N3599: Lingua franca Results

Document #:	N3599
Date:	2025-06-22
Project:	Programming Language C

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

C is the glue which stitches together all the major programming languages. Every programming language and operating system that anybody ever used for anything serious can speak C, and that is highly unlikely to change for decades yet to come.

There is, however, a topic on which most major programming languages can discuss with strong interoperability and commensurateness, but for which standard C has no support whatsoever. This is an object with the form of:

1. Pointer to constant partially opaque descriptor of payload.
2. Variable length payload.

Let us call this a *Result*. The partially opaque descriptor would have the following universal operations:

- Does the Result represent a failure?
- Destroy the Result's value.
- Attempt to clone the Result's value.
- Is the Result semantically equivalent to another Result?
- What is the most similar `errno` value to this Result (if any)?
- Get `const char *` description of the Result value.
- Get globally unique identifier of the Result *domain*'s class (i.e. the partially opaque descriptor of payload) so different instances of the domain can be categorised as similar.
- Get `const char *` description of the domain's name.

Results can be implemented by any C speaking language and **losslessly** encode the original cause of a failure or success as defined by the programming language creating the Result. The above universal operations allow any *other* programming language (including C) to interrogate the Result with sufficient fidelity that one can determine if that Result is equivalent to a local language success or failure condition.

As a quick example, if C++ throws a C++ exception, that C++ exception can be **losslessly** wrapped into a Result value. That Result value can be passed via C into a Rust codebase which has direct language support for Results. Within Rust, you can `match` the Result to a 'file not found'

Rust error code, and if the C++ exception thrown is semantically equivalent to 'not found', it will be matched.

The exact same operation can flow in the opposite direction i.e. Rust Results can be `try...catch` in C++ and they will match the appropriate C++ exception type. This also goes for any language combination e.g. Python exceptions wrapped into a Result might flow via C++ then via Rust then to .NET. The original, unmodified, Python created Result can be understood within .NET, and all the layers in between don't matter due to the lossless value semantics.

Obviously, if a Result begins in a programming language and eventually returns to that language, the complete state can be retrieved in full. This lets you 'package up' state into a Result and retrieve it if that Result happens to return to your programming language. Of course, if a Result gets destroyed elsewhere it is correctly cleaned up by calling a routine defined by the Result's code's domain.

1 History and background

The proposed facility has been shipping in the Boost C++ Libraries since 2018 [1] and it is running right now on billions of devices worldwide. This author has been told of it shipping in every Microsoft Windows, every Linux, every Mac OS, every Android and every iOS device for some years now – and it is currently believed that it is running in some satellites orbiting planet Earth. It is, by now, extremely well tested and has two known complete reimplementations from scratch of the reference implementation.

It is actually the `std::error` reference implementation from 2019's [WG21 P0709] *Zero-overhead deterministic exceptions*, and it has had a long and unfortunate history at WG21 via [WG21 P1028] *SG14 `status_code` and standard `error` object* which – if I am quite blunt about it – has not shown WG21 in a good light when standards bodies are supposed to be about *standardising existing practice*.

The reference implementation has always had a C API which originally implemented a subset of the total available functionality, but in recent years user demand for an ever richer C API has meant the C API has become full fat. I would propose that that C API be standardised by WG14 as **the lingua franca Result type** to interoperate across all C speaking languages.

2 An examples of what the API would look like

We must first tell C about the type, then tell the library runtime additional information about the Result.

```
// Declare to C a Result with a happy value of intptr_t
STD_DECLARE_RESULT_SYSTEM(result_int, intptr_t)

// Save oneself typing out STD_RESULT_SYSTEM(result_int) all the time
typedef STD_RESULT_SYSTEM(result_int) result;

// Our custom C enum
enum c_enum
{
```

```
  c_enum_not_found,
  c_enum_bad_argument
};

// Make a custom status code domain for this C enum
STD_DECLARE_RESULT_SYSTEM_FROM_ENUM(result_int,          // The C Result type declared above
    c_enum,                                              // The C enum we wish to wrap
    "{74ceb994-7622-3a21-07f0-b016aa705585}",            // Unique UUID for this domain
    // Mappings of C enum values to textual description and semantic equivalances to generic codes
    (
      (c_enum_not_found, "item not found", (ENOENT /* there can be a list here */)),
      (c_enum_bad_argument, "invoked wrong", (EINVAL))
      /* this list can continue */
    ))

// Make helper macros
#define SUCCESS(v) STD_MAKE_RESULT_SYSTEM_SUCCESS(result_int, (v))
#define FAILURE(v) STD_MAKE_RESULT_SYSTEM_FROM_ENUM(result_int, c_enum, (v))
```

Macro `STD_RESULT_SYSTEM(result_int)` expands into:

```
// For exposition, this is a concrete type
struct std_status_code_system
{
  void *domain;
  intptr_t value;
};

// What the macro expands into
struct std_result_status_code_system_result_int
{
  intptr_t value;
  unsigned flags;
  struct std_status_code_system error;
};
```

Structs rather than unions are used to aid ABI portability across programming languages. The member `.value` is the success value; the member `.error` is the failure value; bits within `.flags` indicate which member is in use and additional information about the Result. Macro `STD_RESULT_SYSTEM(result_int)` is typedefed into `result`.

Let's see our C implemented Result in action:

```
result positive_only(int x)
{
  if(x < 0)
  {
    return FAILURE(c_enum_bad_argument);
  }
  return SUCCESS(x);
}

bool test(int x)
{
  result r = positive_only(x);
```

```
  if(STD_RESULT_HAS_ERROR(r))
  {
    if(STD_STATUS_CODE_EQUAL_GENERIC(&r.error, EINVAL))
    {
      fprintf(stderr, "Positive numbers only! Error was '%s'\n", STD_STATUS_CODE_MESSAGE(&r.error));
      STD_RESULT_DESTROY(r);
      return false;
    }
  }
  STD_RESULT_DESTROY(r);
  return true;
}
```

You can see semantic equivalence testing at work above via the `STD_STATUS_CODE_EQUAL_GENERIC()` macro where we test if the Result is semantically equivalent to an invalid argument `errno` value. As in the declaration of the custom status code domain above we declared that the `c_enum_bad_argument` custom enum value is semantically equivalent to invalid argument, that operation would in this case return true if `test()` is called with a negative number.

Finally, there is a C macro based try operation:

```
result test2(int x)
{
  STD_RESULT_SYSTEM_TRY(int v,                        // what to set to value if successful
        fprintf(stderr, "Positive numbers only!\n"),  // what cleanup to run if unsuccessful
        positive_only(x));
  return SUCCESS(v + 1);
}
```

This macro invokes function `positive_only(x)`. If it returns success, `int v` is set to the success value returned, `STD_RESULT_DESTROY()` is called and the function returns `SUCCESS(v + 1)`. If the function returns failure, the cleanup stanza (the `fprintf()`) is run, and then the `result.error` returned by `positive_only(x)` is returned immediately by function `test2()` in its `.error` member.

This try operation is exactly the same as in many programming languages – it is an explicit control flow operation to save on typing out boilerplate to propagate failures. You can, of course, inspect the Result and handle it manually if you wish.

3 Conclusion

The proposal is 100% library only and requires nothing additional in the C language. Results are a bit clunky to use from C, any macro based API does cause more typing than one would prefer. But it's manageable, and I personally have implemented a large C23 codebase entirely using these Results without any issues. That C23 codebase interoperated with C++ and Rust, and provided completely transparent pass through of failures and successes between all three programming languages. Performance was excellent as well: on any out of order CPU designed in the past two decades the result passing ceremony is usually free of measurable runtime cost [2].

If WG14 decides that they wish to add this to the C standard library, I would as with all my other proposals create a pure C reference implementation suitable for drop into any C toolchain. I would

like strong committal from the committee before I embark on this – I would not want to implement the library, undergo seven years of committee discussion, for it then to get rejected as with what happened at WG21.

4 References

[N2289] Douglas, Niall
Zero overhead deterministic failure
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2289.pdf

[N2429] Gustedt, Jens
Function failure annotation
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2429.pdf

[WG21 P0709] Sutter, Herb
Zero-overhead deterministic exceptions: Throwing values
https://wg21.link/P0709

[WG21 P1028] Douglas, Niall
SG14 status_code and standard error object for P0709 Zero-overhead deterministic exceptions
https://wg21.link/P1028

[1] *Boost.Outcome*
Douglas, Niall and others
https://ned14.github.io/outcome/

[2] *Boost.Outcome runtime overhead*
Douglas, Niall and others
https://ned14.github.io/outcome/faq/#what-kind-of-runtime-performance-impact-will-using-outco