

metastate - Returns Handling Paradigm

| | |
|------------------|---|
| Document Number: | P2192R2 |
| Date | 2020-09-14 |
| 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. Run-Time](#)
 - [2.3. Interoperability](#)
 - [2.4. Energy](#)
- [3. metastate](#)
 - [3.1. Field](#)
 - [3.1.1. Occupancy states](#)
 - [3.1.2. Definition](#)
- [4. valstat](#)
 - [4.1. my::valstat](#)
 - [4.2. valstat but not as we know it](#)
- [5. Usage](#)
 - [5.1. Callers 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, not using std::optional and not using std::valstat.](#)
 - [8.3. Decoupling from the legacy](#)

Revision history

R2: More elaborate motivation. Better metastate section. Cleaner Appendix examples.

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 paradigm shift. An utility to a solution to a large proportion of deeply rooted issues [3][15]. Implemented for C++ adopters, this would be a tiny std library citizen without a 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.

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

There is none in the std lib of such pervasive paradigm or mechanism. Dotted on the vast C++ std lib landscape there are some attempts to the returns handling solutions.

But none of them serves the customers returns handling requirements. Think 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++.

Motivation is to aid architectures, solving three core requirements categories

Requirement categories, across business domains

1. Run-Time
2. Interoperability
3. Energy

2.2. Run-Time

Perhaps the key reason for appearance of C++ dialects, are to be found in std lib perceived inability to address the strict run-time requirements. That essentially means developing using the C++ core language but [without the std lib](#).

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 fast growing C++ dialects, fragmenting away the industry and markets relying on existence of the standard C++. Here is one

Minimal list of requirements

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

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 computing, are just the tip of the iceberg to which C++ ship might be heading. Perhaps pathetic but true statement.

2.3. Interoperability

Each solution to those strict run-time requirements is adding 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 inter component 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

1. no "error code" as return value
 1. Complexity is in "special" error codes multiplied with different types multiplied with different context
 2. In turn increasing the learning curve for every function in every API
 3. How to decode the error code, becomes a prominent issue
2. no "return arguments" aka "reference arguments" in C++.
 1. type specific, mutable argument solutions are definitely not interoperable.
3. no special globals
 1. Think errno legacy
 2. 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 the inter domain interoperability requirements, not just using standard C++. Examples: [WASM](#), [NodeJS](#), [Android](#) and such.

2.4. Energy

Solving data centers large energy consumption 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. Smaller code and smaller executables means less energy spent when that executable is running.

3. metastate

3.1. Field

Let us postulate the existence of a function:

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

That is (in a simple manner) letting us know what is the state of occupancy of an "field". But what is this "Field thing" ?

C++ "field" is analogous to the database field.

The "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". We will call them : empty and occupied. It is a well known and adopted fact, database field does exist but it can be empty.

As an example, `std::optional` is a well known implementation of a C++ field, sometimes known as "container of one". Class holding only one variable defined in a class. An instance of that class might be tested if it is empty. Not holding a value or "empty". Occupied or "holding a value".

3.1.2. 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 defined as an boolean AND combination of occupancy states of two fields. Namely: Value and Status. A bit more formally.

```
; two possible 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

; metastate is AND combination of
; two fields occupancies
value  ::= field
status ::= field

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

; valstat is made of
valsat ::= value AND status
```

Combination of value *and* status occupancies is giving four possible metastates. We will label them.

| 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 names are just labels, they are not dictating required behaviour.

That is it.

That is the core idea adopters 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 inspect returned values, we inspect the relationship of two fields returned.

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

In C++ there is no need for existence of metastate type. We only care about the relationship between two fields states of occupancy. Metastate capturing is the act of decoding that relationship. Following is canonical capturing of the four metastates using the two fields.

```
// This is not C++.
// two fields are input into the idiom of
// capturing all four possible metastates
//
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*/ }
```

In standard C++ reality we do not need `is_empty()` function from above. Let's assume we use `std::optional` as a field type.

```
// C++
// fields are std::optional instances
auto [value, status] = metastate_enabled_function ();

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

metastate serves well for arriving to cleaner algorithms for complex returns handling. The added benefits are immediate applicability and ability in addressing the requirements from the [Motivation section](#).

This proposal is not describing an [panacea](#). It's value also lies in deliberate simplicity, aiding in solving the interoperability.

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 a metastates carrier

In order to achieve the required wide scope of the metastate coverage, it has to be simple. Metastate actual C++ shape has to be completely context free. Put simply the C++ std lib implementation must not 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. In that respect `std::valstat` is just an recommendation not a mandate.

Synopsis

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

```
#pragma once
// 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 standard 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. `std::valstat` is a recommendation

Type requirements

Both value and status field types, must offer an simple mechanism that reveals their occupancy state. Readily available example of that behaviour is `std::optional` type. In specific contexts a native pointer or any other type can server the same purpose.

What is the meaning of "empty" for a particular type, and what is not, depends on the context. Please see an [example in the appendix](#)

4.1. my::valstat

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

```

// in some 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::optionals.

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

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

// OK metastate captured
// compare the fields occupancy
if ( value && ! status ) {
    /* use the status value from inside a field instance */
    std::cout << "OK metastate captured, value is: " << *value ;
}
```

The other three metastates will be instantiated like so:

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

4.2. valstat but not as we know it

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. For example let us consider some very strict embedded system, run-time environment.

```
// valstat but not as we know it
struct valstat_int_int final {
    int value;
    int status;
};

// both value and status fields in here are int's
auto [ value, status ] = valstat_int_int{ 42, {} }; // OK

// OK metastate captured
// (42 && !0 ) yields true
if ( value && ! status ) { send_signal( value ) ; }

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


That is still standard metastate creating and capturing. It is only we have decided, in that context valstat field types can be two simple int's. We have declared in that context we will think of int as empty if it is zero.

```
// in some specific context int 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 is extremely tight. 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. Callers point of view

Recap. Returns handling: valstat instance 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 project adopting it, the API logic and many other requirements dictated by adopters architects and developers.

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 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 'metastate' type just idioms of metastates capturing.

```
// declaration of a metastate emitting function
// or valstat returning function
template<typename T>
// we use the `my::valstat` type.
// `my::stat` is code value from some code/message mechanism.
my::valstat<T, my::stat >
full_field_info
(database::row /*row_*/ , std::string_view /* field_name */ )
// valstat carry no exception throwing requirements
noexcept ;
```

Primary objective is enabling callers comprehension of a full information passed out of the function. Returns, not just error handling. Satisfying the core requirements from the [motivation section](#).

```
// full return handling after
// the attempted value retrieval
// from the database
// value and status are valstat fields
auto [ value, status ] = full_field_info<int>( db_row, field_name ) ;
```

In this scenario caller is capturing all four metastates.

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

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

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

if ( ! value && ! status ) {
    // metastate captured: empty
    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 "contract", 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 all of them.

```
// implementation in the API namespace
template<typename T>
my::valstat<T, my::stat >
full_field_info
(database::row row_, std::string_view field_name )
```

```

// not throwing exceptions.
    noexcept
{
    // sanity check
    if ( field_name.size() < 1)
        // return ERROR metastate
        return { {}, my::stat::name_empty };

    // using some hypothetical database API
    database::field_descriptor field = row_.fetch( field_name ) ;

    if ( field.in_error() )
        // return ERROR metastate
        return { {}, (my::stat::db_api_error)field.error() };

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

    // db type will have to be cast into the type T
    // we assume T properly handles move/copy etc.
    T field_value{} ;
    if ( false == field.data( field_value ) )
        // return ERROR metastate
        return { {}, (my::stat::type_cast_failed)field.error() };

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

    // just some value
    // 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 too. As long as callers can capture the metastates 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 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.

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

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 readership knows quite well why is that situation considered unresolved. 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 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](#).

"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 is 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 Mysisen, **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 API design issues.

```

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

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.

8.2. Fully functional valstat type, not using std::optional and not using std::valstat.

```

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

    using value_field_type  = T ;
    using status_field_type = std::errc ;

    value_field_type  value;
    status_field_type status;
} ;
}

```

That one valstat variant might be used in a myriad of API's all returning the metastates, carried by the instance of the same type: `errno_valstat<T>`.

```

// all coming from different API's
errno_valstat<const char *> safe_read_line ( FILE * ) noexcept ;
errno_valstat<double> safe_sqrt( double * ) noexcept ;
errno_valstat<const char *> safe_strdup( const char * ) noexcept ;

```

And here are some possible legal 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
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 three above imagined functions can follow the same metastate capturing idiom.

```
// calling any of the three above
// value is a 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.

8.3. Decoupling from the legacy

One can imagine using the metastate paradigm for developing simple but ubiquitous proxy API in front of the legacy API's. (hint: CRT)

```
// notice here we do not 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
}
```

Very simple but fully metastate enabled. The usage:

```
if (auto [ filep , errc ] = modern_fopen( "non_existent_file" , "w+" ); filep )
{
    // filep is a FILE *
    fprintf( filep, "OK" ) ;
} else {
    // ad-hoc usage
    auto message = strerror( ( errc != 0 ? strerror (errc) : "no status" ) ) ;
}
```

Above decouples from decades of "special return values", `errno` globals and POSIX "hash defines" lurking inside any C++ code base today. One can imagine the whole layer of metastate enabled proxy functions, in front of the CRT legacy.
