std::valstat - transparent return type

Document Number:	**P2192R0**
Date	2020-07-03
Audience	SG18 LEWG Incubator
Author	Dusan B. Jovanovic (dbj@dbj.org)

Table of Contents

1. Abstract

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*

1. This is a proposal to introduce a behavioral pattern for optional but standard function return handling.

2. It is an extremely simple type and behavior pattern of both producers and consumers using it.

3. Following is canonical usage, consumers point of view. It tests for the four possible valstat states returned from a function call.

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

// four possible outcomes, aka states
 if (   value &&   status )  { /* info */ }
 if (   value && ! status )  { /* ok   */ }
 if ( ! value &&   status )  { /* error*/ }
 if ( ! value && ! status )  { /* empty*/ }
```

3. Value, applicability and universality of this proposal would be greatly aided by placing one tiny template in the std lib. Concept and benefits are described in this document.

4. This proposal requires no changes in the core language.

2. Introduction

2.1. Guiding principles

1. P2000R1 is probably the closest there is to an set of guiding principles to the better and simpler standard C++.

2. Author does hope, the valstat is seen as strongly in support of the following (P2000R1 -- "Direction for ISO C++", Page 5):

3. *"... we badly need to simplify use of C++ and that leaves us with three alternatives:

- Provide simpler alternatives for simple uses
- Provide simplifying generalizations
- Provide alternatives to complicated and/or error-prone features.

Often, a significant improvement involves a combination of those three..."

4. Perhaps a bold claim, but adoption of this proposal should contributes, to all the three "requirements" above.

2.2. Current State of Affairs

Divergent error handling has fractured the C++ community into incompatible dialects, because of long-standing unresolved problems (in C++ exception handling) -- Herb Sutter, P0709R4

1. Currently (2020 Q3) there is no usable, simple and resilient standard C++ returns concept, or std type. And there is no consensus around such a thing. No thing, no consensus.

2. Time lines for the past possible solutions are measured in years or even decades. Inevitably, sooner or later, each and every project, designs and implements some company, project or library specific concept to handle the non-trivial logic of communicating, from some home grown API.

3. Those efforts are instantly lowering the interoperability between different C++ components, projects and systems, because of using different return types and returns handling concepts.

4. exceptions are alive and well. Majority of solutions are often designed, by "simply" creating specific std::exception hierarchy. Or by using "special" "throwable" types. Or even worse, by mixing the two. Usually the amount of complex code to implement these solutions is many thousands of lines of complex C++.

5. Detrimental effect of exceptions on the final executable is well documented. [6]

6. For the lack of better, primordial C artefact **errno**, is still in a widespread use in modern C++ today (std::errc) .

> system_error is legacy but standard

1. For std::error_code and adjacent types problems please look no further than Summary of SG14 discussion on <system_error>

> There is a number of "modern" return types proposals.

7. They are universally implemented as (very) complex types. Measuring thousands of lines of non trivial C++ code. Through years many of them are abandoned.

8. 2020 Q3 probably the most advanced return type is **outcome**. Please see https://github.com/ned14/outcome. Its single header version measures 7223 lines (6951 sloc). In my humble opinion its usage is far from trivial.

> There is also a "final solution" in the form of P0709

9. Author is in agreement with that concept. That is far reaching core language proposal. Alas, not to be implemented before C++23.

> Inevitably, entropy is on the rise

10. After decades of home grown error handling and accompanying home grown function return types. And there are high profile projects in and out of std lib, meeting these and some more issues head on.

11. Multiplication of function return solutions are gradually complicating the C++ landscape at large, adding the costs and lowering the feasibility of C++ usage. Inevitably entropy rises. Both externally from users point of view, and internally from developers point of view.

2.3. What seems to be the common wisdom

After decades of trying to deliver a common solution the consensus seem to be

1. error or no error is irrelevant question[8]
    1. Returns consuming logic is not always binary:
    2. There are states in between the two
        1. Non expected return might not be an error at all
    3. There are signs the collective is moving in that direction.
2. Complexity is avoided if binary "error or no error" logic is avoided
    1. There are errors when fast exit is not a bad idea
3. Consuming logic is not trivial
    1. It is beneficial to agree to and follow the common return consuming logic
    2. Presence of divergent return types, very quickly raises the level of complexity.

3. Impact On the Standard

1. valstat is a recommended concept, it is not mandated in any way.

2. adopting valstat has no breaking legacy code risks attached to it.

3. valstat adoption should lower the cost of C++ codebase maintenance. The interoperability barriers are be lower because everyone follows the same function return behavioral pattern.

4. At least in this context, C++ community might find it easier to agree on the valstat as an simple "zero cost" mechanism that will make handling function returns, consistent and easy to handle. Both inside the teams and outside for the key stake holders and investors.

4. The Concept

1. Each function will return two instances.
    1. Value
    2. Status
2. Combination of their states is called "Meta State"
    1. The combination is "decoded" by using the boolean AND operator
    2. meta state = state_1 AND state_2
3. The function caller logic is to make decisions by observing the "Meta State" returned.

4.1. Meta State

1. Meta state is state of two or more states. Just like meta language is language of languages, for example.
2. For the purpose of this paper we need two instances and their two states

```
Value  states  = { Empty, Has Value  }
Status states  = { Empty, Has Value  }
```

3. Let's postulate the existence of the state combination function. State combination function receives as arguments, Values and Status instances. It observes their states and returns the resulting meta state.

```
 meta_state = state_combination ( value, status ) ;
```

4. Where meta_state is on of

4.1.1. OK

1. Value instance is in the state "Has Value", Status instance is in the state "Empty". Function returned a value expected. Status is considered as irrelevant.

4.1.2. ERROR

1. Value instance is in the state "Empty", Status instance is in the state "Has Value". Function has **not** returned a value expected. Status instance contains the relevant reason.

4.1.3. INFO

1. Value instance is in the state "Has Value", Status instance is in the state "Has Value". Function returned a value expected. There is also a status message present.

4.1.4. EMPTY

1. Value instance is in the state "Empty", Status instance is in the state "Empty". The "empty return".

```
// function state_combination pseudo code
 meta_state state_combination ( value, status )
 {
    if ( has_value(value) AND is_empty( status) )
       return OK_META_STATE ;

    if ( is_empty(value) AND has_value( status) )
       return ERROR_META_STATE ;

    if ( has_value(value) AND has_value( status) )
       return INFO_META_STATE ;

    if ( is_empty(value) AND is_empty( status) )
       return EMPTY_META_STATE ;
 }
```

4.2. Meta state existence is implied.

1. Adopters of the concept, and its implementation in the shape of std::valstat are using the states of Value and Status. The existence of the "Meta State" instance is required but not implemented. It is implied.
2. Meta state has meaning in the context of API used and function called.

4.3. Is valstat an original concept?

1. There are similar mechanisms or types in few other programming languages. None of them is using the Meta State concept.

2. Most notably GO and Rust are using the dedicated types to deliver non-mandated returns handling concept. (GO online doc: "..This form can be used to improve on a couple of clumsy idioms in C programs: in-band error returns such as -1 for EOF and modifying an argument passed by address..")

3. In contrast to valstat, both are making the key conceptual mistake of handling "error or no error", single state. Both are having only 'error' in the vocabulary. There is no richer return signaling as in 'valstat'. There is no semantic richness of valstat states.

4. Unlike previous concepts, the states signaled using valstat as a concept, are not just one and overly simplistic: "error or not error" state. It is a combination of availability of two pieces of information returned, giving four possible distinct states. Out of two optional values.

5. The Design

"*... behavioral design patterns are design patterns that identify common communication patterns among objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.*" *Source*

5. Proposal is an behavior pattern, with truly tiny implementation in standard C++.

6. Behavior of valstat producers and consumers is almost more important than the type itself. Valstat as a type is almost transparent. It is just a meta state carrier.

7. Both producing and consuming code, are not faced with some special and heavy, return type.

8. Consuming valstat, by using structured binding, does not go against the concept. The concept is to observe together the state of occupancy of value and status returned.

9. Combination of value *and* status are giving four valstat return states aka "meta state" available.

Meta State	Value state	op	Status state
Info	Present	AND	Present
OK	Present	AND	NIL
Error	NIL	AND	Present
Empty	NIL	AND	NIL

10. Both value and status options are always delivered as a pair.

11. This proposal aim is to use standard C++ to do that elegantly, transparently and simply. On both consuming and returning sides of the function divide.

6. Implementation Synopsis

> valstat is just a vessel for passing the meta state.

1. The logic of valstat producing and consuming must be completely context specific. Thus implementation must not dictate the usage besides the core concept.

2. valstat should offer the greatest possible degree of freedom to its adopters. Implementation must be simple, resilient, lightweight and feasible. Almost transparent.

6.1. Canonical implementation

> std::valstat<T,S> as a template is an generic interface whose concrete type allows using the valstat concept.

```
#pragma once
// header <valstat>
// std lib header: <valstat>
#include <optional>

// (c) 2019/2020 by dbj@dbj.org
namespace std
{
 template< typename T, typename S>
    struct [[nodiscard]] valstat
 {
        using value_type = T;
        using status_type = S;

        std::optional<T> value;
        std::optional<S> status;
 };
} // std
```

1. std::valstat just assures the concept existence. It does not mandate its usage. It should exist as a recommended but not mandatory pattern for standard function return creation and handling idiom.

2. Standard "vocabulary" type std::optional is used to implement the: "value is there or not there" concept.

3. The only other feasible way would be to use smart or native pointers. Hopefully there is no need to explain, the increase of complexity in that scenario, although native pointers, is exactly the design option author uses for constrained environments.

4. In this canonical implementation, std::valstat behavior is fully determined by features and capabilities of std::optional.

5. It is in a separate header <valstat>, to allow for complete decoupling from any of the std lib types and headers, but one: <optional>.

6. By using the standard [[nodiscard]] attribute we also enforce the usage of the valstat implementations.

> What about containers?

7. The primary role and purpose of definitions of std::valstat<T,S> is to allow for rich but transparent function return idiom.

8. valstat is not designed to act as element type of various std lib containers.
9. If required by particular adopter, such a secondary role can be easily achieved by the added layer of user code. An effort greatly simplified by using std::optional<T>, sometimes known as "container of one".
10. Why std::optional<T>? Primarily because it encapsulates the solution of the issues of carrying the value, not reference or pointer. standard C++ is value semantics based language. Types handled by std::valstat must be usable in that context. std::optional<\T> requirements will assure that.

7. Usage

7.1. Producers point of view

1. valstat using algorithms, consume and use 4 meta-states available, to shape the code logic.
2. Meta state names are just "names", they do not dictate consuming site logic or behavior. That is left completely to the valstat adopters.
3. API implementers / users, are free to choose if they will use and return them all, one,two or three meta-states. Please consider one simple (but complete) function, built using the valstat.

```
// use the std::valstat alias template definition
// status type is determined as std::string_view
// value type is a template argument
template<typename T>
using my_valstat = std::valstat<T, std::string_view >;

my_valstat<int> add(int lhs, int rhs)
// valstat enabled API does not throw exceptions
   noexcept
{
 if (lhs >= 0) {
  if (INT_MAX - lhs < rhs) {
   // creating valstat instance with empty value option
   // status option contains a string
   // signal the ERROR meta-state
   return {{}, "integer overflow" };
  }
 }
 else {
  if (rhs < INT_MIN - lhs) {
   // signal the ERROR meta-state
   return {{}, "integer overflow" };
  }
 }
 // signal the OK meta-state
 // with value option set
 // status option remains empty
 return { lhs + rhs, {}} ;
}
```

7.2. Consumers point of view

1. valstat return delivers four possible meta-states to be utilized; or not. How and why the consuming code will look, that completely depends on the code author.

2. Most of the time valstat consumers use a structured binding. Both value and status are inside std::optional. There is no 'special' over-elaborated return type required.

```
int int_1 = 42, int_2 = 13 ;

auto [ value, status ] =  add( int_1, int_2 ) ;

// "error" meta-state
if ( ! value && status ) {
/* no value here means 'error'
  caller can use the status if returned,
  proceed following the consuming site logic
*/
}

// "empty" meta-state
if ( ! value && ! status ) {
/*
both value and state are empty
proceed following the consuming site logic
*/
}

// "ok" meta-state
if ( value && ! status ) {
/* value and no status in this context means 'ok',
   proceed following the consuming site logic
*/
}

// "info" meta-state
if ( value && status ) {
/* both value and status are not empty
  proceed following the consuming site logic
*/
}
```

3. Let us re-emphasize: Not all possible four meta-states need to be processed. It depends on the context of the consumers site.

4. Definitely not recommended "shortcuts" are entirely possible

```
if ( auto v_ = add(int_1, int_2).value; v_ )
{
    // v_ aka value is not empty
}
```

1. > valstat is not standing in a way.

7.3. Type requirements

> There are no valstat specific type requirements.

1. The only valstat requirements for the value and status types, are the ones imposed by the std::option type.
2. Over-elaborating by using some special status or value types made specifically to be used with valstat is certainly possible but not recommended.
3. Author has not yet encountered a real need to go beyond strings or fundamental types used as status type.

> Remember: Four valstat meta-states do not depend on the actual value or actual status values.

4. value type can be almost "anything", std::optional allowing.
5. Can concrete valstat template definition (aka type) be moved, copied, assigned, swapped etc, entirely depends on std::optionals embedded.

7.4. What about the constructors?

1. valstat is not core language extension. Thus it can not solve the issue of constructors not throwing extensions but needing to report the outcome.
2. One way to offer a remedy is additional constructor argument as valstat reference.

7.5. Functional referential transparency

1. Valstat makes C++ functions with no side effects possible. Valstat allows for much cleaner C++ functional programing.
2. Honorable readership of this paper is of course well aware, in pure functional languages functions cannot have side effects. That is, they are completely determined by what they return, and cannot do anything else. Most notably they cannot throw exceptions, or set globals.
3. This allows for **referential transparency**: f(42) in purely functional code, can be evaluated once, and the value it returns can be replaced for every other instance of f(42) in the program.

7.6. C interoperability

1. It is entirely possible to use the valstat concept with the C code. And then to transparently consume such a code from C++. Here is a quick example of one of several possible designs.

```
/* C code */
typedef struct int_valstat {
    int * value ;
    const char * status ;
} int_valstat ;
```

```
int_valstat valstat_compliant_c_api ()
{
   static int v42 = 42;

   /* OK state return */
   if ( time_is_right() )
    return (int_valstat){ & v42, NULL };

   /* ERROR state return */
    return (int_valstat){ NULL, "error in valstat_compliant_c_api" };
} ;
```

C consumers point of view

```
/* C code consuming C valstat */
int_valstat valstat = valstat_compliant_c_api () ;

   if ( valstat.value ) {
      return *valstat.value ;
   }
   if ( valstat.state )
   {
      async_log ( *valstat.state );
   }
```

For the C++ consumers, usage idiom is completely unchanged, vs calling some other C++ API.

```
// cpp code consuming C function
auto [value, status] = valstat_compliant_c_api () ;

   if ( value ) {
      return *value ;
   }
   if ( state )
   {
      async_log ( *state );
   }
```

2. If the called valstat enabled function is C or C++, that is entirely transparent to the C++ caller.
3. Added benefit is in having richer signaling from legacy C API's, vs single state "error or no error" legacy concept.

From entirely different context but very fitting sentence:

"..This form can be used to improve on a couple of clumsy idioms in C programs: in-band error returns such as -1 for EOF and modifying an argument passed by address.." -- (online documentation about GO errors package)*

7.7. ABI

1. Of course, C functions returning C valstat structs are eminently usable to add value when solving difficult C++ ABI situations. Solved with C API in front of the C++ component.

2. Using valstat enable C API there are no ABI boundaries to a valstat concept.

7.7.1. OS specific error codes

1. all of the major OS's are using C like, more or less elaborate, "specially coded" values as error codes.
2. mixing valstat and OS specific error codes is obviously very simple.

```
// WIN32 API + valstat
#include <winerror.h>
#include <valstat>

    template<typename T>
    using valstat_hresult = std::valstat<T, HRESULT>;

    // info state, both value and status
    valstat_hresult<int> info_ = { 42,  ((HRESULT)0x00000000L) };
```

7.8. Restricted run time environments

1. Last but not the least, **valstat** makes coding without exceptions much easier.

2. API using valstat is not throwing exceptions; consuming code is using simple local return handling.

3. For restricted runtime, users need non STL C++ valstat variant. Think gaming, IoT, medical instruments, etc. Non STL valstat option **might** look like this synopsis:

```
// valstat variant for
// restricted environments
// still standard C++
template< typename T, typename S>
struct [[nodiscard]] valstat_light final
{
   using type = valstat ;
   using value_type = T;
   using status_type = S;

   value_type * value{};
   status_type * status{};
}
```

4. Author is aware, one of the foundations of standard C++ is value and reference semantics.

5. That C++ template is still valstat concept. But, without the STL presence.

6. Consuming valstat_light has no conceptual difference vs consuming std::valstat.

7. Of course proper care has to be taken to assure value or status are not dangling pointers.

8. Summary

"*.. one of the hardest pieces to write is error handling..*" -- B. Stroustrup, CppCon2019 Key Note

What might be the components of valstat success?

1. **valstat** is primarily a concept; an behavioral pattern. That is almost all adopters have to adopt. The rest is optional.
2. **valstat** is not mandatory
3. **valstat** has almost universal applicability.
4. **valstat** is simple, consistent, logical, easy to understand and feasible to use.
5. **valstat** is extremely light, easy to optimize code. It does not stand in the way of any C++ project to design and use project specific idioms based on valstat. Or not.
6. **valstat** as a template is abstract interface.
7. **valstat** has the potential of increasing interoperability for the C++ landscape at large. "Everyone" using the valstat idiom, means that everyone will consume each others API easier. Even if types returned are "foreign" to the logic of the caller.
8. **valstat** is not "all or nothing" proposal. It can be very gradually introduced.

8.1. Concept ubiquity is the proof of its value.

1. valstat has the potential to improve and simplify the function return universally across the C++ landscape at large. And, simplifications inevitably brings resilience.

2. The more universally it is used, more valuable std::valstat will be. And if it has the prefix std:: it has a chance of becoming truly adopted and useful.

3. valstat consuming code has local return handling. valstat consuming function are undeniably more complex. Writing safer code is not simpler that the opposite.

4. Both writing producing and consuming valstat enabled API is undeniably more complex. But not necessarily slower code.

5. Hardly anyone might be worried it will increase compilation times or slow down execution times considerably.

6. Naturally valstat usage leads to simpler executable because there is no hidden exceptions machinery. Which also gives to the compiler the opportunity to fully optimize the code.

7. After all truly performance critical code need not use valstat.

> Fastest code is not checking function returns. That is certainly not making it resilient, low maintenance code.

2. Barriers to the ubiquity of valstat, as ever with new concepts, will be non-technical. valstat is not standing in the way in any sense.

> C++ community urgently needs to agree on a standard recommended solution for non trivial function returns. Now.

That is the problem author is hoping to solve with this proposal. Time will tell.

---

9. References

- [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
- Craig Ben, **Error size benchmarking: Redux** , [6] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1640r1.html
- Vicente J. Botet Escribá, JF Bastien, **Utility class to represent expected object**,[7] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0323r3.pdf
- Shoop Kirk, **Cancellation is not an Error**, [8] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1677r0.pdf