one -- Is something returned?
- stage two -- Can I use it?

Conceptually metastate paradigm belongs to the same category of the "two stage" returns processing paradigms:

- Stage one: use metastates to determine the outcome
  - not using the type system
- Stage two: use the content returned
  - using the type system

That is in a way same as `std::optional` but without it, so it can be applied across many domains.

3.1. Field

Let us postulate the existence of a function:

```
// returns true if field is empty
      bool is_empty( Field f ) ;
```

That function is letting us know what is the "state of occupancy" of an "field". Is it empty or not. But what is this "Field thing" ?

C++ "field" is analogous to the database field. The name "field" is the name for an "single piece of information", in database theory also known as "field". "field" in the database is what in C++ is: "*a particular piece of data encapsulated within a class or object*" [ref here].

3.1.1. Occupancy states

Field can be in two "occupancy states" (authors term) . We will call them : "empty" and "occupied". It is a well known and adopted fact: database field always exist, but it can be empty.

`std::optional` is a well known implementation of a C++ field, sometimes known as "container of one". It's instance is an object **potentially** holding only one variable defined in it. It might be tested if it is empty; "not holding a value". Or, occupied or "holding a value".

3.2. Definition

> Metastate is the foundation to the family of returns handling idioms

As meta-language is language of languages, **metastate** is "state of states". metastate is an boolean AND combination of occupancy states of two fields. Named: Value and Status. A bit more formal definition is in

Appendix C.

Combination of value *and* status occupancies is giving four possible metastates. We will label them for further use, for stage one of returns processing.

Meta State Label	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 labels are just that: labels. Just indicating the behaviour.

That is it. That is the core, adopters can use when solving their requirements listed in the Motivation section. Put in some simple C++ code, metastate idea is really rather simple and easy to comprehend.

> In adopting the metastate paradigm we do not immediately inspect returned values, we inspect the relationship of two fields returned first.

```
// there is no special type returned
// both returns are fields
// metastates are captured AND-ing their occupancy states
// metastates are used in stage one of returns processing
auto [value, status] = metastate_enabled_function ();
```

Like in the most other languages, in C++ there is no need for existence of some dedicated metastate type. We only care about the relationship between two fields states of occupancy.

Metastate capturing is the stage one of return handling logic. Metastate capturing is the act of decoding the relationship between occupancy states of its two fields. Following is canonical capturing of the four possible metastates, in some pseudo code :

```
// pseudo code
// two fields are input into the idiom of
// capturing all four possible metastates
// This is stage one of return processing
// In stage one types or values of the
// content returned are not used
// they are irrelevant in stage one
//
  if (   is_empty( value ) &&   is_empty( status )  { /* info */ }
  if (   is_empty( value ) && ! is_empty( status )  { /* ok   */ }
  if ( ! is_empty( value ) &&   is_empty( status )  { /* error*/ }
  if ( ! is_empty( value ) && ! is_empty( status )  { /* empty*/ }
```

That is not C++ but it can be. As well as it can be many other languages: C, JavaScript, Python, Java etc. In standard C++ reality we do not need `is_empty()` function from above. We can use `std::optional` as a readily available field type.

```
// C++ code
// fields are std::optional instances
// this is still stage one
// we do not need type or value of a content
// just two fields occupancy states
auto [value, status] = metastate_enabled_function ();

  if (   value &&   status )  { /* info */ }
  if (   value && ! status )  { /* ok   */ }
  if ( ! value &&   status )  { /* error*/ }
  if ( ! value && ! status )  { /* empty*/ }
```

As return is neatly divided in two stages, metastate logic serves well for arriving to cleaner idioms for complex returns handling. The aded benefits are immediate applicability and ability in addressing the requirements from the Motivation section: strict runtime requirements solving across domains. Also arriving to a more energy friendly C++ coded executables.[6]

3.3. What this has to do with the std lib?

This proposal's value also lies in it's deliberate simplicity, aiding in solving the strict operational requirements and interoperability.

Modern software architectures are clusters of interoperating but separated components. Thorny not-a-detail is solving universally applicable returns handling, across language and system barriers. And this is where metastate as a paradigm might help. (Think JSON with metastate inside)

Universal adoption of the metastate paradigm, would be greatly aided by placing one tiny template in the standard C++ std lib. This proposal requires no changes in the core language. Truth to be told there is not a single type proposed. Just one template.

4. valstat

> valstat is an transparent metastates carrier

In order to achieve the required wide scope of the metastate paradigm, implementation has to be simple. Metastate actual programming language shape has to be completely problem domain context free. Put simply the C++ std lib implementation must not influence or dictate the usage beside supporting the paradigm.

"valstat" is a name of the C++ template, offering the greatest possible degree of freedom for metastate C++ adopters. Implementation is simplest possible, resilient, lightweight and feasible. Almost transparent. The only requirement is to give callers, the opportunity to capture the four metastates, returned by some "meta state enabled" API.

4.1. Synopsis

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

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

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

        value_field_type   value;
        status_field_type  status;
 };
} // std
```

std::valstat will be assuring the metastate presence in the realm of ISO C++, as an recommendation. It will not mandate its usage in any way. It should be in a separate header <*valstat*>, to allow for complete decoupling from any of the std lib types and headers. Let us repeat: `std::valstat` is a recommendation. Metastate paradigm can be implemented in many ways in C++; and many other languages.

4.2. Type requirements

Both value and status field types, should offer an simple mechanism that reveals their occupancy state. Readily available example of that behaviour is std::optional.

In specific contexts a native pointer or any other type can server the same purpose, as it will be explained shortly. What is the meaning of "empty" for a particular C++ type, and what is not, depends on the domain context. Please see an example in the appendix

4.3. my::valstat

Is an `std::valstat` "natural" ISO C++ variation we will actually use in most examples in this proposal. We will solve the occupancy requirement imposed on valstat fields by simply using `std::optional`. No thousands of lines of C++ required for some special type. No need to be concerned about the implementation complexity[13].

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

```
        std::optional<S> >;
} // my
```

In standard C++ view of `my::valstat` it is not wrong to relax a metastate definition, as an "AND combination" of two `std::optional`'s.

Now both API coders and API callers (of the `my` namespace) have the universal readily applicable valstat, as an simple template alias. Most of the time valstat C++ users will use a structured binding. Let's see some ad-hoc C++ examples of `my::valstat` direct usage, no functions involved yet:

```
    // OK metastate created
    // both fields are std::option<int> instances
     auto [ value, status ] = my::valstat< int, int >{ 42, {} };

 // stage one: compare the fields occupancy
 // OK metastate captured
 // there is a value but no status returned
     if ( value && ! status ) {
       /* stage two:
          now use the status value and type, taken from a field instance */
         std::cout << "OK metastate captured, value is: " << *value ;
     }
```

If required, the other three metastates wil be created like so:

```
    // both fields are std::option<int> instances
     auto [ value, status ] = my::valstat< int, int >{ 42, 42 }; // INFO
     auto [ value, status ] = my::valstat< int, int >{ {}, {} }; // EMPTY
     auto [ value, status ] = my::valstat< int, int >{ {}, 42 }; // ERROR
```

What metastates are produced and their exact logic, completely depends on the adopters domain.

4.4. bare-bones valstat

After all this postulating, field theory and such, it might come as a surprise, in some circumstances it is quite ok and enough to be using fundamental types for both value and status fields. In a way we apply the "field" concept by using just fundamental types.

Let us consider some very strict embedded system, run-time environment.

```
    // valstat but not as we know it
    // note: this struct might be defined in some C code too
    struct valstat_int_int final {
       int value;
       int status;
    };

    // both value and status fields in here are just integers
     auto [ value, status ] = valstat_int_int{ 42, {} }; // OK
```

```
// Stage one
// OK metastate captured
// (42 && !0 ) yields true
if ( value && ! status ) { uplink( value ) ; }

// other three metastates, but only if required
    auto [ value, status ] = valstat_int_int{ 42, 42 }; // INFO
    auto [ value, status ] = valstat_int_int{ {}, {} }; // EMPTY
    auto [ value, status ] = valstat_int_int{ {}, 42 }; // ERROR
```

That is still metastate paradigm in action. It is only, in some context valstat field types can be two simple integers. We have declared in this context only, we will think of int as empty if it is zero.

```
// stage one
// in some specific narrow context integer is "empty" if it is zero
bool is_empty( int val_ ) {   return ! val_ ; }
```

Above is rather important metastate ability for projects mentioned in the motivation section, as that solution is not using std lib and working under extremely tight conditions. The already mentioned example in the appendix, shows something different but similar.

5. Usage

It is admittedly hard to immediately see the connection between metastate and valstat, and the somewhat bold promises about wide spectrum of benefits, presented in the motivation section.

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

5.1. Users point of view

Recap. Returns handling: valstat instance carries (out of the functions) 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 project, the API logic and many other requirements dictated by adopters architects and developers.

Example bellow is used by metastate adopters operating on some database. In this illustration, adopters use the metastate to pass back (to the caller) full information, obtained after the database field fetching operation. Again, there is no 'special' over-elaborated return type required. That is a good thing. Metastate is a paradigm, there is no complex C++ type, just idioms of two stage return handling.

```
// declaration of a metastate emitting function
// aka valstat returning function
template<typename T>
// we use the `my::valstat` type from above
// `my::stat` is 'code' from some code/message mechanism.
  my::valstat<T, my::stat >
```

```
  full_field_info
  (database::row /*row_*/ , std::string_view /* field_name */ )
// valstat imposes no exception throwing requirements
  noexcept ;
```

Primary objective is enabling callers comprehension of a full information passed out of the function. Full returns, not just error handling. Satisfying the core requirements from the motivation section.

```
// full return handling after
// the attempted field content retrieval
auto [ value, status ] = full_field_info<int>( db_row, field_name ) ;
```

When designing a solution, adopters have decided they will utilise all four metastates. Calling code is capturing all four metastates.

```
// stage one
// metastate captured: info
if (   value &&   status )  {
   // stage two
   std::cout << "\nSpecial value found: " << *value ;
   // *status type is my::stat
   std::cout << "\nStatus is: " << my::status_message(*status) ;
  }

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

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

// metastate captured: empty
if ( ! value && ! status )  {
   // empty feild is not an error
   std::cout << "\nField is empty." ;
  }
```

Please do note, using the same paradigm it is almost trivial to imagine that same calling algorithm in e.g. JavaScript inside some node.js, calling the module written in C++ returning valstat object that JavaScript will understand.

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

5.2. the API point of view

Requirements permitting, API implementers are free to choose if they will use and return them all, one,two or three metastates. In this scenario they return one from the possible four.

```
// implementation in the API namespace
template<typename T>
my::valstat<T, my::stat >
full_field_info
(database::row row_, std::string_view field_name )
// run time requirements do not allow
// throwing exceptions
            noexcept
{
   // sanity check
   if ( field_name.size() < 1)
    // return ERROR metastate
      return { {}, my::stat::name_empty };

   // using some hypothetical database API
   // row is made of fields
   database::field_descriptor field = row_.fetch( field_name ) ;

    // error can be anything not
    // just database related
    if ( field.in_error() )
    // return ERROR metastate
    // status is some my::stat code
      return { {}, my::stat::db_api_error( field.error() ) };

    // design decision
    // empty field is not an error
    // return an EMPTY metastate
    if ( field.is_empty() )
      return { {}, {} };

   // db type will have to be cast into the type T
   // we assume T properly handles value semantics
   T field_value{} ;
   // try getting the value from a database field
   // and casting it into T
   if ( false == field.data( field_value ) )
   // failed, return ERROR metastate
    // status is my::stat code
      return { {},  my::stat::type_cast_failed( field.error() ) };

 // design decision, solving business requirement
 // API contract requires signalling if 'special' value is found
  if ( special_value( field_value ) )
  // design decision
  // return INFO metastate
   return { field_value, my::stat::special_value };
```

```
// value is obtained and ready
// T move-ability is used here
// status is empty
// OK metastate
   return { field_value, {} };
}
```

Basically function returning the metastate is simply returning two fields structure. With all the advantages and disadvantages imposed by the core language rules. Any kind of home grown but functional valstat will work in there too. As long as callers can capture the metastates in two stages and by using its two fields.

Using thread safe abstractions, or asynchronous processing is also not stopping the adopters to return the metastates from their API's.

6. Conclusions

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

Hopefully proving the evolution of error code handling into returns handling does not need much convincing. There are many real returns handling situations where the metastate paradigm can be successfully used. As an common call returns handling paradigm, metastate requires to be ubiquitously adopted to become truly an instrumental to widespread interoperability. From micro to macro levels. From inside the code to inter component calls. From inside an project to inter team situations.

"metastate" is multilingual in nature. Thus adopters from any imperative language are free to implement it in any way they wish too. The key requirement is: interoperability.

Developing standard C++ code using standard library, but in restricted run-time environments, is what one might call a "difficult situation"[3][4][11]. Author is certain readership knows quite well why is that situation considered unresolved in the domain of ISO C++. There is no need for yet another tractate, in the form of proposal to try and explain the background.

Authors primary aim is to achieve widespread adoption of this paradigm. As shown metastate is more than just solving 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.

metastate aims high. And it's proposed scope of is rather wide. But it is a humble paradigm. It is just an simple and effective way of building bridges over one deeply 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 immediately 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]Vicente Botet, JF Bastien, **std::expected** http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0323r8.html
- [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
- [12] A Conversation with Anders Hejlsberg, Part II **The Trouble with Checked Exceptions**, https://www.artima.com/intv/handcuffs.html
- [13] Niall Douglass **Concerns about expected<T, E> from the Boost.Outcome peer review**, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0762r0.pdf
- [14] Library Evolution Working Group **Summary of SG14 discussion on <system_error>**, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0824r1.html
- [15] Joel Spolsky, **Joel On Software -- 13: Exceptions**, https://www.joelonsoftware.com/2003/10/13/13/

8. Appendix A

To me, one of the hallmarks of good programming is that the code looks so simple that you are tempted to dismiss the skill of the author. Writing good clean understandable code is hard work whatever language you are using -- Francis Glassborow

Value of the programming 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.

8.1. metastate as a solution for known and difficult problems

An perhaps very elegant solution to the "index out of bounds" problem. Using my::valstat as already defined above.

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

That pattern alone resolves few difficult and well known design issues.

```
auto [ value, status ] = my_vector[42] ;

// first stage: check the metastates
if ( value  )  { /* we are here just if metastate is OK     */ }
if ( status )  { /* we are here just if metastate is ERROR  */ }
```

No exceptions, no `assert()` and no `exit()` in release builds. `at()` will of course be exactly the same.

As an very real example of the std lib situation in the presence of "no exception" requirement we might show the current MS STL solution to the above:

```
// MS STL <vector> approx line#2590 , circa 2020 Oct
    _NODISCARD reference at(size_type _Off) {
        if (size() <= _Off) {
            _Xran();
        }
        return (*this)[_Off];
    }
```

If index exceeds the size, exception is thrown from the function `_Xran()`. But there are "Requirements Across Domain" issues, we have already described in the motivation section and detailed in Appendix B: Requirements Common Across Domains. To solve those, MS STL authors have resorted to the OS specific solution. In the presence of `_HAS_EXCEPTIONS == 0`, `_Xran()` ultimately raises the structured exception. The whole MS STL is, in that scenario, in the non-standard shape.

Deploying the metastate paradigm above might not result in somewhat controversial MS STL design.

8.2. Fully functional valstat type, minus std::valstat.

Let us assume we need to write a layer of safe proxies to some C run time (CRT) functions. And yes, in the same time, we are under strict requirements already mentioned too many times in this paper. We shall first start by declarin an **"valstat like"** template for this specific context.

```
namespace crt {
template<typename T>
struct [[nodiscard]]  errno_valstat final {

using value_field_type  = T ;
// We want to decouple users
// from the existence of the `errno`.
using status_field_type = std::errc ;

   // value based semantics
    value_field_type   value;
    status_field_type  status;
} ;
} // crt
```

In that narrow context, that valstat variant might be used in a myriad of API's, internally facing the CRT, or CRT like API's. All returning the metastates, carried by the definitions of that one template: `errno_valstat<T>` . Examples:

```
// assume some CRT proxy functions
//
crt::errno_valstat<const char *> safe_read_line ( FILE * ) noexcept ;

crt::errno_valstat<double> safe_sqrt( double * ) noexcept ;

crt::errno_valstat<const char *> safe_strdup( const char * ) noexcept ;
```

And here are some possible legal `crt::errno_valstat<T>` usages from insides of those functions:

```
// EMPTY metastate result -- from safe_read_line ( FILE * ) ;
crt::errno_valstat<const char *> { nullptr , std::errc{} } ;

// INFO metastate  result -- from safe_read_line (FILE *)
crt::errno_valstat<const char *> { & value_ , std::errc::is_a_directory } ;

// OK metastate  result -- from safe_sqrt( double ) API
// make sure that is not address to local instance
crt::errno_valstat<double *> { & value_ , std::errc{} } ;

// ERROR metastate  result -- from safe_strdup( const char *)
crt::errno_valstat<const char *> { nullptr , std::errc::invalid_argument } ;
```

The caller using any of the above imagined crt proxy functions, will follow the same metastate capturing idiom in the stage one of returns processing.

```
// stage one: use metastates only
// returned by calling any of the three above
// value is a native pointer , status is std::errc
auto [value, status] = any_of_the_three_above ();

  if (   value &&   status )  { /* info */ }
  if (   value && ! status )  { /* ok   */ }
  if ( ! value &&   status )  { /* error*/ }
  if ( ! value && ! status )  { /* empty*/ }
```

That is an example to show and explain the suitability of metastate in various non standard contexts. Let's see how would `modern_fopen` might be actually implemented.

```
//
inline crt::errno_valstat<FILE*>
  modern_fopen(const char* name_, const char* mode_)
noexcept
{
    // the ERROR metastate is returned on bad arguments
    if (NULL == name_)
        return { {} ,  errno_to_errc(ec_) };

    if (NULL == mode_)
        return { {} ,  errno_to_errc(ec_) };

    FILE* fp_{};
    int ec_ = fopen_s(&fp_, name_, mode_);

    // file is not opened for any of many reasons
    // the ERROR metastate
    if (NULL == fp_)
        return { {} ,  errno_to_errc(ec_) };

    // return file pointer as a value field
    // and std::errc{} as a status field
    // OK metastate
    return { fp_, {} };
}
```

Very simple but safe, fully metastate enabled. The usage:

```
// C++17 if() syntax
if (auto [ filep , errc ]
    = modern_fopen( "non_existent_file" , "w+" );
   // return processing stage one
   // check if value field is 'empty'
    filep )
```

```
{
   // stage two:
   fprintf( filep, "non_existent_file opened" ) ;
}
else {
   // return processing stage one
   // has determined "value" field is empty
   // check the "status" field occupancy
   if ( errc )  {
   // stage two begins here
   // status is not empty, we can use the content of the status field
   auto message = errc_to_string (errc);
   }
}
```

Above decouples from decades of "special return values" ,`errno` globals and POSIX "hash defines" lurking inside any C++ code base today. True that is done before. But, in addition to that, metastate enabled proxy functions, in front of the CRT legacy, are delivering resilient, uniform and interoperable solution, blessed by std lib too. Available immediately.

9. Appendix B: Requirements Common Across Domains

9.1. Run-Time

Perhaps the key reasons for appearance of C++ dialects, are to be found in the std lib perceived inability to be used for components required to operate in the environments with limited resources available. That essentially means developing using the C++ core language but without the std lib. [1]

One motivation of this paper is to try and offer an "over arching", but simple enough, returns handling paradigm applicable across the C++ landscape. Including across a growing number of C++ dialects, fragmenting away the industry and markets relying on existence of the standard C++.

Minimal list of requirements

(for ISO C++ projects, producing components for restricted environments)

1. can not use try / throw / catch[6][15]
2. can not use <system_error>[14]
3. do not use <iostreams>

For details, authoritative references are provided. Author will be so bold not to delve into the reasons and background of this list, in order to keep this paper simple and focused. Gaming, embedded systems, high performance, mission critical computing, are just the tip of the iceberg.

9.2. Interoperability

Each traditional solution to those strict run-time requirements is one nail in the coffin of interoperability. In danger of sounding like offering an panacea, author will also draw the attention to the malleability of the metastate paradigm to be implemented with wide variety of languages used in developing components of an modern distributed system.

Usability of an API is measured on all levels: from the code level, to the distributed system level. In order to design an API to be **feasibly** usable it has to be interoperable. That leads to three core requirements of

Interoperable API core requirements (to start with)

1. no "error code" as return value
   - Complexity arises from "special" error codes multiplied with different types multiplied with different context
   - In turn increasing the learning curve for every function in every API
   - How to decode the error code, becomes a prominent issue
     - Think Windows: `NTSTATUS`, `HRESULT`, `GetCode()`, `errno`
2. no "return arguments" aka "reference arguments" in C++.
   - language specific mutable argument solutions are definitely not interoperable.
3. no special globals
   - Think errno legacy
   - pure functions can not use globals

Some of the designed-in, simplicity in this paper is an result of deliberate attempt to increase the interoperability (also with other run-time environments and languages).

It is important to understand there are inter domain interoperability requirements, not just using standard C++. Examples: WASM, Node.JS, Android and such.

9.3. Energy

This is operational environment requirement. Operational environment can be satellite in an orbit, or a data center.

Side info: it is not just hardware energy consumption, it is also energy required to cool that data-center hardware down.

Solving data centers energy spending has become an imperative. Most of the server side software is written in C/C++ . Pressure is on, to design and develop using standard C++ but also with energy consumption as an primary requirement.

This one is not a "simple" requirement. Somewhat paradoxically this category of requirements requires less and less code and more and more performance in the same time. Smaller executables means less energy spent on that executable running and less energy for cooling the CPU running it.

10. Appendix C: metastate BNF definition

```
; occupancy states
empty     ::= true | false
has_value ::= true | false
occupancy_state ::== empty | has_value

; field is made of
; occupancy state and value
field ::=  occupancy_state AND value
```

```
; 'value' and 'status' are fields
value  ::= field
status ::= field

; metastate is AND combination of
; two fields occupancy states
; combination of occupancy_states of two fields
metastate ::= is_empty(value) AND is_empty(status)

; valstat is a record made of two fields
valsat ::= value AND status
```

Backus–Naur form or Backus normal form (BNF) is a metasyntax notation for context-free grammars.