N3766: C Lingua Franca Results

Document #:	N3766
Date:	2025-12-04
Project:	Programming Language C

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

At the Brno meeting, I presented [N3599] *Lingua franca Results* which pitched a universal failure interoperation mechanism for all C speaking languages. This is the same proposed C++ `std::error` implementation from 2019's [WG21 P0709] *Zero-overhead deterministic exceptions* at WG21, except for C. The C++ reference implementation 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 had two known complete reimplementations from scratch of the reference implementation.

I say 'had', because there are now *three* complete reimplementations from scratch, and you can find a shiny new pure C implementation at `https://github.com/ned14/wg14_result`. It is known to work on x86, x64, ARMv7, AArch64, and RISC-V (ESP32). It requires C 11 to compile, but its header files only require C 90. It has language bindings support for:

- All SWIG supported languages: C#, D, Go, Guile, Java, Javascript, Lua, OCaml, Octave, Perl, PHP, R, Ruby, Scilab, Tcl/Tk. And FORTRAN via a special fork of SWIG.
- Rust via Rust FFI bindgen.
- C++, in which the C object lifetime management functions are called for you (this is optional, and can be disabled).

The reference implementation supports unity builds, header only builds, and it is 100% ABI compatible with the C++ reference implementation at [1] to the point that you can freely cast between pointers to the C++ implementation and the C implementation, or mix and match C++ and C implementation freely. The reference implementation can be dropped into any C 11 or later standard library.

1 Examples of use

Before I write out all the proposed normative wording, I'd like to find out if WG14 likes the current API, or if it hates the API so much it's a showstopper to further standardisation.

You may find when reading the following examples that the reference API documentation at `https://ned14.github.io/wg14_result/` is useful. I apologise for it being the standard doxygen

generated reference API docs format, but we as an industry continue to fail to create a comprehensively superior replacement.

1.1 Example of use in end user code in C

```c
// To declare to C a Result with an int value type, we use the
// STDC_RESULT_DECLARE(T, name) macro. This declares the type
// and any support functions for that type.
STDC_RESULT_DECLARE(int, int)

// We then typedef it to something concrete
typedef STDC_RESULT(int) result_int;

STDC_RESULT_DECLARE(bool, bool)
typedef STDC_RESULT(bool) result_bool;

// A Result with a void (i.e. none) value type is predeclared.
typedef stdc_result_with_void result_void;

result_int test1(int x)
{
  if(x > 0)
  {
    // Return a successful Result named int with value x
    return STDC_RESULT_MAKE_SUCCESS(int, x);
  }
  // Return a failure Result named int with status code
  // made from the enum stdc_status_code_errc
  return STDC_RESULT_MAKE_FAILURE(
    int, stdc_status_code_system_make_from_errc(
       stdc_status_code_errc_argument_out_of_domain));
}

result_int test2(int x)
{
  // Perform a TRY operation which evaluates an expression.
  // If that expression returns a failed Result, this operation
  // early exits this function returning that same failure
  // immediately. Otherwise it extracts the successful value
  // within the returned Result into the first macro argument.
  // If we early exit, we run the second macro argument just
  // before we return.
  //
  // The TRY macro has argument count based overloading, so
  // depending on argument count it can do more stuff. This
  // example has three macro arguments: out, cleanup, expression.
  STDC_RESULT_TRY(
    int r /* variable to extract any successful value into */,
    printf(
      "test2 finds test1 returns failure.\n") /* cleanup to perform if TRY fails */,
    test1(x) /* result returning expression to TRY */);

  printf("test2 finds test1 returns successful value %d.\n", r);
  return STDC_RESULT_MAKE_SUCCESS(int, r);
}

```

```c
result_void test3(int x)
{
  // This is an example of a four macro argument TRY operation:
  // out, rettype, cleanup, expression.
  STDC_RESULT_TRY(
    int r /* variable to extract any successful value into */,
    void /* override the return result type */,
    printf(
      "test3 finds test1 returns failure.\n") /* cleanup to perform if TRY fails */,
    test1(x) /* result returning expression to TRY */);

  printf("test3 finds test1 returns successful value %d.\n", r);

  // This is how to return a successful Result with no value type.
  return STDC_RESULT_MAKE_SUCCESS(void, );
}
```

1.2 Example of use in end user code in Rust

Rust's FFI `bindgen` tool has no ability to see C preprocessor macros, so we have a little bit of shim Rust code to tell Rust about a C layout compatible C Result type. That type isn't compatible with Rust's own Result type, so we provide a `to_result()` function which losslessly converts a C Result type into a Rust Result type, after which Rust's TRY operator works exactly as normal.

It should be noted that the Rust Result uses an error type of `stdc_status_code_system` i.e. it's the original error value, and therefore this works without touching the original failure data. We also tell Rust how to display one of these, Rust will automatically go fetch the status code message string as needed. The C Result type therefore works 100% perfectly in Rust, the integration is as good as in C++.

```rust
// Declare an extern C function returning a C Result with int value
unsafe extern "C" {
    pub fn test1(x: ::std::os::raw::c_int) -> result_with<::std::os::raw::c_int>;
}

// This is a Rust function returning a Rust Result with no value
fn test2(x: i32) -> WG14Result<()> {
  // Call the extern C function, which returns a C Result
  let c_res = unsafe { test1(x as ::std::os::raw::c_int) };

  // Convert the C Result into a Rust Result and perform Rust TRY operation
  // This works exactly the same as the C macro TRY operation above
  let _ = to_result(c_res)?;

  // Return success
  Ok(())
}
```

1.3 Example of use defining a custom status code domain

This is something which can be automated away in C++ using metaprogramming. Unfortunately, without compiler magic to stamp out the boilerplate, we have to stamp out that boilerplate manually which is tedious. Still, it all works just fine.

```c
// This is the enum we are going to wrap into a status code domain
enum Code
{
  Code_success1,
  Code_goaway,
  Code_success2,
  Code_error2
};

/* We will need to implement each of the operations that a status code domain
implements. These correspond to virtual member functions in the C++
implementation, so we need to use a macro to add in the appropriate C++
__thiscall calling convention sugar.
*/
static int STDC_RESULT_VTABLE_API(
Code_domain_vtable_name,
struct STDC_RESULT_PREFIX(status_code_domain_vtable_name_args) * args);

static void STDC_RESULT_VTABLE_API(
Code_domain_vtable_payload_info,
struct STDC_RESULT_PREFIX(status_code_domain_vtable_payload_info_args) * args);

static bool
STDC_RESULT_VTABLE_API(Code_domain_vtable_failure,
                       const STDC_RESULT_PREFIX(status_code_untyped) * code);

static bool
STDC_RESULT_VTABLE_API(Code_domain_vtable_equivalent,
                       const STDC_RESULT_PREFIX(status_code_untyped) * code1,
                       const STDC_RESULT_PREFIX(status_code_untyped) * code2);

static void STDC_RESULT_VTABLE_API(
Code_domain_vtable_generic_code,
struct STDC_RESULT_PREFIX(status_code_domain_vtable_generic_code_args) * args);

static int STDC_RESULT_VTABLE_API(
Code_domain_vtable_message,
struct STDC_RESULT_PREFIX(status_code_domain_vtable_message_args) * args);

/* Define the vtable for the status code domain. Being constexpr means
most compilers will place it into the const section of the linked binary,
exactly the same as a C++ vtable. This below is compatible with the vtable
layouts for GCC, clang and MSVC, but it may not be so on some toolchains.
If not, we may need to add a macro wrapper to insert platform specific
vtable sugar.
*/
static constexpr
stdc_status_code_domain_vtable Code_domain_vtable = {
  Code_domain_vtable_name,
  Code_domain_vtable_payload_info,
  Code_domain_vtable_failure,
  Code_domain_vtable_equivalent,
  Code_domain_vtable_generic_code,
  Code_domain_vtable_message,
  nullptr,
```

```
  stdc_default_erased_copy_impl,
  stdc_default_erased_destroy_impl
};

/* Define a domain instance. It need not be a unique instance, uniqueness
is determined by a randomly chosen integer. That integer can be specified
using a UUID, the macro below parses the string at constexpr time and emits
an integer. Again, the below is compatible with a C++ object with virtual
functions i.e. a vptr for GCC, clang and MSVC, but it may not be so on some
toolchains. If not, we may need to add a macro wrapper to insert platform
specific vptr sugar.
*/
static constexpr
stdc_status_code_domain Code_domain = {&Code_domain_vtable,
                                       STATUS_CODE_DOMAIN_UNIQUE_ID_FROM_UUID(
                                       "430f1201-94fc-06c7-430f-120194111111")
};

// After all the boilerplate is done, declare a status code
// named Code with a payload type of enum Code
STATUS_CODE_WITH_PAYLOAD_DECLARE(enum Code, Code)

// And typedef that type to a concrete alias
typedef STATUS_CODE_WITH_PAYLOAD(Code) StatusCode;

/* Prints:

StatusCode failure has value 1 (goaway) is success 0 is failure 1

*/
int main(void)
{
  constexpr StatusCode
  failure2 = STATUS_CODE_WITH_PAYLOAD_MAKE(Code, Code_goaway);

  printf("StatusCode failure has value %d (%s) is success %d is failure %d\n",
         failure2.value, status_code_message(failure2).c_str,
         status_code_is_success(failure2), status_code_is_failure(failure2));
  return 0;
}
```

2 References

[N3599] Douglas, Niall
Lingua franca Results
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3599.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