Deterministic Disappointment

Niall Douglas

Contents:

1. What is disappointment?
2. What is determinism?
3. The direction of C++
4. Future disappointment in C++?
5. Achieving the future today
    a. C++ 11 `<system_error>`
    b. C++ 11 P1028 SG14 `status_code`
    c. C++ 14 (Boost.) Outcome

What is disappointment?

Aspects of disappointment

- Used in wide, not narrow contracts
  - OR, in wide->to->narrow contracting!
- Programmer anticipated (i.e. likely) failure handled differently to programmer unanticipated (i.e. exceptional) failure
- Current best practice for new C++ code bases e.g. Filesystem, Networking

```
bool std::filesystem::copy_file(
           const std::filesystem::path &from,
           const std::filesystem::path &to);
```

```
bool std::filesystem::copy_file(
           const std::filesystem::path &from,
           const std::filesystem::path &to,
           std::error_code &ec);
```

What is determinism?

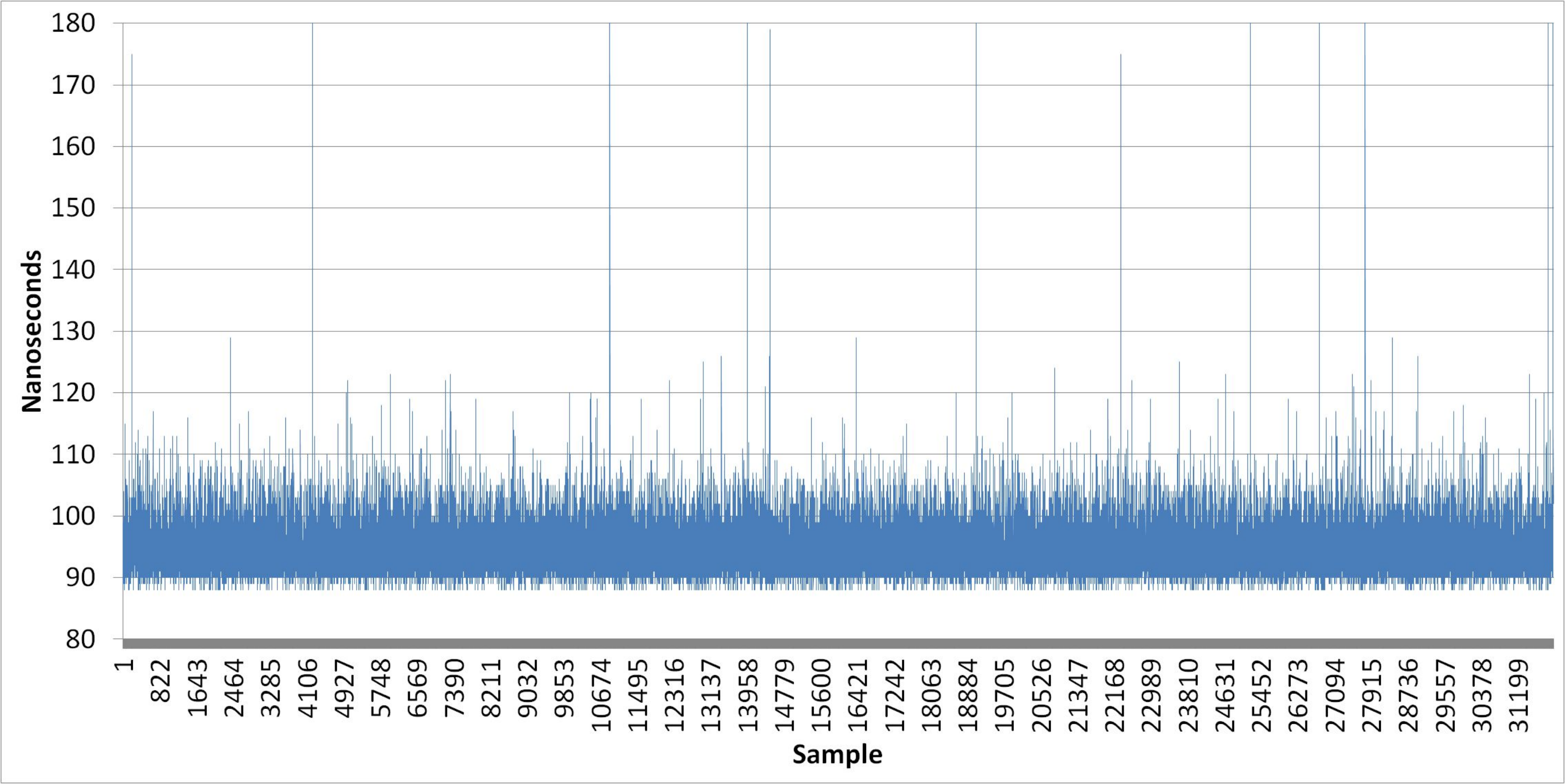
Determinism

- NOT, I repeat NOT, amortised predictability
- NOT, I repeat NOT, median or mean

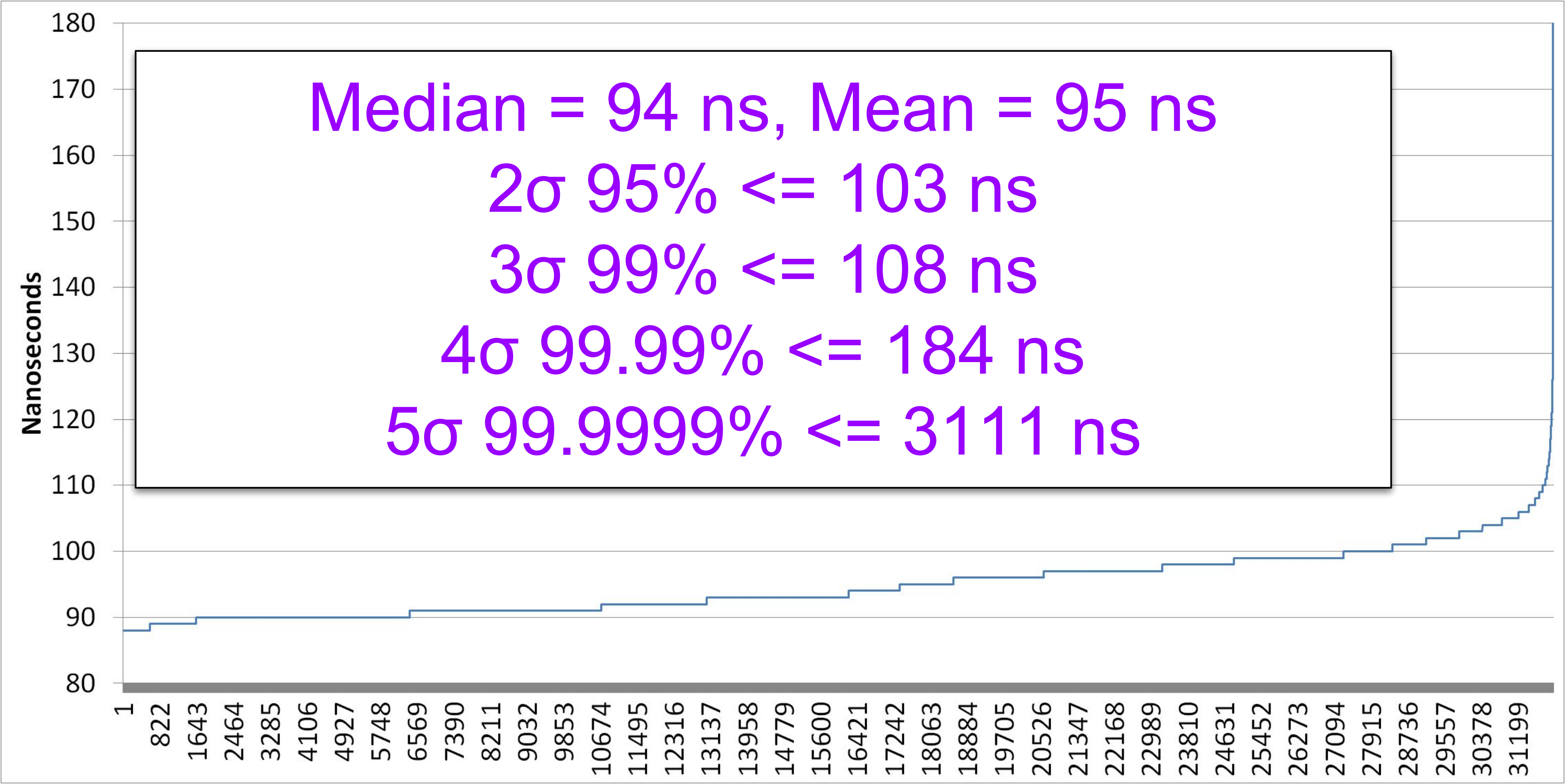
Has a very specific meaning:

1. Worst possible execution in time or space
2. OR worst possible execution at 2 - 5 sigma

   (~95%, ~99%, ~99.99%, ~99.9999%)



Random 4Kb memcpy in 100Mb region of RAM on Haswell



Median = 94 ns, Mean = 95 ns

2σ 95% <= 103 ns

3σ 99% <= 108 ns

4σ 99.99% <= 184 ns

5σ 99.9999% <= 3111 ns

The Direction of C++

by Beman, Howard, Bjarne, Daveed & Michael

https://wg21.link/P0939 quote 1:

“C++ rests on two pillars:

- *A direct map to hardware*
- *Zero-overhead abstraction in production code”*

“Depart from those and the language is no longer C++”

https://wg21.link/P0939 quote 2:

“Over the long term, we must strengthen these two pillars:

- *Better support for modern hardware*
- *More expressive, simpler, and safer abstraction mechanisms (without added overhead)”*

Future disappointment in C++?

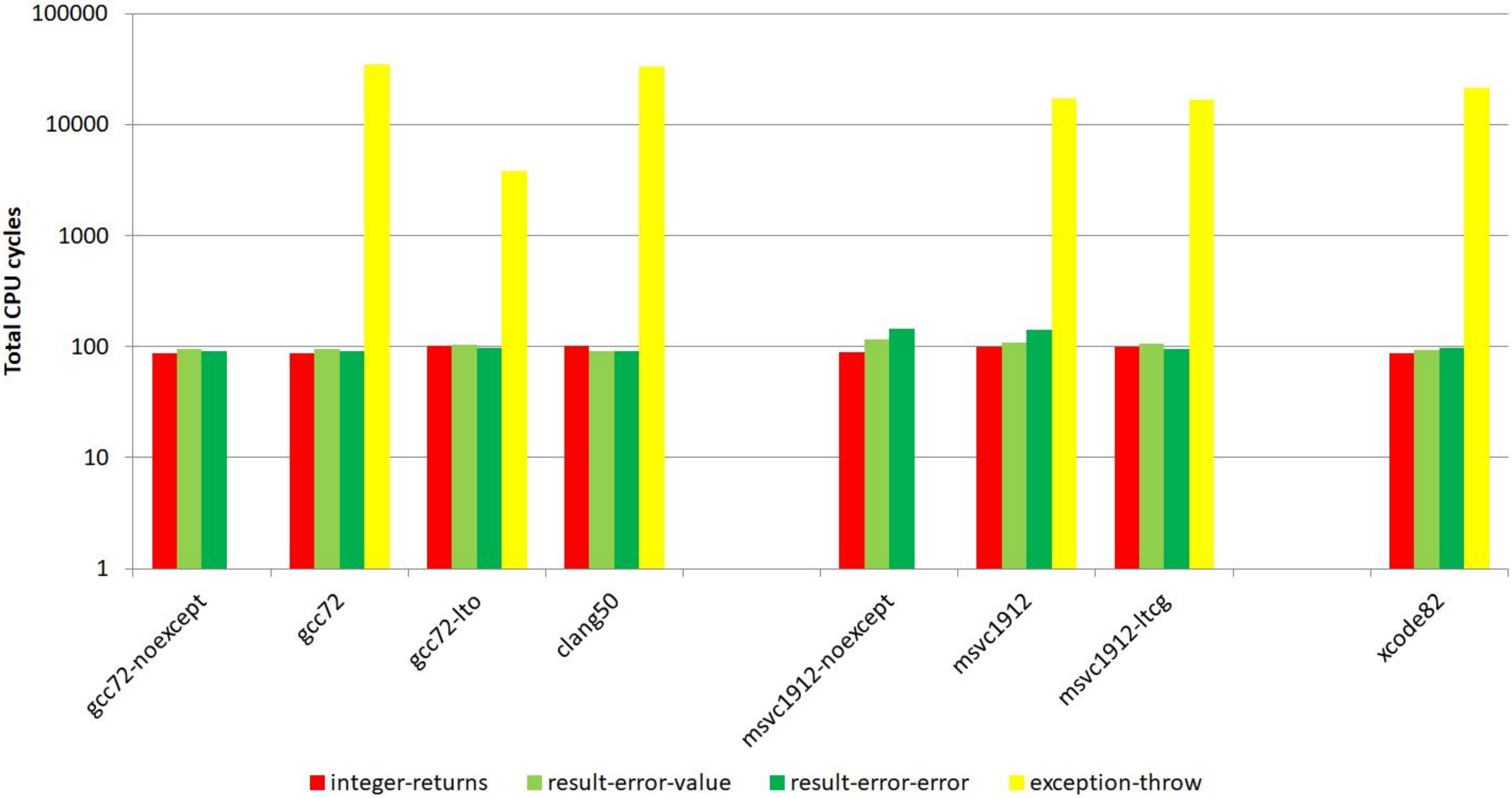
History of C++ exceptions

- Added to Cfront in 1992 by HP
  - After much consensus building!
- The following assumptions were made for the design:
  - Are used primarily for (abort, not resume) error handling
  - Are rare compared to function definitions
  - Occur infrequently compared to function calls

History of C++ exceptions

- “Zero overhead” in the successful code path except for:
    - Inhibits code folding by the optimiser
        - Increased CPU cache loading
    - Adds 15-38% to final binary size due to EH tables
        - Games, embedded folk simply disable exceptions altogether
- And hideously slow for the throw-catch!

Cost of returning error up ten stack frames on x64



History of C++ exceptions

- Lots of C++ coding guidelines ban their use
  - Value added not worth their cost in terms of maintenance, extra testing, and bugs introduced
- End up with lots of C++ incompatible with lots of other C++ due to lack of exception safety
  - Can’t use STL in games
  - Can’t allow exceptions to pass through Qt

P0709: Zero overhead deterministic exceptions - Throwing values

by Herb

https://wg21.link/P0709

P0709 Zero overhead exceptions

- New alternative exception mechanism
  - Value-based in addition to type-based
  - Value throws are always of `std::error` which is defined to be no more than two CPU registers in size (note `std::error_code` exactly ticks this box)
- Code can throw exceptions via old or new mechanisms
  - Required for backwards binary compatibility

P0709 Zero overhead exceptions

- The “non-recoverable” exceptions `std::bad_alloc`, `std::logic_error` etc become default process terminating
- Anywhere in the STL which was not `noexcept` due to potential `bad_alloc`, `logic_error` etc becomes `noexcept`
- (`std::error_code`& overloads in standard library get deprecated)

```
int safe_divide(int i, int j) throws {
  if (j == 0)
   throw arithmetic_errc::divide_by_zero;
  if (i == INT_MIN && j == -1)
   throw arithmetic_errc::integer_divide_overflows;
  if (i % j != 0)
   throw arithmetic_errc::not_integer_division;
  else return i / j;
}

double caller(double i, double j, double k) throws {
  return i + safe_divide(j, k);
}
```

```cpp
int caller2(int i, int j) {  // no throws!
  try {
    return safe_divide(i, j);
  } catch(error e) {
    if (e == std::errc::result_out_of_range)
      return 0;
    if (e == std::errc::invalid_argument)
      return i / j; // ignore
    if (e == std::errc::argument_out_of_domain)
      return INT_MIN;
    throw std::system_error(e);  // Throw as type-based
  }
}
```

Summary

- Opt-in value-based throws replace EH table bloat with fatter, cache heavier, code
    - BUT which is more optimisable, foldable, etc
- Makes the STL much less “throwey”
    - Becomes useful to exceptions-disabled users
- Exception throws become as lightweight as control flow
    - BUT still comes with control flow inversion

P1095R0/N2289: Zero overhead deterministic failure - A unified mechanism for C and C++

by Niall (me)

https://wg21.link/P1095

P1095 Zero overhead failure

“A proposed universal mechanism for enabling C speaking programming languages to tell C code, and potentially one another, about failure and disappointment”

- One possible implementation of P0709
- Implements the value-based exception throw mechanism into the C language

P1095 Zero overhead failure

- For C functions marked `fails(E)`, calling convention changes to:
  - Return `union` of declared function return type `T` and failure type `E`
  - Discriminant is returned via some architecture-specific lightweight mechanism
    - E.g. CPU carry flag
  - Fails-functions must be explicitly called with `catch(...)` or `try(...)`

P1095 Zero overhead failure

- There is a boilerplate expansion `fails_errno` which causes the setting of `errno` to be returned via `fails(struct { T; int; })` instead
  - This enables lots of currently impure C and POSIX functions to be marked pure e.g. `<tgmath.h>`
  - Improves math code optimisation significantly
  - This neatly sidesteps a major problem before WG21 for the last four years

P1095 Zero overhead failure

- In C++, functions may be marked `throws`, `throws(E)`, `fails(E)`, `noexcept` or nothing
    - `fails(E)` functions require explicit calls of throws/fails functions via `try(...)` and `catch(...)` - solves the flow inversion problem!
    - `throws` functions silently inject a `try(...)` around any calls of throws/fails functions if not otherwise specified

```
int safe_divide(int i, int j) fails(arithmetic_errc) {
  if (j == 0)
    return failure(divide_by_zero);
  if (i == INT_MIN && j == -1)
    return failure(integer_divide_overflows);
  if (i % j != 0)
    return failure(not_integer_division);
  else return i / j;
}

double caller(double i, double j, double k) fails(arithmetic_errc)
{
  return i + try(safe_divide(j, k));
}
```

```c
int caller2(int i, int j) {
  struct {
    union { int value; arithmetic_errc error; };
    _Bool failed;
  } r = catch(safe_divide(i, j));
  if(!r.failed)
    return r.value;
  if(r.error == divide_by_zero)
    return 0;
  if(r.error == integer_divide_overflows)
    return i / j; // ignore
  if(r.error == not_integer_division)
    return INT_MIN;
}
```

Summary

- One possible implementation of P0709
- Solves a few very long standing problems in C and POSIX at once
- Finally enables C code to call C++ code without exception translation wrappers!
  - Which means Rust, Python etc also can call C++ code directly without wrappers!
  - Also C++ can send exceptions to/from C!

Achieving the future today

C++ 11 <system_error>

C++ 11 `<system_error>`

- Probably the most commonly used STL header nobody has heard of
  - Provides the “advanced” error and exception infrastructure
  - Makes up ~20% of the tokens of many other STL headers e.g. `<array>`, `<complex>`, `<optional>`
- For deterministic disappointment, we only care about a subset …

C++ 11 `<system_error>`

- `std::error_code`
  - Integer + reference to explanatory category
- `std::errc`
  - `enum` of POSIX's common causes of failure
- `std::generic_category()`
  - Category for `std::errc`
- `std::system_category()`
  - Category for host system causes of failure
- `std::system_error()`
  - Exception type for throwing a `std::error_code`

```
std::error_code write(const char *buffer, size_t bytes) {
  do {
    ssize_t thiswrite = ::write(fd, buffer, bytes);      // disappoint?
    if(thiswrite >= 0) { buffer += thiswrite; bytes -= thiswrite; }
    else if(EAGAIN != errno) {           // Handle this locally (retry)
      std::error_code ec(errno, std::system_category());
      // Anticipated disappointment (part of control flow)
      if(ENOSPC == errno || EACCES == errno)
        return ec;
      // Unanticipated disappointment (abort and unwind stack)
      throw std::system_error(ec);
    }
  } while(bytes > 0);
  return {};  // default error code has convention of "no error here"
}
```

P1028: SG14 status_code and standard error object for P0709 Zero-overhead deterministic exceptions

by Niall (me) and SG14

https://wg21.link/P1028

https://ned14.github.io/status-code/

P1028 SG14 status_code

- Solves a long list of minor issues with `<system_error>` (see https://wg21.link/P0824)
  - As have become apparent only in hindsight
- Much nicer codegen than `<system_error>`
- Doesn't drag in most of the STL as includes like `<system_error>`
- Exceptions-disabled friendly

P1028 SG14 status_code

- Implements a proposed `std::error` for P0709 *Zero overhead deterministic exceptions* which is built on by P1095 *Zero overhead deterministic failure*
- Works in any C++ 11 compiler
  - >= GCC 5, >= clang 3.3, >= VS2015
- But NOTE that though approved unanimously by SG14, has not been judged by LEWG yet!

(Boost.) Outcome

by Niall (me)

https://ned14.github.io/outcome/

(Boost.) Outcome

- First new vocabulary library in Boost in many years!
- Only a year and a complete rewrite to get past Boost peer review!
- Probably consumed about 3,500 hours of my time over four years, tens of thousands of hours if including all effort invested by everybody

(Boost.) Outcome

- Lets you set per-namespace rules about local deterministic error handling
  - How and when local failure ought to be converted to exception throws
  - How local error handling ought to interact with third party or unknown local error handling
  - How payload ought to be lazily/eagerly converted when transitioning from this local error handling to other forms of error handling

(Boost.) Outcome

- Can completely substitute for C++ exceptions in a library or executable
  - Is deterministic
  - Is very lightweight, both at compile and runtime
  - Works well over arbitrary, unknown, third party libraries each with their own custom local implementations
  - Works fine with C++ exceptions globally disabled
  - Looks very like Rust/Swift/Go error handling

(Boost.) Outcome

- Unsurprisingly it is essentially a library implementation of *P1095R0/N2289: Zero overhead deterministic failure - A unified mechanism for C and C++*
  - C++ 14 minimum, C++ 20 preferred
  - >= clang 4.0.1, >= GCC 6.3, >= VS2017
- But can work with `std::error_code`, SG14 `status_code`, Boost, or your custom type

Without Outcome

```
int open_file(const std::filesystem::path &p,
              std::error_code &ec) noexcept {
  if(p.empty()) {
    ec = make_error_code(std::errc::invalid_argument);
    return -1;
  }
  ec.clear();  // surprisingly easy to forget to do
  int fd = ::open(p.c_str(), O_RDONLY);
  if(fd >= 0)
    return fd;
  ec = { errno, std::system_category() };
  return -1;
}
```

```
std::error_code ec;
int fd = open_file(path, ec);
if(-1 == fd)  // lots of people incorrectly write if(ec) here
{
  std::cerr << "Failed to open path due to "
            << ec.message() << std::endl;
  abort();
}
ssize_t bytesread = ::read(fd, buffer, bytes);
```

With Outcome

```cpp
result<int> open_file(const std::filesystem::path &p) noexcept
{
  if(p.empty())
    return std::errc::invalid_argument;

  int fd = ::open(p.c_str(), O_RDONLY);
  if(fd >= 0)
    return fd;
  return { errno, std::system_category() };
}
```

```
auto _fd = open_file(path);
if(!_fd)
{
  std::cerr << "Failed to open path due to "
            << _fd.error().message() << std::endl;
  abort();
}
int fd = _fd.value();
ssize_t bytesread = ::read(fd, buffer, bytes);
```

```cpp
// If it failed, throw its .error() as a std::system_error
int fd = open_file(path).value();
ssize_t bytesread = ::read(fd, buffer, bytes);
```

Thank you

And let the questions begin!

https://ned14.github.io/outcome/

https://www.linkedin.com/in/nialldouglas/

Available January 2019, >= 90% REMOTE only