Implement Standard Library

Design Decisions, Optimisations and Testing in Implementing Libc++

HUI XIE

Cppcon
The C++ Conference
2025
September 13

What is libc++

- A Standard Library Implementation
- Part of LLVM Project
- Ships with `clang`

Contents

- Various Optimisations in the Library
  - `std::expected`
  - `stop_token`
  - `ranges::for_each`, `ranges::copy`
  - `flat_map`
- Testing

About Me

- Hui Xie (GitHub @huixie90)
- Software Developer @Qube Research & Technologies
- libc++ contributor
- BSI (WG21 UK National Body) member
- WG21 (C++ Standard Committee) member
- Active in SG9 (Ranges Study Group)
- Write C++ Proposals

std::expected

```cpp
std::expected<MyData, MyError> compute();

int main() {
  std::expected<MyData, MyError> res = compute();

  if (res.has_value()) {
    // some code that uses res.value();
  } else {
    // some code that handles res.error();
  }
}
```

std::expected

```cpp
class Foo {
    int i;
    char c;
    bool b;
};

enum class ErrCode : int { Err1, Err2, Err3 };
```

```cpp
static_assert(sizeof(Foo) == 8); // on most implementations
```

```cpp
static_assert(sizeof(std::expected<Foo, ErrCode>) == ?);
```

`std::expected` in libstdc++

```cpp
// gcc libstdc++
template <class Val, class Err>
class expected {
    union {
        Val val_;
        Err err_;
    };
    bool has_val_;
};
```

```cpp
static_assert(sizeof(std::expected<Foo, ErrCode>) == ?);
```

std::expected in libstdc++

```
// gcc libstdc++
static_assert(sizeof(std::expected<Foo, ErrCode>) == 12);
```

```
int3 | int2 | int1 | int0 | char | bool | xxxx | xxxx | has_val | xxxx | xxxx | xxxx
<-------------- Foo Data --------------->
                                        <-  Foo pad. ->
<--------------- std::expected<Foo, ErrCode> Data -------------->
                                                                <-  expected pad. ->
<-------------------------- std::expected<Foo, ErrCode> --------------------------->
```

`std::expected` in libc++

```
// clang libc++  simplified version
template <class Val, class Err>
class expected {
    union U {
        [[no_unique_address]] Val val_;
        [[no_unique_address]] Err err_;
    };
    [[no_unique_address]] U u_;
    bool has_val_;
};
```

```
static_assert(sizeof(std::expected<Foo, ErrCode>) == ?);
```

The attribute-token no_unique_address specifies that a non-static data member is a potentially-overlapping subobject.

std::expected in libc++

```
int3 | int2 | int1 | int0 | char | bool | has_val | xxxx
<-------------- Foo Data --------------->
                                        <--  Foo pad. -->
<----- std::expected<Foo, ErrCode> Data ---------->
                                                  <ex pd>
<------------ std::expected<Foo, ErrCode> -------------->
```

```
// clang libc++
static_assert(sizeof(std::expected<Foo, ErrCode>) == 8);
```

- What are the benefits?

`sizeof(std::expected)` Smaller, Better ?

- Smaller Memory Footprint
- Better Cache Locality
- `std::expected` is likely to be used as a return type

std::expected return Type

```
std::expected<Foo, ErrCode> compute() { return Foo{}; }
```

```
# gcc libstdc++
compute():
        mov     DWORD PTR [rsp-24], 0
        xor     eax, eax
        mov     BYTE PTR [rsp-24], 1
        mov     ecx, DWORD PTR [rsp-24]
        mov     rdx, rcx
        ret
```

```
# clang libc++
compute():
        movabs  rax, 281474976710656
        ret
```

Takeaway 1

- Reuse tail padding with `[[no_unique_address]]`
  - Including the padding of an empty type

A Nasty Bug

```
template <class Val, class Err>
class expected {
    union U {
        [[no_unique_address]] Val val_;
        [[no_unique_address]] Err err_;
    };
    [[no_unique_address]] U u_;
    bool has_val_;

  public:
    expected(const expected& other)
        : has_val_(other.has_val_) {
        if (has_val_) {
          std::construct_at(std::addressof(u_.val_), other.u_.val_);
        } else {
          std::construct_at(std::addressof(u_.err_), other.u_.err_);
        }
    }
};
```

A Nasty Bug

```
int main() {
    std::expected<Foo, int> e1(Foo{});
    std::expected e2(e1);
    assert(e2.has_value());
}
```

```
output.s: /app/example.cpp:15: int main(): Assertion `e2.has_value()' failed.
Program terminated with signal: SIGSEGV
```

Zero-Initialisation

To zero-initialize an object or reference of type T means: … if T is a (possibly cv-qualified) non-union class type, its padding bits ([basic.types.general]) are initialized to zero bits and …

A Nasty Bug

```
template <class Val, class Err>
class expected {
    union U {
        [[no_unique_address]] Val val_;
        [[no_unique_address]] Err err_;
    };
    [[no_unique_address]] U u_;
    bool has_val_;

  public:
    expected(const expected& other)
        : has_val_(other.has_val_) {
        if (has_val_) {
          std::construct_at(std::addressof(u_.val_), other.u_.val_);
        } else {
          std::construct_at(std::addressof(u_.err_), other.u_.err_);
        }
    }
};
```

Takeaway 2

- Don't mix `[[no_unique_address]]` with manual lifetime management (`union`, `construct_at`, `placement-new`).

stop_source, stop_token, stop_callback

```cpp
std::stop_source stop_src;

// Thread 1
std::stop_token token = stop_src.get_token();
while(!token.stop_requested()) {
  // do some work
}

// Thread 2
std::stop_token token = stop_src.get_token();
std::stop_callback cb(token, [] { /*clean up*/});

// Thread 3
std::stop_token token = stop_src.get_token();
std::stop_callback cb(token, [] { /* do stuff */ });

// Thread 4
stop_src.request_stop();
```

Under The Hood

```
class stop_token {
  std::shared_ptr<__stop_state> state_;
};

class stop_source {
  std::shared_ptr<__stop_state> state_;
};

template <class Callback>
class stop_callback {
  [[no_unique_address]] Callback callback_;
  std::shared_ptr<__stop_state> state_;
};
```

But What About The Shared State?

- `stop_requested()`: needs a flag to hold whether a stop was requested
- `stop_possible()`: needs to count how many `stop_sources` exist
- `stop_callback`: needs to store a list of refs to all the `stop_callbacks`
- `stop_callback`: needs to synchronise the list of callbacks
  - major requirement is that the whole thing is `noexcept`, ruling out `mutex`
- The state needs a ref count to manage its lifetime

A Naive Implementation

```
class __stop_state {
  std::atomic<bool> stop_requested_;
  std::atomic<unsigned> stop_source_count_; // for stop_possible()
  std::list<stop_callback*> stop_callbacks_;
  std::mutex list_mutex_;
};
```

```
static_assert(sizeof(__stop_state) == 72);
```

```
static_assert(sizeof(stop_token) == 16);
```

libc++'s Implementation

```
class __stop_state {
  // The "callback list locked" bit implements a 1-bit lock to guard
  // operations on the callback list
  //
  //       31 - 2          |  1                   |    0           |
  //  stop_source counter  | callback list locked | stop_requested |
  atomic<uint32_t> state_ = 0;

  // Reference count for stop_token + stop_callback + stop_source
  atomic<uint32_t> ref_count_ = 0;

  // Lightweight intrusive non-owning list of callbacks
  // Only stores a pointer to the root node
  __intrusive_list_view<stop_callback_base> callback_list_;
};
```

```
static_assert(sizeof(__stop_state) == 16);
```

```
static_assert(sizeof(stop_token) == 8);
```

Takeaway 3

- Sometimes we can reuse unused bits to save space
- Look for existing padding bytes for free storage

Segmented Iterators `for_each`

```
std::deque<int> d = ...

// 1
for (int& i : d) {
    i = std::clamp(i, 200, 500);
}

// 2
std::ranges::for_each(d, [](int& i) {
    i = std::clamp(i, 200, 500);
});
```

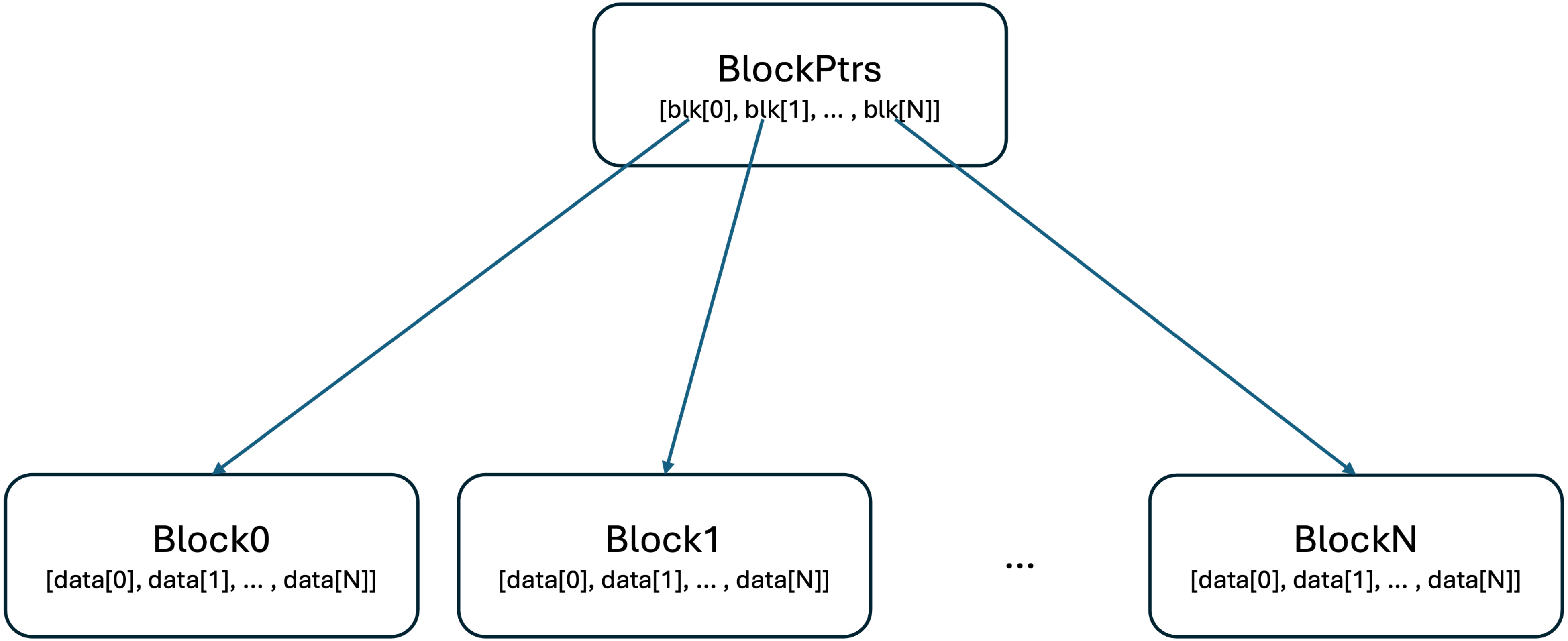
Benchmarking Iteration

```
Benchmark                                        for loop      for_each          speed up

[for_loop vs. for_each]/32                        12.5 ns       3.69 ns             3.4x
[for_loop vs. for_each]/8192                      2973 ns        259 ns            11.5x
[for_loop vs. for_each]/65536                    24221 ns       3327 ns             7.3x


clang20 -O3 cpu: M4 MAX 16 cores, memory: 48GB
```

std::deque


BlockPtrs
[blk[0], blk[1], ... , blk[N]]
Block0
[data[0], data[1], ... , data[N]]
Block1
[data[0], data[1], ... , data[N]]
...
BlockN
[data[0], data[1], ... , data[N]]

std::deque for Loop

```
deque_iterator begin = d.begin();
deque_iterator end = d.end();
for ( ; begin != end; ++begin ) {
  int& i = *begin;
  i = std::clamp(i, 200, 500);
}
```

```
deque_iterator& operator++() {
  if (++elem_ptr_ - *block_iter_ == block_size) {
    ++block_iter_;
    elem_ptr_ = *block_iter_;
  }
  return *this;
}
```

Segmented Iterators Concept

- Segmented Iterators and Hierarchical Algorithms, Matt Austern
- We introduced this `Segmented Iterator` concept into libc++
- Iterators over ranges of sub-ranges
- Allows algorithms to operate over these multi-level iterators natively

for_each_segment

```
template <class SegmentedIterator, class Func>
void __for_each_segment(SegmentedIterator first, SegmentedIterator last, Func func) {
  using Traits = SegmentedIteratorTraits<SegmentedIterator>;
  auto seg_first = Traits::segment(first);
  auto seg_last  = Traits::segment(last);
  // some code handles the first segment ...
  // iterate over the segments
  while (seg_first != seg_last) {
    func(Traits::begin(seg_first), Traits::end(seg_first));
    ++seg_first;
  }
  // some code handles the last segment ...
}
```

```
template <class SegmentedIterator, class Func>
  requires is_segmented_iterator<SegmentedIterator>
void for_each(SegmentedIterator first, SegmentedIterator last, Func func) {
  std::__for_each_segment(first, last, [&](auto inner_first, auto inner_last) {
    std::for_each(inner_first, inner_last, func);
  });
}
```

Code Generation Comparison

for loop

ranges::for_each

x86-64 clang (trunk) -O3 -stdlib=libc++ -std=c++26

```
        cmp     rcx, rax
        je      .LBB0_6
BB0_3:
        mov     r8d, dword ptr [rcx]
        cmp     r8d, 201
        cmovl   r8d, esi
        cmp     r8d, 500
        cmovge  r8d, edi
        mov     dword ptr [rcx], r8d
        add     rcx, 4
        mov     r8, rcx
        sub     r8, qword ptr [rdx]
        cmp     r8, 4096
        jne     .LBB0_5
        mov     rcx, qword ptr [rdx +
        add     rdx, 8
        jmp     .LBB0_5
BB0_6:
        ret
```

x86-64 clang (trunk) -O3 -stdlib=libc++ -std=c++26

```
BB0_10:
        movdqu  xmm2, xmmword ptr [rc
        movdqu  xmm3, xmmword ptr [rc
        movdqa  xmm4, xmm2
        pcmpgtd xmm4, xmm0
        pand    xmm2, xmm4
        pandn   xmm4, xmm0
        por     xmm4, xmm2
        movdqa  xmm2, xmm3
        pcmpgtd xmm2, xmm0
        pand    xmm3, xmm2
        pandn   xmm2, xmm0
        por     xmm2, xmm3
        movdqa  xmm3, xmm1
        pcmpgtd xmm3, xmm4
        pand    xmm4, xmm3
        pandn   xmm3, xmm1
        por     xmm3, xmm4
```

Optimizing ranges::copy

```
std::vector<std::vector<int>> v = ...;
std::vector<int> out;

// 1
out.reserve(total_size);
for (const auto& inner : v) {
  for (int i: inner) {
    out.push_back(i);
  }
}

// 2
out.resize(total_size);
std::ranges::copy(v | std::views::join, out.begin());
```

Benchmarking ranges::copy

```
Benchmark                              for loop        copy        speed up

[for_loop vs. copy]/32                   490 ns      322 ns           1.5x
[for_loop vs. copy]/8192              160632 ns    63372 ns           2.5x
[for_loop vs. copy]/65536            1066403 ns   208083 ns           5.1x
```

`join_view::iterator` is a Segmented Iterator

```
template <class Iter, class OutIter>
  requires is_segmented_iterator<Iter>
pair<Iter, OutIter> __copy(Iter first, Iter last, OutIter result) {
  std::__for_each_segment(first, last, [&](auto inner_first, auto inner_last) {
      result = std::__copy(inner_first, inner_last, std::move(result)).second;
  });
  return std::make_pair(last, std::move(result));
}
```

```
template <class In, class Out>
  requires can_lower_copy_assignment_to_memmove<In, Out>
pair<In*, Out*> __copy(In* first, In* last, Out* result) {
  std::memmove(result, first, last - first);
  return std::make_pair(last, result + n);
}
```

Takeaway 4

- We constantly add optimisations to algorithms. Please use them
- We use general concepts to capture optimisation opportunities
- Optimisations are often generic and can compose with each other

flat_map Insertion

```
// flat_map<int, double>
class flat_map {
  std::vector<int> keys_; // always sorted
  std::vector<double> values_;
  [[no_unique_address]] std::less<int> compare_;
};
```

```
std::flat_map<int, double> m1 = ...;
std::flat_map<int, double> m2 = ...;

// Insert the elements of m2 into m1
for (const auto& [key, val] : m2) {
  m1.emplace(key, val);
}
```

What is the time complexity?

flat_map::insert_range

```
template<container-compatible-range<value_type> R>
constexpr void insert_range(R&& rg);
```

- Complexity: N `+ MlogM`, where N is `size()` before the operation and M is `ranges::distance(rg)`.

```
std::flat_map<int, double> m1 = ...;
std::flat_map<int, double> m2 = ...;

// Insert the elements of m2 into m1
m1.insert_range(m2);
```

P3567 `insert_range(sorted_unique, rg)`

```
template<container-compatible-range<value_type> R>
  void insert_range(sorted_unique_t, R&& rg);
```

- Complexity: Linear in N, where N is `size()` after the operation.

```
std::flat_map<int, double> m1 = ...;
std::flat_map<int, double> m2 = ...;

// Insert the elements of m2 into m1
m1.insert_range(std::sorted_unique, m2);
```

flat_map::insert_range Implementation

```
template<container-compatible-range<value_type> R>
constexpr void insert_range(R&& rg) {

  __append(ranges::begin(rg), ranges::end(rg)); // O(M)

  auto zv = ranges::views::zip(keys_, values_);
  ranges::sort(zv.begin() + old_size, zv.end()); // O(MLogM)

  ranges::inplace_merge(zv.begin(), zv.begin() + old_size, zv.end()); // O(M+N)

  auto dup_start = ranges::unique(zv).begin(); // O(M+N)
  __erase(dup_start); // O(M+N)
}
```

How to __append ?

```
template <class InputIterator, class Sentinel>
void __append(InputIterator first, Sentinel last) {
  for (; first != last; ++first) {
    std::pair<Key, Val> kv = *first;
    keys_.insert(keys_.end(), std::move(kv.first));
    values_.insert(values_.end(), std::move(kv.second));
  }
}
```

- This misses existing optimisations in `vector::insert(pos, first, last)`
- But we are given a range of “pairs”, not two ranges
- Can we reuse these optimizations in some cases?

Introducing a Concept `product_iterator`

- iterators that *aggregate* multiple underlying iterators
- `flat_map::iterator` is `product_iterator`

```
template <class Iter>
  requires is_product_iterator_of_size<Iter, 2>
void __append(Iter first, Iter last)
{
  using Traits = product_iterator_traits<Iter>;

  keys_.insert(keys_.end(),
      Traits::template get_iterator<0>(first),
      Traits::template get_iterator<0>(last));

  values_.insert(values_.end(),
      Traits::template get_iterator<1>(first),
      Traits::template get_iterator<1>(last));
}
```

Benchmarking `insert_range`

```
Benchmark                       insert_pair  product_iterator        speed up

[insert_range]/32                   149 ns             74 ns            2.0x
[insert_range]/8192               26682 ns           2995 ns            8.9x
[insert_range]/65536             226235 ns          27844 ns            8.1x
```

Are there any other `product_iterator`?

```
std::flat_map<int, double> m = ...;
std::vector<int> newKeys = ...;
std::vector<double> newValues = ...;

// Insert newKeys and newValues into m
```

```
m.insert_range(std::views::zip(newKeys, newValues));
```

`zip_view::iterator` is also a `product_iterator`

Takeaway 5

- Use the most precise API for what you're trying to achieve
  - `insert_range` instead of `insert` in a loop
  - Use `sorted_unique` if the inputs are already sorted
- Use library facilities (e.g `views::zip`) to benefit from concept-based optimisations

Testing in libc++

- We test almost every single word in the standard

What Shall We Test?

```
constexpr expected(expected&& rhs) noexcept(see below);
```

- Constraints:
  - `is_move_constructible_v<T>` is `true` and
  - `is_move_constructible_v<E>` is `true`.
- Effects: If `rhs.has_value()` is `true`, direct-non-list-initializes val with `std::move(*rhs)`. Otherwise, direct-non-list-initializes `unex` with `std::move(rhs.error())`.
- Postconditions: `rhs.has_value()` is unchanged; `rhs.has_value() == this->has_value()` is true.
- Throws: Any exception thrown by the initialization of `val` or `unex`.
- Remarks: The exception specification is equivalent to `is_nothrow_move_constructible_v<T> && is_nothrow_move_constructible_v<E>`.
- This constructor is trivial if
  - `is_trivially_move_constructible_v<T>` is `true` and
  - `is_trivially_move_constructible_v<E>` is `true`.

Testing `constexpr`

```
constexpr bool test() {
  // Test point 1
  {
    std::expected<MoveOnly, int> e1 = ...;
    std::expected<MoveOnly, int> e2 = std::move(e1);
    assert(e2.has_value());
    assert(e2.value() == ... );
  }
  // more test points

  return true;
}

int main() {
  test();
  static_assert(test());
}
```

- Share compile time and run time tests

Testing Constraints

```
constexpr expected(expected&& rhs) noexcept(see below);
```

- Constraints: `is_move_constructible_v<T>` is `true` and `is_move_constructible_v<E>` is `true`.

- Constraints translate to `requires`

```
struct Foo { Foo(Foo&&) = delete;};

static_assert(std::is_move_constructible_v<std::expected<int,int>>);
static_assert(!std::is_move_constructible_v<std::expected<Foo,int>>);
static_assert(!std::is_move_constructible_v<std::expected<int,Foo>>);
static_assert(!std::is_move_constructible_v<std::expected<Foo,Foo>>);
```

Testing Mandates

```
template<class U = remove_cv_t<T>> constexpr T value_or(U&& v) const &;
```

- Mandates: `is_copy_constructible_v<T>` is `true` and `is_convertible_v<U, T>` is true.
- Mandates translate to `static_assert`
- Test that usage that violates the mandate should not compile

```
const std::expected<NonCopyable, int> f1{5};
f1.value_or(5); // expected-note{{in instantiation of function template ...}}
// expected-error-re@*:* {{static assertion failed {{.*}}value_type has to be copy ...}}
```

```
-Xclang -verify
```

Testing {Hardened} Preconditions

```
constexpr const T* operator->() const noexcept;
constexpr T* operator->() noexcept;
```

- Hardened preconditions: `has_value()` is `true`.
- Preconditions/Hardened preconditions translate to `assert`/`contract_assert`/other assert macros

```
std::expected<int, int> e{std::unexpect, 5};
TEST_LIBCPP_ASSERT_FAILURE(e.operator->(),
    "expected::operator-> requires the expected to contain a value");
```

- `TEST_LIBCPP_ASSERT_FAILURE` installs assertion handler, forks the process and matches the child process `stderr` with the expected errors

Takeaway 6

- Share the same test code for `constexpr` and runtime
- Use negative tests to ensure that things fail **as expected**
- `-Xclang -verify` is very useful to test `static_asserts`

Contribute to libc++

- Getting Started
- Github Issues

QRT is Hiring

- https://www.qube-rt.com/careers/

If You Do Mix `[[no_unique_address]]` with `union`, `std::construct_at`, or placement `new`

- Be ready for bugs

Dude, Where is `my_char` ?

```
struct MyStruct {
  [[no_unique_address]] std::expected<Foo, ErrCode> my_foo;
  char my_char;
};

MyStruct s{.my_foo = Foo{}, .my_char = 'x'};

s.my_foo.emplace(Foo{});
```

```
// Assertion failure!
assert(s.my_char == 'x');
```

Wait, is `my_char` in the Padding?

```
template <class Val, class Err>
class expected {
    union U {
        [[no_unique_address]] Val val_;
        [[no_unique_address]] Err err_;
    };
    [[no_unique_address]] U u_;
    bool has_val_;
};
```

```
int3 | int2 | int1 | int0 | char | bool | has_val | my_char
<-------------- Foo Data --------------->
                                        <---  Foo pad. --->
<----- std::expected<Foo, ErrCode> Data ---------->
                                                  <ex pad.>
<------------- std::expected<Foo, ErrCode> --------------->
<----------------------- MyStruct ------------------------>
```

```
s.my_foo.emplace(Foo{}); // It calls construct_at and clears the padding!
```

Stop Users from Reusing expected Paddings

```
template <class Val, class Err>
class expected {
  struct repr {
    union U {
        [[no_unique_address]] Val val_;
        [[no_unique_address]] Err err_;
    };
    [[no_unique_address]] U u_;
    bool has_val_;
  };

  repr repr_; // No [[no_unique_address]] here
};
```

- `[[no_unique_address]]` is not transitive

It Works

```cpp
struct MyStruct {
  [[no_unique_address]] std::expected<Foo, ErrCode> my_foo;
  char my_char;
};
```

```
int3 | int2 | int1 | int0 | char | bool | has_val | xxxx | my_char | xxxx | xxxx | xxxx
<-------------- Foo Data --------------->
                                        <---  Foo pad. --->
<------------------------ repr Data -------------->
                                                 <repr pad.>
<------------ std::expected<Foo, ErrCode> Data ----------->
<------------ std::expected<Foo, ErrCode> ---------------->
<----------------------- MyStruct Data ---------------------------->
                                                                   <-MyStruct padding ->
<-------------------------------------- MyStruct -------------------------------------->
```

But Can We do Better?

```
struct MyStruct {
  [[no_unique_address]] std::expected<int, ErrCode> my_int;
  char my_char;
};
```

```
int3 | int2 | int1 | int0 | has_val | xxxx | xxxx | xxxx | my_char | xxxx | xxxx | xxxx
<---------------- repr Data ---------->
                                    <--- repr padding --->
<------------- std::expected<int, ErrCode> Data ---------->
<------------- std::expected<int, ErrCode> ---------------->
<------------------------ MyStruct Data ----------------------------->
                                                                <-MyStruct padding ->
<--------------------------------------- MyStruct ---------------------------------------->
```

```
s.my_int.emplace(42); // construct_at(int*, int) is very safe! int has no padding
```

Final Fix

```
template <class Val, class Err>
struct repr {
    union U {
        [[no_unique_address]] Val val_;
        [[no_unique_address]] Err err_;
    };
    [[no_unique_address]] U u_;
    bool has_val_;
};

template <class Val, class Err>
struct expected_base {
    repr<Val, Err> repr_; // no [[no_unique_address]]
};
template <class Val, class Err> requires bool_is_not_in_padding
struct expected_base {
    [[no_unique_address]] repr<Val, Err> repr_;
};

template <class Val, class Err>
class expected : expected_base<Val, Err> {};
```