



Beyond C++17

PART II

Mateusz Pusz
May 6, 2019

ひ款す 国出のシ品 政羅
ト社明 をに義と 字印 ひ款す 国出のシ
印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

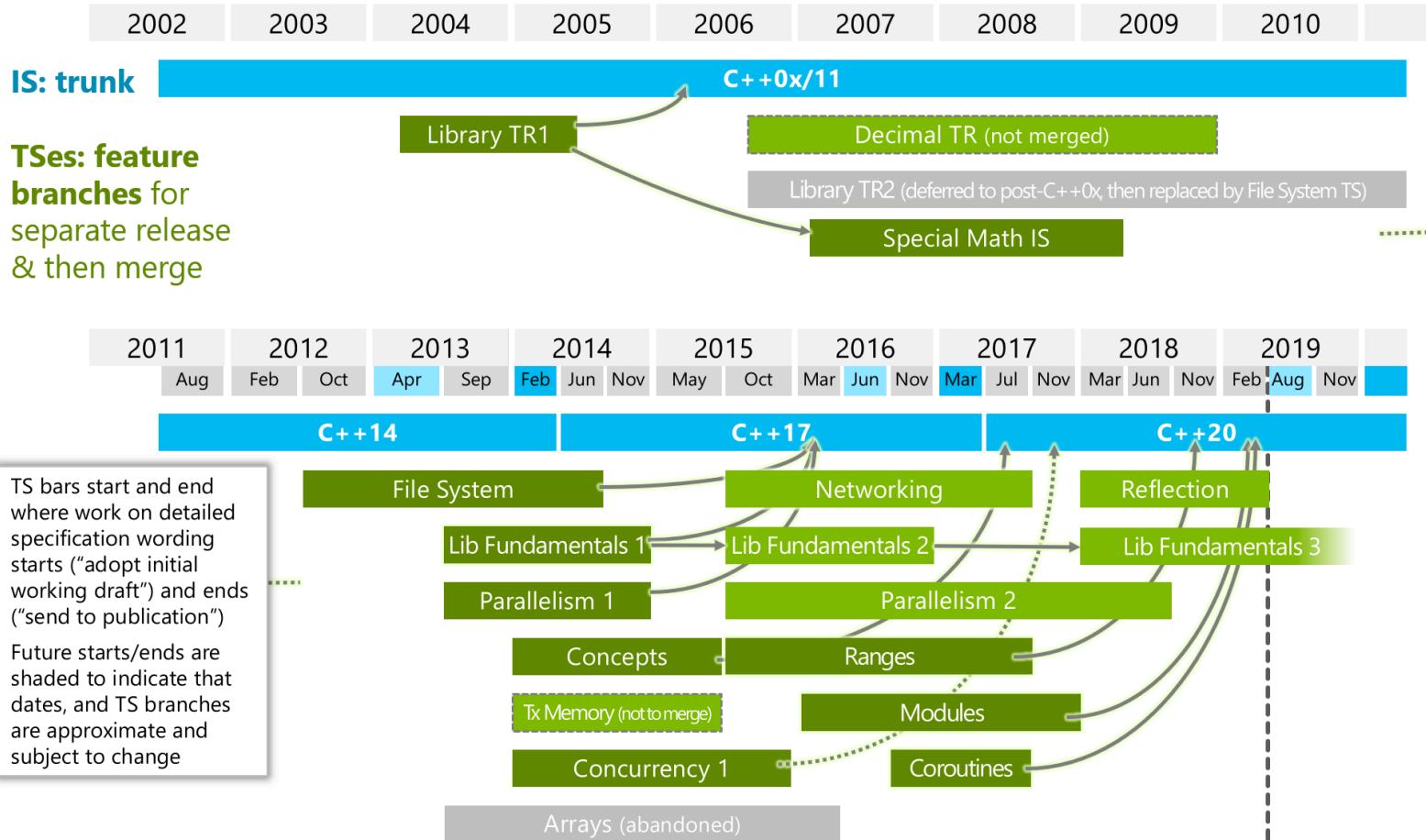
印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

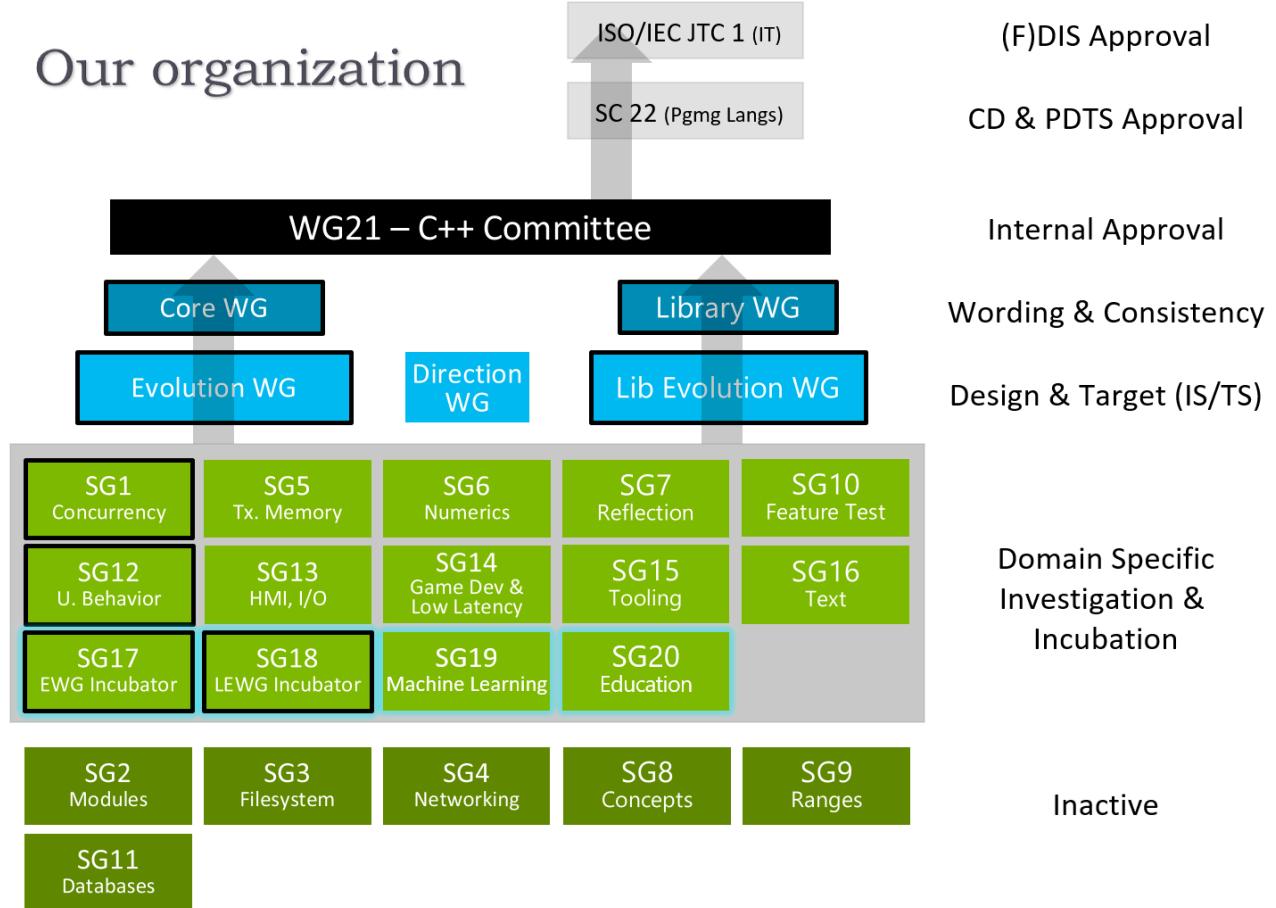
印 ひ款す 国出のシ品 政羅
シ品 政羅は 国出のシ品 政羅は 国出のシ

C++ Timeline



ISO C++ Committee structure

Our organization



Disclaimer

- Last year was *really productive* in the ISO C++ Committee
- We do not have time to iterate over all of the changes accepted for C++20
 - *less important changes are skipped*
- We even do not have enough time to describe the most important changes in detail
 - refer to talks dedicated to specific subjects for more details
 - study the proposal paper

Finding a paper - <https://wg21.link>

- Usage info
 - wg21.link
- Get paper
 - wg21.link/nXXXX
 - wg21.link/pXXXX - latest version (e.g. wg21.link/p0919)
 - wg21.link/pXXXXrX
- Get working draft
 - wg21.link/std
 - wg21.link/std11
 - wg21.link/std14
 - wg21.link/std17

BEYOND C++17: PART I

MAJOR C++20 FEATURES RECAP

Constraints and concepts

```
template<typename T>
concept EqualityComparable = requires(T a, T b) {
    a == b; requires Boolean<decltype(a == b)>; // simplified definition
};
```

Constraints and concepts

```
template<typename T>
concept EqualityComparable = requires(T a, T b) {
    a == b; requires Boolean<decltype(a == b)>; // simplified definition
};
```

```
template<typename T>
    requires EqualityComparable<T>
void f(T&& t)
{
    if(t == other) { /* ... */ }
}
```

Constraints and concepts

```
template<typename T>
concept EqualityComparable = requires(T a, T b) {
    a == b; requires Boolean<decltype(a == b)>; // simplified definition
};
```

```
template<typename T>
    requires EqualityComparable<T>
void f(T&& t)
{
    if(t == other) { /* ... */ }
}
```

```
void foo()
{
    f("abc"s); // OK
    std::mutex mtx;
    std::unique_lock<std::mutex> lock{mtx};
    f(mtx); // Error: not EqualityComparable
}
```

P0515 P0768 P0905 Consistent comparison

C++17

```
class P {
    int x;
    int y;
public:
    friend bool operator==(const P& a, const P& b)
    { return a.x==b.x && a.y==b.y; }
    friend bool operator< (const P& a, const P& b)
    { return a.x<b.x || (a.x==b.x && a.y<b.y); }
    friend bool operator!=(const P& a, const P& b)
    { return !(a==b); }
    friend bool operator<=(const P& a, const P& b)
    { return !(b<a); }
    friend bool operator> (const P& a, const P& b)
    { return b<a; }
    friend bool operator>=(const P& a, const P& b)
    { return !(a<b); }
    // ... non-comparison functions ...
};
```

P0515 P0768 P0905 Consistent comparison

C++17

```
class P {
    int x;
    int y;
public:
    friend bool operator==(const P& a, const P& b)
    { return a.x==b.x && a.y==b.y; }
    friend bool operator< (const P& a, const P& b)
    { return a.x<b.x || (a.x==b.x && a.y<b.y); }
    friend bool operator!=(const P& a, const P& b)
    { return !(a==b); }
    friend bool operator<=(const P& a, const P& b)
    { return !(b<a); }
    friend bool operator> (const P& a, const P& b)
    { return b<a; }
    friend bool operator>=(const P& a, const P& b)
    { return !(a<b); }
    // ... non-comparison functions ...
};
```

C++20

```
class P {
    int x;
    int y;
public:
    auto operator<=>(const P&) const = default;
    // ... non-comparison functions ...
};
```

- **a <=> b** returns an object that compares
 - **<0** if **a < b**
 - **>0** if **a > b**
 - **==0** if **a** and **b** are equal/equivalent
- *Memberwise* semantics by default
- Commonly known as a **spaceship** operator

P0515 P0768 P0905 Consistent comparison

```
class ci_string {
    std::string s;
public:
    // ...

    friend bool operator==(const ci_string& a, const ci_string& b) { return ci_compare(a.s.c_str(), b.s.c_str()) != 0; }
    friend bool operator< (const ci_string& a, const ci_string& b) { return ci_compare(a.s.c_str(), b.s.c_str()) < 0; }
    friend bool operator!=(const ci_string& a, const ci_string& b) { return !(a == b); }
    friend bool operator> (const ci_string& a, const ci_string& b) { return b < a; }
    friend bool operator>=(const ci_string& a, const ci_string& b) { return !(a < b); }
    friend bool operator<=(const ci_string& a, const ci_string& b) { return !(b < a); }

    friend bool operator==(const ci_string& a, const char* b) { return ci_compare(a.s.c_str(), b) != 0; }
    friend bool operator< (const ci_string& a, const char* b) { return ci_compare(a.s.c_str(), b) < 0; }
    friend bool operator!=(const ci_string& a, const char* b) { return !(a == b); }
    friend bool operator> (const ci_string& a, const char* b) { return b < a; }
    friend bool operator>=(const ci_string& a, const char* b) { return !(a < b); }
    friend bool operator<=(const ci_string& a, const char* b) { return !(b < a); }

    friend bool operator==(const char* a, const ci_string& b) { return ci_compare(a, b.s.c_str()) != 0; }
    friend bool operator< (const char* a, const ci_string& b) { return ci_compare(a, b.s.c_str()) < 0; }
    friend bool operator!=(const char* a, const ci_string& b) { return !(a == b); }
    friend bool operator> (const char* a, const ci_string& b) { return b < a; }
    friend bool operator>=(const char* a, const ci_string& b) { return !(a < b); }
    friend bool operator<=(const char* a, const ci_string& b) { return !(b < a); }
};
```

P0515 P0768 P0905 Consistent comparison

```
class ci_string {
    std::string s;
public:
    // ...

    std::weak_ordering operator<=>(const ci_string& b) const { return ci_compare(s.c_str(), b.s.c_str()); }
    std::weak_ordering operator<=>(const char* b) const { return ci_compare(s.c_str(), b); }
};
```

P0515 P0768 P0905 Consistent comparison

```
class ci_string {
    std::string s;
public:
    // ...

    std::weak_ordering operator<=>(const ci_string& b) const { return ci_compare(s.c_str(), b.s.c_str()); }
    std::weak_ordering operator<=>(const char* b) const { return ci_compare(s.c_str(), b); }
};
```

- <compare> header needed when user manually provides <=> implementation

TYPE RETURNED FROM OPERATOR<=>()	A<B SUPPORTED	A<B NOT SUPPORTED
a==b => f(a)==f(b)	std::strong_ordering	std::strong_equality
a==b => f(a)!=f(b)	std::weak_ordering	std::weak_equality

P0355 Extending chrono to Calendars and Time Zones

FEATURES

- Minimal extensions to `<chrono>` to support calendar and time zone libraries
- A proleptic Gregorian calendar (civil calendar)
- A time zone library based on the IANA Time Zone Database
- `strftime`-like formatting and parsing facilities with fully operational support for fractional seconds, time zone abbreviations, and UTC offsets
- Several `<chrono>` clocks for computing with leap seconds which is also supported by the IANA Time Zone Database

P0355 Extending chrono to Calendars and Time Zones

EXAMPLES

```
constexpr year_month_day ymd1{2016y, month{5}, day{29}};
constexpr auto ymd2 = 2016y/may/29d;
constexpr auto ymd3 = sun[5]/may/2016;
```

P0355 Extending chrono to Calendars and Time Zones

EXAMPLES

```
constexpr year_month_day ymd1{2016y, month{5}, day{29}};  
constexpr auto ymd2 = 2016y/may/29d;  
constexpr auto ymd3 = sun[5]/may/2016;
```

```
constexpr system_clock::time_point tp = sys_days{sun[5]/may/2016}; // Convert date to time_point  
static_assert(tp.time_since_epoch() == 1'464'480'000'000'000us);  
constexpr auto ymd = year_month_weekday{floor<days>(tp)};           // Convert time_point to date  
static_assert(ymd == sun[5]/may/2016);
```

P0355 Extending chrono to Calendars and Time Zones

EXAMPLES

```
constexpr year_month_day ymd1{2016y, month{5}, day{29}};  
constexpr auto ymd2 = 2016y/may/29d;  
constexpr auto ymd3 = sun[5]/may/2016;
```

```
constexpr system_clock::time_point tp = sys_days{sun[5]/may/2016}; // Convert date to time_point  
static_assert(tp.time_since_epoch() == 1'464'480'000'000'000us);  
constexpr auto ymd = year_month_weekday{floor<days>(tp)};           // Convert time_point to date  
static_assert(ymd == sun[5]/may/2016);
```

```
auto tp = sys_days{2016y/may/29d} + 7h + 30min + 6s + 153ms; // 2016-05-29 07:30:06.153 UTC  
zoned_time zt = {"Asia/Tokyo", tp};                                // 2016-05-29 16:30:06.153 JST  
std::cout << zt << '\n';
```

P0122

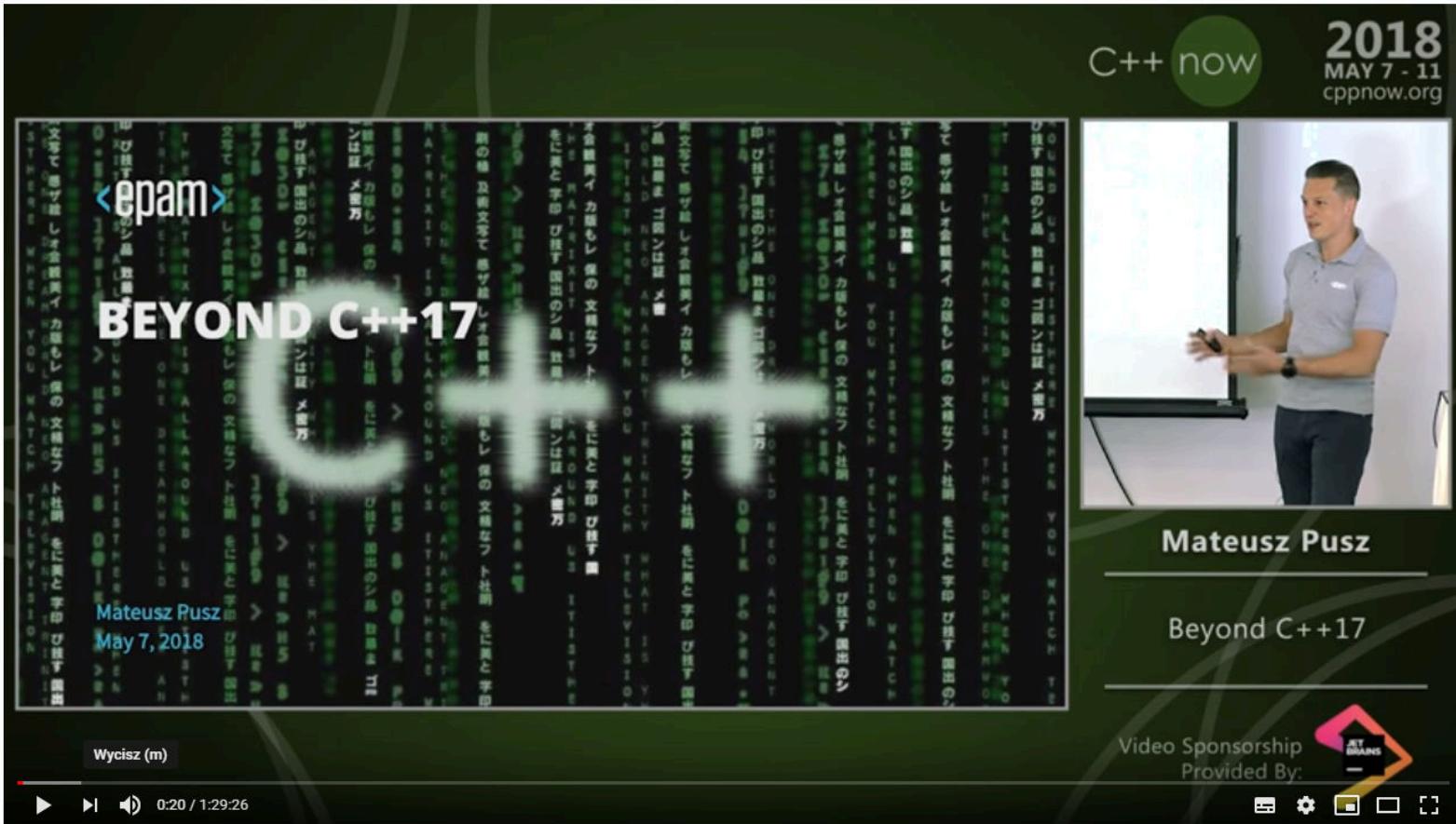
The **span** type is an abstraction that provides a view over a contiguous sequence of objects, the storage of which is owned by some other object.

The **span** type is an abstraction that provides a view over a contiguous sequence of objects, the storage of which is owned by some other object.

VIEW, NOT CONTAINER

- Simply a *view* over another object's contiguous storage – it *does not own* the elements that are accessible through its interface (similarly to `std::string_view`)
- Never performs *any free store allocations*

More details from Part I



C++Now 2018: Mateusz Pusz "Beyond C++17"

Let's vote

OPTION A

- Big and famous C++20 features
- More on Concepts and Constraints
- Ranges (quick overview)
- More on `<=>`
- Contracts
- Modules
- Coroutines (quick overview)

OPTION B

- Lots of less advertised good C++20 stuff
- Class Types in Non-Type Template Parameters
- Fixing aggregates
- `constexpr` all the things
- Immediate functions
- `explicit(bool)`
- More on `span`
- Changes to containers and algorithms
- Fixing `variant`
- `std::bit_cast()`
- More...

BEYOND C++17: PART II

OPTION A: BIG AND FAMOUS C++20 FEATURES

P1141 Yet another approach for constrained declarations

TYPE PARAMETERS

```
template<typename S>  
void foo();
```

NON-TYPE PARAMETERS

```
template<auto S>  
void foo();
```

P1141 Yet another approach for constrained declarations

TYPE PARAMETERS

```
template<typename S>  
void foo();
```

```
template<Concept S>  
void foo();
```

NON-TYPE PARAMETERS

```
template<auto S>  
void foo();
```

```
template<Concept auto S>  
void foo();
```

P1141 Yet another approach for constrained declarations

TYPE PARAMETERS

```
template<typename S>  
void foo();
```

```
template<Concept S>  
void foo();
```

NON-TYPE PARAMETERS

```
template<auto S>  
void foo();
```

```
template<Concept auto S>  
void foo();
```

- **template template** parameters are **not supported** in this short form

P1141 Yet another approach for constrained declarations

TYPE PARAMETERS

```
template<typename S>  
void foo();
```

```
template<Concept S>  
void foo();
```

NON-TYPE PARAMETERS

```
template<auto S>  
void foo();
```

```
template<Concept auto S>  
void foo();
```

- **template template** parameters are **not supported** in this short form
- **Concept** is restricted to be a concept that takes a **type parameter** or **type parameter pack**
 - **non-type** and **template template** concepts are **not supported** in this short form

P1141 Yet another approach for constrained declarations

FULL NOTATION

```
template<typename... T>
    requires Concept<T> && ... && true
void f(T...);
```

P1141 Yet another approach for constrained declarations

FULL NOTATION

```
template<typename... T>
    requires Concept<T> && ... && true
void f(T...);
```

SHORTHAND NOTATION

```
template<Concept... T>
void f(T...);
```

P1141 Yet another approach for constrained declarations

FULL NOTATION

```
template<typename... T>
    requires Concept<T> && ... && true
void f(T...);
```

SHORTHAND NOTATION

```
template<Concept... T>
void f(T...);
```

TERSE NOTATION

```
void f(Concept auto... T);
```

P1141 Yet another approach for constrained declarations

FULL NOTATION

```
template<typename... T>
    requires Concept<T, int> && ... && true
void f(T...);
```

SHORTHAND NOTATION

```
template<Concept<int>... T>
void f(T...);
```

TERSE NOTATION

```
void f(Concept<int> auto... T);
```

P1141 Yet another approach for constrained declarations

CONSISTENCE WITH C++14 LAMBDAS

```
[](auto a, auto& b, const auto& c, auto&& d) { /* ... */ };
```

```
void f1(auto a, auto& b, const auto& c, auto&& d) { /* ... */ }
```

P1141 Yet another approach for constrained declarations

CONSISTENCE WITH C++14 LAMBDAS

```
[](auto a, auto& b, const auto& c, auto&& d) { /* ... */ };
```

```
void f1(auto a, auto& b, const auto& c, auto&& d) { /* ... */ }
```

CONSTRAINING THE AUTO TYPE

```
[](Concept auto a, Concept auto& b, const Concept auto& c, Concept auto&& d) { /* ... */ };
```

```
void f2(Concept auto a, Concept auto& b, const Concept auto& c, Concept auto&& d) { /* ... */ }
```

P1141 Yet another approach for constrained declarations

CONSISTENCE WITH C++14 LAMBDAS

```
[](auto a, auto& b, const auto& c, auto&& d) { /* ... */ };
```

```
void f1(auto a, auto& b, const auto& c, auto&& d) { /* ... */ }
```

CONSTRAINING THE AUTO TYPE

```
[](Concept auto a, Concept auto& b, const Concept auto& c, Concept auto&& d) { /* ... */ };
```

```
void f2(Concept auto a, Concept auto& b, const Concept auto& c, Concept auto&& d) { /* ... */ }
```

The appearance of **auto** in a function parameter list tells us that we are dealing with a *function template*

P1141 Yet another approach for constrained declarations

RETURN TYPE

```
Concept auto f5(); // deduced and constrained return type; not a template function
```

P1141 Yet another approach for constrained declarations

RETURN TYPE

```
Concept auto f5(); // deduced and constrained return type; not a template function
```

VARIABLE TYPE

```
Concept auto x2 = f2();
```

P1141 Yet another approach for constrained declarations

RETURN TYPE

```
Concept auto f5(); // deduced and constrained return type; not a template function
```

VARIABLE TYPE

```
Concept auto x2 = f2();
```

NON-TYPE TEMPLATE PARAMETER

```
template<Concept auto N>
void f7();
```

P1141 Yet another approach for constrained declarations

RETURN TYPE

```
Concept auto f5(); // deduced and constrained return type; not a template function
```

VARIABLE TYPE

```
Concept auto x2 = f2();
```

NON-TYPE TEMPLATE PARAMETER

```
template<Concept auto N>
void f7();
```

NEW EXPRESSION

```
auto alloc_next() { return new Concept auto(this->next_val()); }
```

P1141 Yet another approach for constrained declarations

CONVERSION OPERATOR

```
struct X {  
    operator Concept auto() { /* ... */ }  
};
```

P1141 Yet another approach for constrained declarations

CONVERSION OPERATOR

```
struct X {  
    operator Concept auto() { /* ... */ }  
};
```

DECLTYPE(AUTO)

```
auto f() -> Concept decltype(auto);
```

```
Concept decltype(auto) x = f();
```

P1141 Yet another approach for constrained declarations

CONVERSION OPERATOR

```
struct X {  
    operator Concept auto() { /* ... */ }  
};
```

DECLTYPE(AUTO)

```
auto f() -> Concept decltype(auto);
```

```
Concept decltype(auto) x = f();
```

Exception: **auto** in structure bindings cannot be constrained (at least for now)

P1084 Today's return-type-requirements Are Insufficient

- After more than 3 years of experience in the development and use of concepts we see *issues with the original design*
- We have concepts notation to specify (via the trailing-return type notation `->`) that a constraint is *satisfied if an expression E is convertible to a type T*
- It is better to have constraints that are satisfied if *`decltype(E)` is exactly the type T*

P1084 Today's return-type-requirements Are Insufficient

- After more than 3 years of experience in the development and use of concepts we see *issues with the original design*
- We have concepts notation to specify (via the trailing-return type notation `->`) that a constraint is *satisfied if an expression E is convertible to a type T*
- It is better to have constraints that are satisfied if *decltype(E) is exactly the type T*

```
{ E } -> Concept<Args...>;
```

is equivalent to

```
E; requires Concept<decltype((E)), Args...>;
```

P1084 Today's return-type-requirements Are Insufficient

BEFORE

```
template<class T>
concept StrictTotallyOrdered =
    EqualityComparable<T> &&
    requires(const remove_reference_t<T>& a,
             const remove_reference_t<T>& b) {
        a < b; requires Boolean<decltype(a < b)>;
        a > b; requires Boolean<decltype(a > b)>;
        a <= b; requires Boolean<decltype(a <= b)>;
        a >= b; requires Boolean<decltype(a >= b)>;
    };
```

P1084 Today's return-type-requirements Are Insufficient

AFTER

```
template<class T>
concept StrictTotallyOrdered =
    EqualityComparable<T> &&
    requires(const remove_reference_t<T>& a,
             const remove_reference_t<T>& b) {
        { a < b } -> Boolean;
        { a > b } -> Boolean;
        { a <= b } -> Boolean;
        { a >= b } -> Boolean;
    };
```

P0898 Standard Library Concepts

CORE LANGUAGE CONCEPTS

```
template <class T, class U>           concept Same;
template <class Derived, class Base>  concept DerivedFrom;
template <class From, class To>        concept ConvertibleTo;
template <class T, class U>           concept CommonReference;
template <class T, class U>           concept Common;
template <class T>                   concept Integral;
template <class T>                   concept SignedIntegral;
template <class T>                   concept UnsignedIntegral;
template <class LHS, class RHS>      concept Assignable;
template <class T>                   concept Swappable;
template <class T, class U>           concept SwappableWith;
template <class T>                   concept Destructible;
template <class T, class... Args>     concept Constructible;
template <class T>                   concept DefaultConstructible;
template <class T>                   concept MoveConstructible;
template <class T>                   concept CopyConstructible;
```

P0898 Standard Library Concepts

COMPARISON CONCEPTS

```
template <class B>           concept Boolean;
template <class T>            concept EqualityComparable;
template <class T, class U>  concept EqualityComparableWith;
template <class T>            concept StrictTotallyOrdered;
template <class T, class U>  concept StrictTotallyOrderedWith;
```

P0898 Standard Library Concepts

OBJECT CONCEPTS

```
template <class T> concept Movable;
template <class T> concept Copyable;
template <class T> concept Semiregular;
template <class T> concept Regular;
```

P0898 Standard Library Concepts

CALLABLE CONCEPT

```
template <class F, class... Args>    concept Invocable;
template <class F, class... Args>    concept RegularInvocable;
template <class F, class... Args>    concept Predicate;
template <class R, class T, class U> concept Relation;
template <class R, class T, class U> concept StrictWeakOrder;
```

P0898 Standard Library Concepts

NUMERIC CONCEPT

```
template <class G> concept UniformRandomBitGenerator;
```

P0896 The One Ranges Proposal

- *Concepts* for iterators, sentinels, ranges, and views
- *New tools*
 - sentinels
 - a few new iterators
 - views
 - range adaptor objects
- *Customization points*

P0896 The One Ranges Proposal

- *Concepts* for iterators, sentinels, ranges, and views

- *New tools*

- sentinels
- a few new iterators
- views
- range adaptor objects

- *Customization points*

```
vector<int> ints{0, 1, 2, 3, 4, 5};

auto even = [](int i){ return 0 == i % 2; };
auto square = [](int i) { return i * i; };

for(int i : ints | view::filter(even) | view::transform(square))
    std::cout << i << ' '; // prints: 0 4 16
```

P0896 The One Ranges Proposal

ITERATOR CONCEPTS

```
template<class In>           concept Readable;
template<class Out, class T> concept Writable;
template<class I>            concept WeaklyIncrementable;
template<class I>            concept Incrementable;
template<class I>            concept Iterator;
template<class S, class I>   concept Sentinel;
template<class S, class I>   concept SizedSentinel;
template<class I>            concept InputIterator;
template<class I, class T>   concept OutputIterator;
template<class I>            concept ForwardIterator;
template<class I>            concept BidirectionalIterator;
template<class I>            concept RandomAccessIterator;
template<class I>            concept ContiguousIterator;
```

P0896 The One Ranges Proposal

ITERATORS AND SENTINELS

```
inline constexpr default_sentinel_t default_sentinel{};  
inline constexpr unreachable_t unreachable{};
```

```
template<Semiregular S>  
class move_sentinel;
```

P0896 The One Ranges Proposal

ITERATORS AND SENTINELS

```
inline constexpr default_sentinel_t default_sentinel{};  
inline constexpr unreachable_t unreachable{};
```

```
template<Semiregular S>  
class move_sentinel;
```

```
template<Iterator I>  
class counted_iterator;
```

```
template<Iterator I, Sentinel<I> S>  
requires !Same<I, S>  
class common_iterator;
```

P0896 The One Ranges Proposal

A range is an iterator and a sentinel that designate the *beginning and end* of the computation

P0896 The One Ranges Proposal

A range is an iterator and a sentinel that designate the *beginning and end* of the computation

A counted range is an iterator and a count that designate the *beginning and the number of elements* to which the computation is to be applied

P0896 The One Ranges Proposal

A range is an iterator and a sentinel that designate the *beginning and end* of the computation

A counted range is an iterator and a count that designate the *beginning and the number of elements* to which the computation is to be applied

An iterator and a sentinel denoting a range *are comparable*

P0896 The One Ranges Proposal

RANGE CONCEPTS

```
template<class T>          concept Range;
template<class T>          concept SizedRange;
template<class R, class T> concept OutputRange;
template<class T>          concept InputRange;
template<class T>          concept ForwardRange;
template<class T>          concept BidirectionalRange;
template<class T>          concept RandomAccessRange;
template<class T>          concept ContiguousRange;
template<class T>          concept CommonRange;
```

P0896 The One Ranges Proposal

IEWS

```
template<class T> concept View;
template<class T> concept ViewableRange;
```

P0896 The One Ranges Proposal

IEWS

```
template<class T> concept View;
template<class T> concept ViewableRange;
```

```
template<class D>
  requires is_class_v<D> && Same<D, remove_cv_t<D>>
class view_interface;
```

P0896 The One Ranges Proposal

VIEWS

```
template<class T> concept View;
template<class T> concept ViewableRange;
```

```
template<class D>
  requires is_class_v<D> && Same<D, remove_cv_t<D>>
class view_interface;
```

```
template<Iterator I, Sentinel<I> S = I,
         subrange_kind K = SizedSentinel<S, I> ? subrange_kind::sized : subrange_kind::unsized>
  requires (K == subrange_kind::sized || !SizedSentinel<S, I>)
class subrange;
```

P0896 The One Ranges Proposal

VIEWS

```
template<class T> concept View;
template<class T> concept ViewableRange;
```

```
template<class D>
  requires is_class_v<D> && Same<D, remove_cv_t<D>>
class view_interface;
```

```
template<Iterator I, Sentinel<I> S = I,
         subrange_kind K = SizedSentinel<S, I> ? subrange_kind::sized : subrange_kind::unsized>
  requires (K == subrange_kind::sized || !SizedSentinel<S, I>)
class subrange;
```

```
template<View V>
  requires (!CommonRange<V>)
class common_view;
```

P0896 The One Ranges Proposal

VIEWS

```
template<ViewableRange R>
using all_view = /* ... */;

template<InputRange V, IndirectUnaryPredicate<iterator_t<V>> Pred>
    requires View<V>
class filter_view;

template<InputRange V, CopyConstructible F>
    requires View<V> && is_object_v<F> && RegularInvocable<F&, iter_reference_t<iterator_t<V>>>
class transform_view;

template<WeaklyIncrementable W, Semiregular Bound = unreachable_sentinel_t>
    requires weakly-equality-comparable-with<W, Bound>
class iota_view;

template<View>
class take_view;
```

P0896 The One Ranges Proposal

VIEWS

```
template<InputRange V>
requires View<V> && InputRange<iter_reference_t<iterator_t<V>>> &&
(is_reference_v<iter_reference_t<iterator_t<V>>> || View<iter_value_t<iterator_t<V>>>)
class join_view;

template<class T>
requires is_object_v<T>
class empty_view;

template<CopyConstructible T>
requires is_object_v<T>
class single_view;

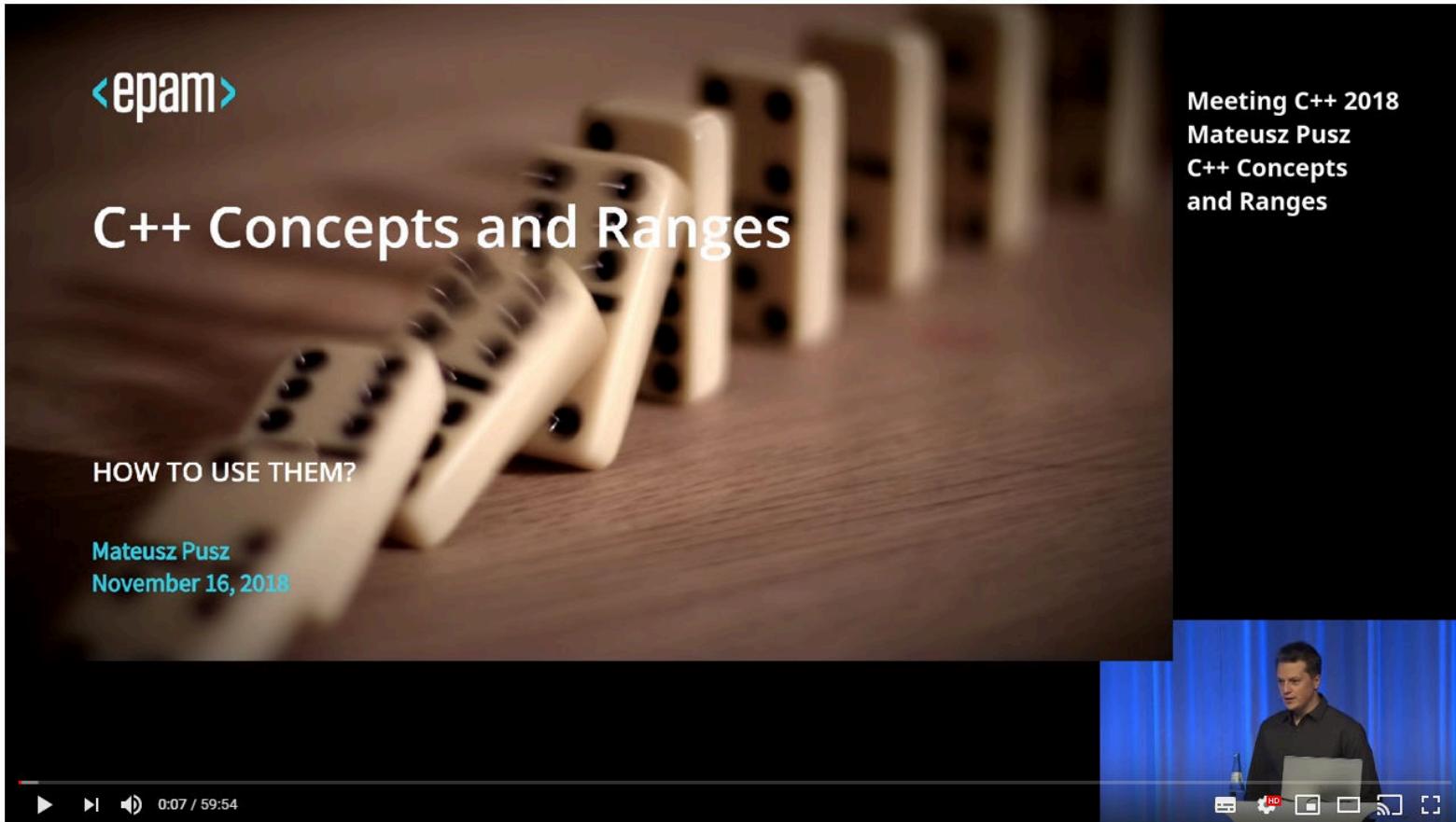
template<InputRange V, ForwardRange Pattern>
requires View<V> && View<Pattern> &&
IndirectlyComparable<iterator_t<V>, iterator_t<Pattern>, ranges::equal_to<>> &&
(ForwardRange<V> || tiny_range<Pattern>)
class split_view;
```

P0896 The One Ranges Proposal

IEWS

```
template<View V>
    requires BidirectionalRange<V>
class reverse_view;
```

More info on Concepts and Ranges



C++ Concepts and Ranges - Mateusz Pusz - Meeting C++ 2018

P1185 <=> != ==

MOTIVATION

```
struct S {  
    std::vector<std::string> names;  
    auto operator<=>(const S&) const = default;  
};
```

P1185 <=> != ==

MOTIVATION

```
struct S {  
    std::vector<std::string> names;  
    auto operator<=>(const S&) const = default;  
};
```

```
template<typename T, typename A1, typename A2>  
std::strong_ordering operator<=>(const std::vector<T, A1>& lhs, const std::vector<T, A2>& rhs)  
{  
    const size_t min_size = std::min(lhs.size(), rhs.size());  
    for(size_t i = 0; i != min_size; ++i)  
        if(const auto cmp = std::compare_3way(lhs[i], rhs[i]); cmp != 0) return cmp;  
    return lhs.size() <=> rhs.size();  
}
```

P1185 <=> != ==

MOTIVATION

```
struct S {  
    std::vector<std::string> names;  
    auto operator<=>(const S&) const = default;  
};
```

```
template<typename T, typename A1, typename A2>  
std::strong_ordering operator<=>(const std::vector<T, A1>& lhs, const std::vector<T, A2>& rhs)  
{  
    const size_t min_size = std::min(lhs.size(), rhs.size());  
    for(size_t i = 0; i != min_size; ++i)  
        if(const auto cmp = std::compare_3way(lhs[i], rhs[i]); cmp != 0) return cmp;  
    return lhs.size() <=> rhs.size();  
}
```

```
std::vector<int> v1 = { 1, 2, 3, 4, 5 };  
std::vector<int> v2 = { 1, 2, 3, 4 };  
if(v1 == v2) { /* ... */ }
```

P1185 <=> != ==

SOLUTION

- Change the **candidate set for operator lookup**
 - the **equality** operators *will not consider* \leftrightarrow candidates
 - **inequality** *will consider* equality as a candidate
 - no changes for the relational operators

SOURCE	TODAY	PROPOSED
$a == b$	$a == b$ $(a \leftrightarrow b) == 0$ $0 == (b \leftrightarrow a)$	$a == b$ $b == a$
$a != b$	$a != b$ $(a \leftrightarrow b) != 0$ $0 != (a \leftrightarrow b)$	$a != b$ $!(a == b)$ $!(b == a)$

P1185 <=> != ==

SOLUTION

- Change the meaning of **defaulted equality operators**
 - **operator==()** = **default** generates a member-wise equality comparison
 - **operator!=()** = **default** generates a call to negated **op==**

```
struct X {  
    A a;  
    B b;  
  
    auto operator<=>(const X&) const = default;  
    bool operator==(const X&) const = default;  
    bool operator!=(const X&) const = default;  
};
```

```
bool X::operator==(const X& rhs) const { return a == rhs.a && b == rhs.b; }  
bool X::operator!=(const X& rhs) const { return !(*this == rhs); }
```

SOLUTION

- Change the **definition of the strong structural equality**
 - a type T is *having strong structural equality* if each subobject recursively has defaulted == and none of the subobjects are floating point types

Strong structural equality is the criteria to allow a class to be used as a non-type template parameter

P1185 <=> != ==

SOLUTION

- defaulted <=> to also generate a defaulted ==
 - we still get optimal equality

```
// all six comparisons
struct A {
    auto operator<=>(const A&) const = default;
    // bool operator==(const A&) const = default; // implicitly defined
};

// just equality, no relational
struct B {
    bool operator==(const B&) const = default;
};
```

P1120 Consistency improvements for `<=>` and other comparison operators

MOTIVATION

The specification of the `<=> operator` reexamined the rules underlying comparison operators, and *disallowed some cases that have historically been allowed, but error-prone*, for the other relational and equality operators

P1120 Consistency improvements for `<=>` and other comparison operators

MOTIVATION

The specification of the `<=> operator` reexamined the rules underlying comparison operators, and *disallowed some cases that have historically been allowed, but error-prone*, for the other relational and equality operators

SOLUTION

Improve the consistency of `<=>` with other operators, by deprecating some cases where they diverge and tweaking the `<=>` rules

- **Deprecate** usual arithmetic conversions between *enumeration types and floating-point types*
- **Deprecate** usual arithmetic conversions between *two distinct enumeration types*
- **Deprecate** two-way comparisons where *both operators are array type*
- **Permit** three-way comparison between *unscoped enumeration types and integral types*

P1120 Consistency improvements for `<=>` and other comparison operators

- The ability to apply the usual *arithmetic conversions* on operands where one is of *enumeration* type and the other is of a *different enumeration* type or a *floating-point* type is **deprecated**

```
enum E1 { e };
enum E2 { f };
bool b = e <= 3.7;      // deprecated
int k = f - e;         // deprecated
auto cmp = e <=> f;   // ill-formed
```

P1120 Consistency improvements for `<=>` and other comparison operators

- The ability to apply the usual *arithmetic conversions* on operands where one is of *enumeration* type and the other is of a *different enumeration* type or a *floating-point* type is **deprecated**

```
enum E1 { e };
enum E2 { f };
bool b = e <= 3.7;      // deprecated
int k = f - e;          // deprecated
auto cmp = e <=> f;    // ill-formed
```

- Equality and relational* comparisons between two operands of *array type* are **deprecated**

```
int arr1[5];
int arr2[5];
bool same = arr1 == arr2;    // deprecated, same as &arr[0] == &arr[1],
                            // does not compare array contents
auto cmp = arr1 <=> arr2;  // ill-formed
```

P0542 Contracts

```
[[contract-attribute modifier identifier: expression]]
```

P0542 Contracts

```
[[contract-attribute modifier identifier: expression]]
```

- **contract-attribute**

- **expects** - function *precondition*
- **ensures** - function *postcondition*
- **assert** - statement *verification*

P0542 Contracts

```
[[contract-attribute modifier identifier: expression]]
```

- **contract-attribute**
 - **expects** - function *precondition*
 - **ensures** - function *postcondition*
 - **assert** - statement *verification*
- **modifier** (optional) - defines assertion level
 - **axiom** - formal comments *not evaluated at run-time*
 - **default** - the cost of run-time checking is assumed to be *small*
 - **audit** - the cost of run-time checking is assumed to be *large*

P0542 Contracts

```
[[contract-attribute modifier identifier: expression]]
```

- **contract-attribute**
 - **expects** - function *precondition*
 - **ensures** - function *postcondition*
 - **assert** - statement *verification*
- **modifier** (optional) - defines assertion level
 - **axiom** - formal comments *not evaluated at run-time*
 - **default** - the cost of run-time checking is assumed to be *small*
 - **audit** - the cost of run-time checking is assumed to be *large*
- **identifier** (optional) - used only for *function return value* in postconditions

P0542 Contracts: Example

```
int f(int x)
  [[expects audit: x>0]]
  [[ensures axiom res: res>1]];

void g()
{
    int x = f(5);
    int y = f(12);
    [[assert: x+y>0]]
    //...
}
```

P0542 Contracts: Example

```
int f(int x)
  [[expects audit: x>0]]
  [[ensures axiom res: res>1]];

void g()
{
    int x = f(5);
    int y = f(12);
    [[assert: x+y>0]]
    //...
}
```

```
bool positive(int* p) [[expects: p!=nullptr]]
{
    return *p > 0;
}

bool g(int* p) [[expects: positive(p)]];

void test()
{
    g(nullptr);      // Contract violation
}
```

P0542 Contracts: Example

```
int f(int x)
  [[expects audit: x>0]]
  [[ensures axiom res: res>1]];

void g()
{
  int x = f(5);
  int y = f(12);
  [[assert: x+y>0]]
  //...
}
```

```
bool positive(int* p) [[expects: p!=nullptr]]
{
  return *p > 0;
}

bool g(int* p) [[expects: positive(p)]];

void test()
{
  g(nullptr);    // Contract violation
}
```

All contracts are required to pass name lookup independently of their assertion level

P0542 Contracts: Redeclaration

- Redeclaration of a function *either has the contract* or completely *omits it*

```
int f(int x)
  [[expects: x>0]]
  [[ensures r: r>0]];
```

P0542 Contracts: Redeclaration

- Redeclaration of a function *either has the contract* or completely *omits it*

```
int f(int x)
  [[expects: x>0]]
  [[ensures r: r>0]];
```

```
int f(int x);           // OK: no contract
```

```
int f(int x)
  [[expects: x>=0]];    // Error: missing ensures and different expects condition
```

```
int f(int y)
  [[expects: y>0]]
  [[ensures res: res>0]]; // OK: same contract even though names differ
```

P0542 Contracts: Redeclaration rules

- Preconditions and postconditions appear in *the same order*
- Their *modifiers are the same*
- Each *conditional expression would satisfy the ODR* if it appeared in function definition, except for renaming of parameters and return value identifiers

P0542 Contracts: Contract violation

- If a contract violation is detected, a **violation handler** will be invoked

```
void(const std::contractViolation&); // violation handler signature
```

P0542 Contracts: Contract violation

- If a contract violation is detected, a **violation handler** will be invoked

```
void(const std::contractViolation&); // violation handler signature
```

```
namespace std {
    class contractViolation {
    public:
        int line_number() const noexcept;
        string_view file_name() const noexcept;
        string_view function_name() const noexcept;
        string_view comment() const noexcept;
        string_view assertion_level() const noexcept;
    };
}
```

P0542 Contracts: Contract violation

- If a contract violation is detected, a **violation handler** will be invoked

```
void(const std::contractViolation&); // violation handler signature
```

```
namespace std {
    class contractViolation {
    public:
        int lineNumber() const noexcept;
        string_view fileName() const noexcept;
        string_view functionName() const noexcept;
        string_view comment() const noexcept;
        string_view assertionLevel() const noexcept;
    };
}
```

- Establishing violation handler and setting its argument is *implementation defined*
- *Violation continuation mode* can be *off*(default) or *on*
 - *off* calls **std::terminate()** after completing the execution of the violation handler

P0542 Contracts: Contract violation

- If a user-provided violation handler *exits by throwing an exception* and a contract is violated on a call to a **constexpr** function then **std::terminate()** is called

```
void f(int x) noexcept [[expects: x>0]];

void g()
{
    f(0); // std::terminate() if violation handler throws
    //...
}
```

P0542 Contracts: Side effects

- Contracts conditions are expected to be observable side-effect free
- Any observable effect in a contract condition leads to *undefined behavior*

```
int x;
volatile int y;

void f(int n) [[expects: n>x]]; // OK
void g(int n) [[expects: n>x++]]; // Undefined behavior
void h(int n) [[expects: n++>0]]; // Undefined behavior
void j()
{
    int n=3;
    [[assert: ++n>3]];           // Undefined behavior
    //...
}
```

P0542 Contracts: Side effects

- Calling a function that potentially might modify a variable or global state is also an *undefined behavior*

```
bool might_increment(int& x);

void f(int n) [[expects: might_increment(n)]]; // Undefined behavior
```

P0542 Contracts: Side effects

- Calling a function that potentially might modify a variable or global state is also an *undefined behavior*

```
bool might_increment(int& x);

void f(int n) [[expects: might_increment(n)]]; // Undefined behavior
```

```
bool is_valid(int x)
{
    std::cerr << "checking x\n";
    return x>0;
}

void g(int n) [[expects: is_valid(n)]];           // Undefined behavior
```

P0542 Contracts: Side effects

- *Local side effects in functions invoked in the conditional expression are allowed*
- They are not observable from outside that function

```
bool is_valid(int x)
{
    int a=1;
    while(a<x) {
        if(x % a == 0) return true;
        a++;
    }
    return false;
}

void f(int n) [[expects: is_valid(x)]]
```

P0542 Contracts: `constexpr`

- Contracts conditions appearing in a `constexpr` function must operate only on `constexpr` context

```
int min=-42;
constexpr int max=42;

constexpr int g(int x)
    [[expects: min<=x]]      // Error
    [[expects: x<max]]       // OK
{
    //...
    [[assert: 2*x < max]]; // OK
    [[assert: ++min > 0]]; // Error
    //...
}
```

P0542 Contracts: Postconditions using arguments

- If a *postcondition uses a function parameter* and the function body makes *modifications of its value* the **behavior is undefined**

P0542 Contracts: Postconditions using arguments

- If a *postcondition uses a function parameter* and the function body makes *modifications of its value* the **behavior is undefined**

```
int f(int x)
  [[ensures r: r==x]]
{
  return ++x; // Undefined behavior
}
```

P0542 Contracts: Postconditions using arguments

- If a *postcondition uses a function parameter* and the function body makes *modifications of its value* the **behavior is undefined**

```
int f(int x)
  [[ensures r: r==x]]
{
  return ++x; // Undefined behavior
}
```

```
int g(int * p)
  [[ensures r: p!=nullptr]]
{
  *p = 42; // OK, p is not modified
}
```

P0542 Contracts: Postconditions using arguments

- If a *postcondition uses a function parameter* and the function body makes *modifications of its value* the **behavior is undefined**

```
int f(int x)
  [[ensures r: r==x]]
{
  return ++x; // Undefined behavior
}
```

```
int g(int * p)
  [[ensures r: p!=nullptr]]
{
  *p = 42; // OK. p is not modified
}
```

```
int h(int x)
  [[ensures r: r==x]]
{
  potentially_modify(x); // Undefined behavior if x is modified
  return x;
}
```

P0542 Contracts: Classes

- An overriding function shall have *the same list of contract conditions* as the overridden function
- The list of contract conditions in the overriding function *may be omitted*
 - assumed to be the list of the overridden function

```
class X : noncopyable {
public:
    virtual ~X();
    virtual int f(int a) [[expects: a>0]];
    virtual int g(int a) [[expects: a>0]];
    virtual int h(int a) [[expects: a>0]];
}
```

```
class Y : public X {
public:
    int f(int a) override; // OK
    int g(int a) override [[expects: a>0]]; // OK
    int h(int a) override [[expects: a>1]]; // Ill-formed
}
```

P0542 Contracts: Function pointers

- A function pointer *shall not include* contract conditions

```
typedef int (*fpt)() [[ensures r: r!=0]]; // Ill-formed
```

P0542 Contracts: Function pointers

- A function pointer *shall not include* contract conditions

```
typedef int (*fpt)() [[ensures r: r!=0]]; // Ill-formed
```

- A call through a function pointer to functions with contract conditions *performs contract assertions checking once*

```
int g(int x)
[[expects: x>=0]]
[[ensures r: r>x]]
{
    return x+1;
}

int (*pf)(int) = g; // OK
```

P0542 Contracts: Asserts

C ASSERT

- uses preprocessor macro
- does not understand commas

```
assert(c==std::complex<float>{0,0});
```

- enforcement enabled in compile time only
(debug build by default)

P0542 Contracts: Asserts

C ASSERT

- uses preprocessor macro
- does not understand commas

```
assert(c==std::complex<float>{0,0});
```

- enforcement enabled in compile time only
(debug build by default)

CONTRACT ASSERT

- language feature
- works just fine with all the language features

```
[[assert: c==std::complex<float>{0,0}]]
```

- possible to enable the enforcement in runtime
in release build
- **audit** and **axiom** for more control and verbosity
- likely to get better performance
 - enable compilers to perform more optimizations

P1289 Access control in contract conditions

MOTIVATION

- Right now access controls enforce that only members with the *same or better access level* are used in the predicates
- In some cases *additional artificial API has to be added* exclusively to provide contract checks

P1289 Access control in contract conditions

MOTIVATION

- Right now access controls enforce that only members with the *same or better access level* are used in the predicates
- In some cases *additional artificial API has to be added* exclusively to provide contract checks

```
class X {  
    double* ptr_;  
public:  
    // ...  
    // ptr must be non-null  
    double get_value() const  
    {  
        return *ptr_;  
    }  
};
```

P1289 Access control in contract conditions

MOTIVATION

- Right now access controls enforce that only members with the *same or better access level* are used in the predicates
- In some cases *additional artificial API has to be added* exclusively to provide contract checks

```
class X {  
    double* ptr_;  
public:  
    // ...  
    // ptr must be non-null  
    double get_value() const  
    {  
        return *ptr_;  
    }  
};
```

```
class X {  
    double* ptr_;  
public:  
    // ...  
    double* get_ptr() const { return ptr_; }  
  
    double get_value() const  
        [[expects: get_ptr() != nullptr]]  
    {  
        return *ptr_;  
    }  
};
```

P1289 Access control in contract conditions

SOLUTION

- Allow that *every member* can be used in a contract conditions
- Makes *contract checking orthogonal to programmatic APIs*
 - apply contracts to any existing API without changing the API
 - design contract-enforced APIs without having to introduce API entry points that expose the correctness checks

P1289 Access control in contract conditions

SOLUTION

- Allow that *every member* can be used in a contract conditions
- Makes *contract checking orthogonal to programmatic APIs*
 - apply contracts to any existing API without changing the API
 - design contract-enforced APIs without having to introduce API entry points that expose the correctness checks

```
class X {
    double* ptr_;
public:
    // ...
    double get_value() const [[expects: ptr_ != nullptr]]
    {
        return *ptr_;
    }
};
```

P1103 Merging Modules

- An *alternative to header files*
- *Isolate* the effect of *macros*
- Enable *scalable builds*
- Adds a *new encapsulation boundary*
 - variable
 - function
 - class
 - **module**

P1103 Merging Modules: Module translation unit structure

```
module;          // global module fragment
#preprocessor-directive // global module fragment
...
#preprocessor-directive // global module fragment

module-declaration // module preamble
import-declaration // module preamble
...
import-declaration // module preamble

declaration
...
declaration

module : private; // inline module implementation partition
implementation // inline module implementation partition
...
implementation // inline module implementation partition
implementation // inline module implementation partition
```

P1103 Merging Modules: Module

A module unit is a translation unit (TU) that contains a module declaration

P1103 Merging Modules: Module

A module unit is a translation unit (TU) that contains a module declaration

A named module is the collection of module units with the same module name

P1103 Merging Modules: Module

A module unit is a translation unit (TU) that contains a module declaration

A named module is the collection of module units with the same module name

A module is either a named module or the global module

P1103 Merging Modules: Module vs Translation Units

- 1 A *single source file*
- 2 *Interface and implementation units*
- 3 Both interface and implementation spreaded among *many partition units*

P1103 Merging Modules: Module vs Translation Units

- 1 A *single source file*
- 2 *Interface and implementation units*
- 3 Both interface and implementation spreaded among *many partition units*
- 4 Module can import "legacy" header files

P1103 Merging Modules: Module vs Translation Units

- 1 A *single source file*
- 2 *Interface and implementation units*
- 3 Both interface and implementation spreaded among *many partition units*
- 4 Module can import "legacy" header files
- 5 Module can be imported in a non-modular "legacy" implementation

P1103 Merging Modules: Module Units

INTERFACE UNIT

- Defines the *external interface* of the module

```
export module module_name;
```

P1103 Merging Modules: Module Units

INTERFACE UNIT

- Defines the *external interface* of the module

```
export module module_name;
```

IMPLEMENTATION UNIT

- Provides *implementation details* within the semantic scope of the module

```
module module_name;
```

P1103 Merging Modules: Module Units

INTERFACE UNIT

- Defines the *external interface* of the module

```
export module module_name;
```

MODULE INTERFACE PARTITION

- Part of the module interface

```
export module module_name : partition_name;
```

IMPLEMENTATION UNIT

- Provides *implementation details* within the semantic scope of the module

```
module module_name;
```

P1103 Merging Modules: Module Units

INTERFACE UNIT

- Defines the *external interface* of the module

```
export module module_name;
```

MODULE INTERFACE PARTITION

- Part of the module interface

```
export module module_name : partition_name;
```

IMPLEMENTATION UNIT

- Provides *implementation details* within the semantic scope of the module

```
module module_name;
```

MODULE IMPLEMENTATION PARTITION

- Part of the implementation details of the module

```
module module_name : partition_name;
```

P1103 Merging Modules: Module Interface Unit

```
export module foo;

import a;
export import b;
// ... more imports ...

// module definition
```

P1103 Merging Modules: Module Interface Unit

```
export module foo;  
  
import a;  
export import b;  
// ... more imports ...  
  
// module definition
```

- *Imports may not appear* after the end of the module preamble

P1103 Merging Modules: Module Interface Unit

```
export module foo;  
  
import a;  
export import b;  
// ... more imports ...  
  
// module definition
```

- *Imports may not appear* after the end of the module preamble

A named module shall contain exactly one module interface unit without partition name, known as the primary module interface unit of the module

P1103 Merging Modules: Exporting declarations

- A *declaration can be exported* by the use of the **export keyword** in a module interface unit

```
export module A;

export int a;

export namespace A {}
namespace B {
    export int n;
}

export {
    void f();
}
```

P1103 Merging Modules: Exporting declarations

- A *declaration can be exported* by the use of the **export keyword** in a module interface unit

```
export module A;

export int a;

export namespace A {}
namespace B {
    export int n;
}

export {
    void f();
}
```

- *Exported declarations* in a module interface unit *are visible to name lookup* in contexts that *import this module* interface unit

P1103 Merging Modules: Namespace name exports

- The *name of a namespace is exported* if it is ever *declared within an export declaration*
- A namespace name is also *implicitly exported if any name within it is exported* (recursively)

P1103 Merging Modules: Namespace name exports

- The *name of a namespace is exported* if it is ever *declared within an export declaration*
- A namespace name is also *implicitly exported if any name within it is exported* (recursively)

```
export module namespaces;

export namespace A { // A is exported
    int n;           // A::n is exported
}

namespace B {
    export int n;    // B::n is exported and B is implicitly exported
    int m;           // B::m is not exported
}

namespace C {
    int n;
}

export namespace C {} // C is exported, C::n is not
```

P1103 Merging Modules: Imports

- Importing a TU makes its *interface available to the importing code* (including declaration names, semantic effects of declarations, and macros)

P1103 Merging Modules: Imports

- Importing a TU makes its *interface available to the importing code* (including declaration names, semantic effects of declarations, and macros)
- The *interface of a module* can be imported by importing its interface unit

P1103 Merging Modules: Imports

- Importing a TU makes its *interface available to the importing code* (including declaration names, semantic effects of declarations, and macros)
- The *interface of a module* can be imported by importing its interface unit
- *Partitions of the current module* can be imported by name

```
export module A;  
export struct X {};
```

```
import A;  
X x{};
```

P1103 Merging Modules: Imports

- Importing a TU makes its *interface available to the importing code* (including declaration names, semantic effects of declarations, and macros)
- The *interface of a module* can be imported by importing its interface unit
- *Partitions of the current module* can be imported by name

```
export module A;  
export struct X {};
```

```
import A;  
X x{};
```

If the imported TU and the current TU *are owned by the same module*, all namespace-scope names and macros from the imported translation unit are made visible in the current TU, *regardless of whether they are exported*

Example: Exporting declarations

A.CPP

```
export module A; // interface of module A
int foo() { return 1; }
export int bar();
```

Example: Exporting declarations

A.CPP

```
export module A; // interface of module A
int foo() { return 1; }
export int bar();
```

A-IMPL.CPP

```
module A;
int bar() { return foo() + 1; }
```

- Implicitly imports the interface of A
- **foo()** is visible here (in the same module) even though it was not exported from A

Example: Exporting declarations

A.CPP

```
export module A; // interface of module A
int foo() { return 1; }
export int bar();
```

A-IMPL.CPP

```
module A;

int bar() { return foo() + 1; }
```

- Implicitly imports the interface of A
- **foo()** is visible here (in the same module) even though it was not exported from A

UNRELATED.CPP

```
import A;

int main()
{
    bar();      // OK
    foo();      // Error
}
```

- **bar** was exported by A and is visible here

Example: Reexporting a module

Q.CPP

```
export module Q;
export int sq(int i) { return i*i; }
```

Example: Reexporting a module

Q.CPP

```
export module Q;
export int sq(int i) { return i*i; }
```

R.CPP

```
export module R;
export import Q;
```

Example: Reexporting a module

Q.CPP

```
export module Q;
export int sq(int i) { return i*i; }
```

R.CPP

```
export module R;
export import Q;
```

MAIN.CPP

```
import R;
int main()
{
    return sq(9); // OK: sq from module Q
}
```

P1103 Merging Modules: Module partitions

- A complete module can be defined in a single source file
- If needed both the implementation and the interface *may be divided into multiple files*

P1103 Merging Modules: Module partitions

- A complete module can be defined in a single source file
- If needed both the implementation and the interface *may be divided into multiple files*

INTERFACE

- The module interface may be split across multiple files called **module interface partitions**

```
export module foo:part;
```

- An entity *may be declared in one partition and defined in another*
 - sometimes necessary to resolve dependency cycles
- The *primary module interface unit* for a module is required to *transitively import and re-export* all of the *interface partitions* of the module

P1103 Merging Modules: Module partitions

- A complete module can be defined in a single source file
- If needed both the implementation and the interface *may be divided into multiple files*

IMPLEMENTATION

- **Module implementation partitions** allow sharing declarations between the implementation units without including them in the *module interface unit*

```
module foo:part;
```

- *Cannot contain exported declarations*
- All their declarations are *visible to other translation units in the same module* that import the partition
- *Can be imported* into the interface of a module, but *cannot be exported*

P1103 Merging Modules: Module partitions

- A complete module can be defined in a single source file
- If needed both the implementation and the interface *may be divided into multiple files*

IMPORTING MODULE PARTITIONS

- Module partitions are an implementation detail of the module, and *cannot be named outside the module*
- An import declaration of a module partition *cannot be given a module name*, only a partition name

```
module foo;  
import :part;      // imports foo:part  
import bar:part;   // syntax error  
import foo:part;   // syntax error
```

Example: Module partitions

MODULE PARTITIONS

```
export module widget:base;  
export class Widget {};
```

```
export module widget:bolt;  
import :base;  
export class Bolt : Widget {};
```

Example: Module partitions

MODULE PARTITIONS

```
export module widget:base;  
export class Widget {};
```

```
export module widget:bolt;  
import :base;  
export class Bolt : Widget {};
```

INTERFACE UNIT

```
export module widget;  
export import :base;  
export import :bolt;  
  
void frob(Widget*);
```

Example: Module partitions

MODULE PARTITIONS

```
export module widget:base;  
export class Widget {};
```

```
export module widget:bolt;  
import :base;  
export class Bolt : Widget {};
```

```
module widget:utils;  
import widget;  
import std::vector;  
  
inline void frob_helper(  
    const std::vector<Widget*>& widgets)  
{  
    for(Widget* w : widgets)  
        frob(w);  
}
```

INTERFACE UNIT

```
export module widget;  
export import :base;  
export import :bolt;  
  
void frob(Widget*);
```

Example: Module partitions

MODULE PARTITIONS

```
export module widget:base;  
export class Widget {};
```

```
export module widget:bolt;  
import :base;  
export class Bolt : Widget {};
```

```
module widget:utils;  
import widget;  
import std::vector;  
  
inline void frob_helper(  
    const std::vector<Widget*>& widgets)  
{  
    for(Widget* w : widgets)  
        frob(w);  
}
```

INTERFACE UNIT

```
export module widget;  
export import :base;  
export import :bolt;  
  
void frob(Widget*);
```

IMPLEMENTATION UNIT

```
module widget;  
import :utils;  
  
void frob(Widget* w)  
{  
    frob_helper(children(w));  
}
```

Example: Module partitions

MODULE PARTITIONS

```
export module widget:base;  
export class Widget {};
```

```
export module widget:bolt;  
import :base;  
export class Bolt : Widget {};
```

```
module widget:utils;  
import widget;  
import std::vector;  
  
inline void frob_helper(  
    const std::vector<Widget*>& widgets)  
{  
    for(Widget* w : widgets)  
        frob(w);  
}
```

INTERFACE UNIT

```
export module widget;  
export import :base;  
export import :bolt;
```

```
void frob(Widget*);
```

IMPLEMENTATION UNIT

```
module widget;  
import :utils;  
  
void frob(Widget* w)  
{  
    frob_helper(children(w));  
}
```

Example: Module partitions

MODULE PARTITIONS

```
export module widget:base;  
export class Widget {};
```

```
export module widget:bolt;  
import :base;  
export class Bolt : Widget {};
```

```
module widget:utils;  
import widget;  
import std::vector;  
  
inline void frob_helper(  
    const std::vector<Widget*>& widgets)  
{  
    for(Widget* w : widgets)  
        frob(w);  
}
```

INTERFACE UNIT

```
export module widget;  
  
void frob(Widget*);
```

IMPLEMENTATION UNIT

```
module widget;  
import :utils;  
  
void frob(Widget* w)  
{  
    frob_helper(children(w));  
}
```

P1103 Merging Modules: Inline module implementation partition

- A complete module (with both interface and implementation) *can be defined in a single source file* by separating the interface from the implementation

```
export module m;
struct s;
export using s_ptr = s*;

module : private;
struct s {};
```

P1103 Merging Modules: Inline module implementation partition

- A complete module (with both interface and implementation) *can be defined in a single source file* by separating the interface from the implementation

```
export module m;  
struct s;  
export using s_ptr = s*;  
  
module : private;  
struct s {};
```

Shall appear only in a primary module interface unit which should be the only module unit of its module

P1103 Merging Modules: Support for non-modular code

- Permits existing header files to be used from modular code without sacrificing modularity
 - *names do not leak* into other translation units that import the module unit
 - *compilation performance* is not sacrificed by recompiling the same header files on every inclusion
 - users do not need to resort to the preprocessor to access the interfaces of non-modular libraries
- Support provided with
 - **Global module fragment**
 - **Header Units**

P1103 Merging Modules: Module use from non-modular code

- Modules and header units *can be imported into non-modular code*
 - imports can appear anywhere (not restricted to a preamble)
- Permits *“bottom-up” modularization*
 - using library with modular interface as a dependency in "legacy" code
 - defining its header interface in terms of the modular interface

MODULE_FOO.CPP

```
export module foo;
export import "some-header.h";
```

LEGACY.CPP

```
#include <vector>
import foo;
#include "some-header.h"; // treated as an
                        // import of the corresponding
                        // header unit
```

P0912 Merge Coroutines TS into C++20 working draft

- Coroutines dramatically *simplify development of asynchronous code*
- Are available and *in use for 5 years*
- *Shipping implementations* from two major compiler vendors (MSVC & clang)
- The software built using coroutines on Linux and Windows is powering the foundational services of Windows Azure cloud services

P0912 Merge Coroutines TS into C++20 working draft

- Coroutines dramatically *simplify development of asynchronous code*
- Are available and *in use for 5 years*
- *Shipping implementations* from two major compiler vendors (MSVC & clang)
- The software built using coroutines on Linux and Windows is powering the foundational services of Windows Azure cloud services

THIRD TIME LUCKY

APPROVE	AGAINST	ABSTAIN
48	4	15

Coroutine

A function is a **coroutine** if it contains

- **`co_return`** statement
- **`co_await`** expression
- **`co_yield`** expression
- **`for co_await()`** statement

Coroutines

```
task<int> f();

task<void> g1()
{
    int i = co_await f();
    std::cout << "f() => " << i << std::endl;
}

template<typename... Args>
task<void> g2(Args&&...) // OK: ellipsis is a pack expansion
{
    int i = co_await f();
    std::cout << "f() => " << i << std::endl;
}

task<void> g3(int a, ...) // error: variable parameter list not allowed
{
    int i = co_await f();
    std::cout << "f() => " << i << std::endl;
}
```

coroutine_traits

```
template<class R, class... ArgTypes>
struct coroutine_traits;
```

```
template<class R, class... ArgTypes>
    requires requires { typename R::promise_type; }
struct coroutine_traits<R, ArgTypes...> {
    using promise_type = typename R::promise_type;
};
```

Program defined specializations of this template shall define a publicly accessible nested type named **promise_type**

coroutine_handle

```
template<class Promise = void>
struct coroutine_handle;
```

- `coroutine_handle<P>` can be used *to refer to a suspended or executing coroutine*
- A *default constructed* `coroutine_handle` object *does not refer to any coroutine*

`coroutine_handle`

```
template<class Promise = void>
struct coroutine_handle;
```

- `coroutine_handle<P>` can be used *to refer to a suspended or executing coroutine*
- A *default constructed* `coroutine_handle` object *does not refer to any coroutine*

If a program declares an explicit or partial specialization of `coroutine_handle`, the behavior is undefined

coroutine_handle

```
template<>
struct coroutine_handle<void> {
    // construct/reset
    constexpr coroutine_handle() noexcept;
    constexpr coroutine_handle(nullptr_t) noexcept;
    coroutine_handle& operator=(nullptr_t) noexcept;

private:
    void* ptr = nullptr; // exposition only
};
```

coroutine_handle

```
template<>
struct coroutine_handle<void> {
    // construct/reset
    constexpr coroutine_handle() noexcept;
    constexpr coroutine_handle(nullptr_t) noexcept;
    coroutine_handle& operator=(nullptr_t) noexcept;

    // export/import
    constexpr void* address() const noexcept { return ptr; }
    constexpr static
        coroutine_handle from_address(void* addr);

private:
    void* ptr = nullptr; // exposition only
};
```

- **from_address(address()) == *this**

coroutine_handle

```
template<>
struct coroutine_handle<void> {
    // construct/reset
    constexpr coroutine_handle() noexcept;
    constexpr coroutine_handle(nullptr_t) noexcept;
    coroutine_handle& operator=(nullptr_t) noexcept;

    // export/import
    constexpr void* address() const noexcept { return ptr; }
    constexpr static
        coroutine_handle from_address(void* addr);

    // observers
    constexpr explicit operator bool() const noexcept
    { return address() != nullptr; }
    bool done() const;

private:
    void* ptr = nullptr; // exposition only
};
```

- **done()** returns **true** if the coroutine is suspended at its final suspend point

coroutine_handle

```
template<>
struct coroutine_handle<void> {
    // construct/reset
    constexpr coroutine_handle() noexcept;
    constexpr coroutine_handle(nullptr_t) noexcept;
    coroutine_handle& operator=(nullptr_t) noexcept;

    // export/import
    constexpr void* address() const noexcept { return ptr; }
    constexpr static
        coroutine_handle from_address(void* addr);

    // observers
    constexpr explicit operator bool() const noexcept
    { return address() != nullptr; }
    bool done() const;

    // resumption
    void operator()() const;
    void resume() const;
    void destroy() const;
private:
    void* ptr = nullptr; // exposition only
};
```

```
void operator()() const;
void resume() const;
```

- Resume the execution of the coroutine
- If the coroutine was suspended at its final suspend point behavior is undefined

```
void destroy() const;
```

- Destroys the coroutine
- A concurrent resumption of the coroutine via **resume()**, **operator()**, or **destroy()** may result in a data race

coroutine_handle

```
template<class Promise>
struct coroutine_handle : coroutine_handle<> {
    // construct/reset
    using coroutine_handle<>::coroutine_handle;
    static coroutine_handle from_promise(Promise&);
    coroutine_handle& operator=(nullptr_t) noexcept;

    // export/import
    constexpr static coroutine_handle from_address(void* addr);

    // promise access
    Promise& promise() const;
};
```

Example: Generator

```
generator f() { co_yield 1; co_yield 2; }

int main()
{
    auto g = f();
    while(g.move_next()) std::cout << g.current_value() << std::endl;
}
```

Example: Generator

```
generator f() { co_yield 1; co_yield 2; }

int main()
{
    auto g = f();
    while(g.move_next()) std::cout << g.current_value() << std::endl;
}
```

```
struct generator {
    struct promise_type;
    using handle = std::coroutine_handle<promise_type>;
    generator(const generator&) = delete;
    generator(generator&& rhs) : coro_(std::exchange(rhs.coro_, nullptr)) {}
    ~generator() { if(coro_) coro_.destroy(); }

private:
    generator(handle h) : coro_(h) {}
    handle coro_;
};
```

Example: Generator

```
generator f() { co_yield 1; co_yield 2; }

int main()
{
    auto g = f();
    while(g.move_next()) std::cout << g.current_value() << std::endl;
}
```

```
struct generator {
    struct promise_type;
    using handle = std::coroutine_handle<promise_type>;

    generator(const generator&) = delete;
    generator(generator&& rhs) : coro_(std::exchange(rhs.coro_, nullptr)) {}
    ~generator() { if(coro_) coro_.destroy(); }

    bool move_next() { return coro_ ? (coro_.resume(), !coro_.done()) : false; }

private:
    generator(handle h) : coro_(h) {}
    handle coro_;
};
```

Example: Generator

```
generator f() { co_yield 1; co_yield 2; }

int main()
{
    auto g = f();
    while(g.move_next()) std::cout << g.current_value() << std::endl;
}
```

```
struct generator {
    struct promise_type;
    using handle = std::coroutine_handle<promise_type>;

    generator(const generator&) = delete;
    generator(generator&& rhs) : coro_(std::exchange(rhs.coro_, nullptr)) {}
    ~generator() { if(coro_) coro_.destroy(); }

    bool move_next() { return coro_ ? (coro_.resume(), !coro_.done()) : false; }
    int current_value() { return coro_.promise().current_value; }
private:
    generator(handle h) : coro_(h) {}
    handle coro_;
};
```

Example: Generator

```
generator f() { co_yield 1; co_yield 2; }

int main()
{
    auto g = f();
    while(g.move_next()) std::cout << g.current_value() << std::endl;
}
```

```
struct generator {
    struct promise_type;
    using handle = std::coroutine_handle<promise_type>;

    generator(const generator&) = delete;
    generator(generator&& rhs) : coro_(std::exchange(rhs.coro_, nullptr)) {}
    ~generator() { if(coro_) coro_.destroy(); }

    bool move_next() { return coro_ ? (coro_.resume(), !coro_.done()) : false; }
    int current_value() { return coro_.promise().current_value; }
private:
    generator(handle h) : coro_(h) {}
    handle coro_;
};
```

- `::operator new(size_t, noexcept_t)` will be used if allocation is needed

Example: Generator

```
generator f() { co_yield 1; co_yield 2; }

int main()
{
    auto g = f();
    while(g.move_next()) std::cout << g.current_value() << std::endl;
}
```

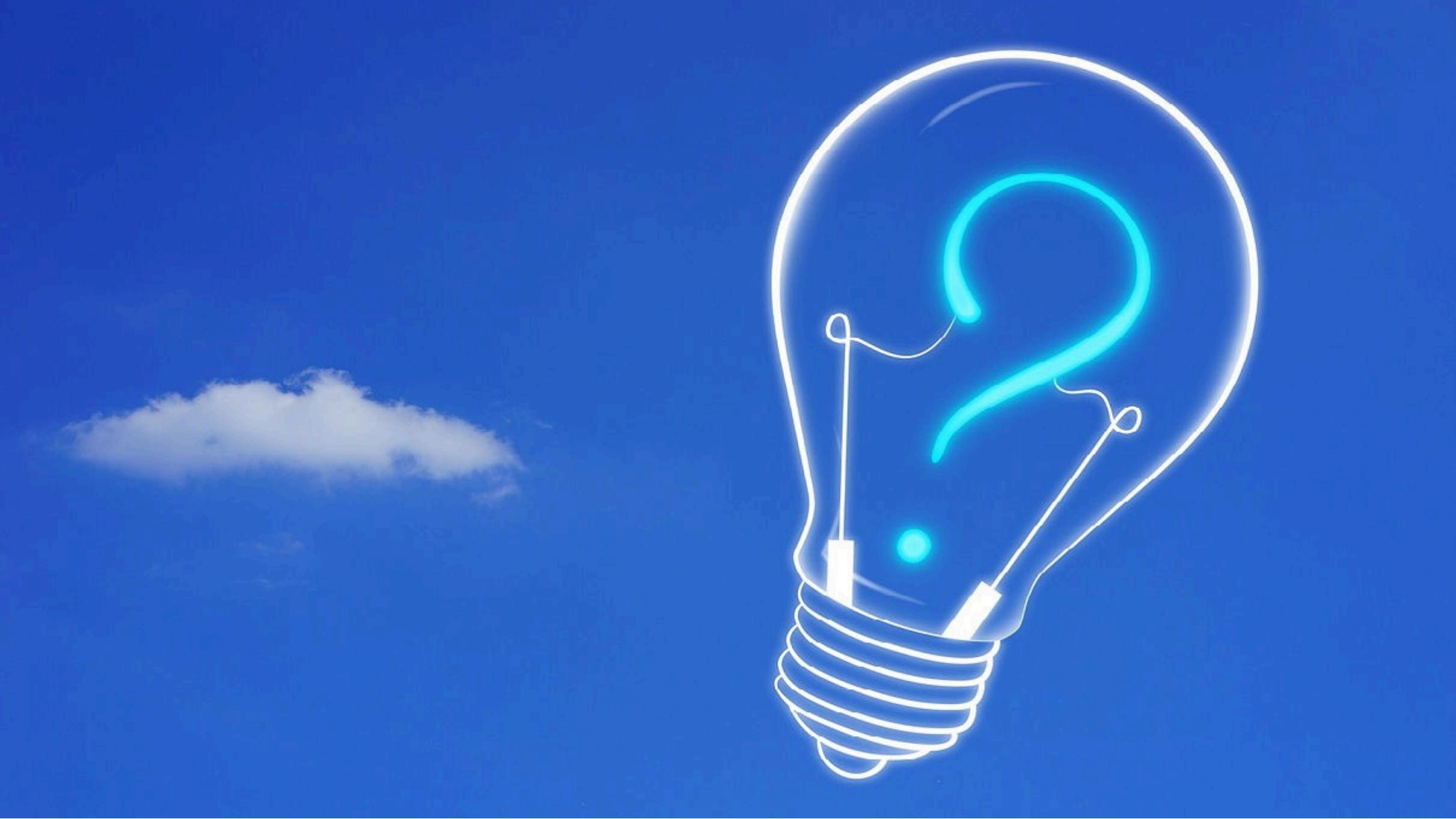
```
struct generator::promise_type {
    int current_value;

    static auto get_return_object_on_allocation_failure() { return generator{nullptr}; }

    auto get_return_object() { return generator::handle::from_promised(*this); }
    auto initial_suspend() { return std::suspend_always{}; }
    auto final_suspend() { return std::suspend_always{}; }
    void unhandled_exception() { std::terminate(); }
    void return_void() {}
    auto yield_value(int value)
    {
        current_value = value;
        return std::suspend_always{};
    }
};
```

Next meetings

DATE	PLACE	SUBJECT
15-20 Jul 2019	Cologne, Germany	CWG+LWG: Complete CD wording EWG+LEWG: Working on C++23 features + CWG/LWG design clarification questions C++20 draft wording is feature complete, start CD ballot
04-09 Nov 2019	Belfast, Northern Ireland	CD ballot comment resolution
10-15 Feb 2020	Prague, Czech Republic	CD ballot comment resolution, C++20 technically finalized, start DIS ballot
01-06 Jun 2020	Bulgaria	First meeting of C++23
Nov 2020	New York, NY, USA	
22-27 Feb 2021	Kona, HI, USA	



CAUTION
Programming
is addictive
(and too much fun)

BEYOND C++17: PART II

OPTION B: LOTS OF LESS ADVERTISED GOOD C++20 STUFF

P0732 Class Types in Non-Type Template Parameters

- Allow *non-union class types* to appear in non-type template parameters
- Require that types used as such, have **strong structural equality**
 - all of their bases and non-static data members recursively, *have a non-user-provided operator<=> returning* a type that is implicitly *convertible to std::strong_equality*, and *contain no references*

P0732 Class Types in Non-Type Template Parameters

- Allow *non-union class types* to appear in non-type template parameters
- Require that types used as such, have **strong structural equality**
 - all of their bases and non-static data members recursively, *have a non-user-provided operator<=> returning* a type that is implicitly *convertible to std::strong_equality*, and *contain no references*

P1185 <=> != ==

- Changes the **definition of the strong structural equality**
 - a type T is having strong structural equality if *each subobject recursively has defaulted == and none of the subobjects are floating point types*

P0732 Class Types in Non-Type Template Parameters

C++17

```
template<size_t seconds>
class fixed_timer { /* ... */ };
```

C++20

```
template<std::chrono::seconds seconds>
class fixed_timer { /* ... */ };
```

P0732 Class Types in Non-Type Template Parameters

C++17

```
template<size_t seconds>
class fixed_timer { /* ... */ };
```

C++17

```
template<char... Id>
class entity { /* ... */ };

entity<'h', 'e', 'l', 'l', 'o'> e;
```

C++20

```
template<std::chrono::seconds seconds>
class fixed_timer { /* ... */ };
```

C++20

```
template<fixed_string Id>
class entity { /* ... */ };

entity<"hello"> e;
```

P0732 Class Types in Non-Type Template Parameters

C++17

```
template<size_t seconds>
class fixed_timer { /* ... */ };
```

C++17

```
template<char... Id>
class entity { /* ... */ };

entity<'h', 'e', 'l', 'l', 'o'> e;
```

C++20

```
template<std::chrono::seconds seconds>
class fixed_timer { /* ... */ };
```

C++20

```
template<fixed_string Id>
class entity { /* ... */ };

entity<"hello"> e;
```

C++20

```
template<fixed_string Str>
auto operator""_udl();

"hello"_udl;
```

P0732 Class Types in Non-Type Template Parameters

C++17

```
template<Dimension D1, Unit U1, Number Rep1, Dimension D2, Unit U2, Number Rep2>
[[nodiscard]] quantity<dimension_divide_t<D1, D2>,
    upcasting_traits_t<unit<dimension_divide_t<D1, D2>,
        std::ratio_divide<typename U1::ratio, typename U2::ratio>>>,
    std::common_type_t<Rep1, Rep2>>
constexpr operator/(const quantity<D1, U1, Rep1>& lhs,
    const quantity<D2, U2, Rep2>& rhs);
```

P0732 Class Types in Non-Type Template Parameters

C++17

```
template<Dimension D1, Unit U1, Number Rep1, Dimension D2, Unit U2, Number Rep2>
[[nodiscard]] quantity<dimension_divide_t<D1, D2>,
    upcasting_traits_t<unit<dimension_divide_t<D1, D2>,
        std::ratio_divide<typename U1::ratio, typename U2::ratio>>,
        std::common_type_t<Rep1, Rep2>>
constexpr operator/(const quantity<D1, U1, Rep1>& lhs,
    const quantity<D2, U2, Rep2>& rhs);
```

C++20

```
template<Dimension D1, Unit U1, Number Rep1, Dimension D2, Unit U2, Number Rep2>
[[nodiscard]] quantity<dimension_divide_t<D1, D2>,
    upcasting_traits_t<unit<dimension_divide_t<D1, D2>, U1::ratio / U2::ratio>>,
    std::common_type_t<Rep1, Rep2>>
constexpr operator/(const quantity<D1, U1, Rep1>& lhs,
    const quantity<D2, U2, Rep2>& rhs);
```

P0732 Class Types in Non-Type Template Parameters

C++17

```
template<Dimension D1, Unit U1, Number Rep1, Dimension D2, Unit U2, Number Rep2>
[[nodiscard]] quantity<dimension_divide_t<D1, D2>,
    upcasting_traits_t<unit<dimension_divide_t<D1, D2>,
        std::ratio_divide<typename U1::ratio, typename U2::ratio>>>,
    std::common_type_t<Rep1, Rep2>>
constexpr operator/(const quantity<D1, U1, Rep1>& lhs,
    const quantity<D2, U2, Rep2>& rhs);
```

C++20

```
template<Dimension D1, Unit U1, Number Rep1, Dimension D2, Unit U2, Number Rep2>
[[nodiscard]] quantity<D1 / D2, upcasting_traits_t<unit<D1 / D2, U1::ratio / U2::ratio>>, std::common_type_t<Rep1, Rep2>>
constexpr operator/(const quantity<D1, U1, Rep1>& lhs,
    const quantity<D2, U2, Rep2>& rhs);
```

P1008 Prohibit aggregates with user-declared constructors

MOTIVATION

- The current rule of an *aggregate initialization* says that an aggregate cannot have user-provided, inherited, or explicit constructors, but explicitly deleted or defaulted constructors are allowed

P1008 Prohibit aggregates with user-declared constructors

MOTIVATION

- The current rule of an *aggregate initialization* says that an aggregate cannot have user-provided, inherited, or explicit constructors, but explicitly deleted or defaulted constructors are allowed

```
struct X {  
    X() = delete;  
};  
  
static_assert(  
    !std::is_default_constructible_v<X>);  
  
X x1; // ill-formed  
X x2{}; // compiles!
```

P1008 Prohibit aggregates with user-declared constructors

MOTIVATION

- The current rule of an *aggregate initialization* says that an aggregate cannot have user-provided, inherited, or explicit constructors, but explicitly deleted or defaulted constructors are allowed

```
struct X {  
    X() = delete;  
};  
  
static_assert(  
    !std::is_default_constructible_v<X>);  
  
X x1;    // ill-formed  
X x2{}; // compiles!
```

```
struct X {  
private:  
    X() = default;  
};  
  
static_assert(  
    !std::is_default_constructible_v<X>);  
  
X x1;    // ill-formed  
X x2{}; // compiles!
```

P1008 Prohibit aggregates with user-declared constructors

MOTIVATION

- The current rule of an *aggregate initialization* says that an aggregate cannot have user-provided, inherited, or explicit constructors, but explicitly deleted or defaulted constructors are allowed

```
struct X {  
    int i = 4;  
    X() = default;  
};  
  
X x1(3); // ill-formed  
X x2{3}; // compiles!
```

P1008 Prohibit aggregates with user-declared constructors

MOTIVATION

- The current rule of an *aggregate initialization* says that an aggregate cannot have user-provided, inherited, or explicit constructors, but explicitly deleted or defaulted constructors are allowed

```
struct X {  
    int i = 4;  
    X() = default;  
};  
  
X x1(3); // ill-formed  
X x2{3}; // compiles!
```

```
struct X {  
    int i = 4;  
    X(int) = delete;  
};  
  
X x1(3); // ill-formed  
X x2{3}; // compiles!
```

P1008 Prohibit aggregates with user-declared constructors

MOTIVATION

- The current rule of an *aggregate initialization* says that an aggregate cannot have user-provided, inherited, or explicit constructors, but explicitly deleted or defaulted constructors are allowed

```
struct X {  
    int i = 4;  
    X() = default;  
};  
  
X x1(3); // ill-formed  
X x2{3}; // compiles!
```

```
struct X {  
    int i = 4;  
    X(int) = delete;  
};  
  
X x1(3); // ill-formed  
X x2{3}; // compiles!
```

```
struct X {  
    int i;  
    X() = default;  
};  
  
X x{4}; // compiles
```

P1008 Prohibit aggregates with user-declared constructors

MOTIVATION

- The current rule of an *aggregate initialization* says that an aggregate cannot have user-provided, inherited, or explicit constructors, but explicitly deleted or defaulted constructors are allowed

```
struct X {  
    int i = 4;  
    X() = default;  
};  
  
X x1(3); // ill-formed  
X x2{3}; // compiles!
```

```
struct X {  
    int i = 4;  
    X(int) = delete;  
};  
  
X x1(3); // ill-formed  
X x2{3}; // compiles!
```

```
struct X {  
    int i;  
    X() = default;  
};  
  
X x{4}; // compiles
```

```
struct Y {  
    int i;  
    Y();  
};  
Y::Y() = default;  
  
Y y{4}; // ill-formed
```

P1008 Prohibit aggregates with user-declared constructors

SOLUTION

An aggregate is an array or a class with no ~~user-provided, explicit,~~ user-declared or inherited constructors, no private or protected non-static data members, no virtual functions, and no virtual, private, or protected base classes.

P0960 Allow initializing aggregates from a parenthesized list of values

MOTIVATION

```
struct X { X(int a, int b); /* ... */ };
struct Y { int a, b; };
```

```
X x1{1, 2}; // OK
X x2(1, 2); // OK
Y y1{1, 2}; // OK
Y y2(1, 2); // Fail
```

P0960 Allow initializing aggregates from a parenthesized list of values

SOLUTION

- Allow using `()` to initialize aggregates in addition to `{}` to unify initialization practices

P0960 Allow initializing aggregates from a parenthesized list of values

SOLUTION

- Allow using `()` to initialize aggregates in addition to `{}` to unify initialization practices
- Differences
 - for a *non-array aggregate A*
 - `A{b}` will *not convert* `b` to `A`, unless `b` is of type `A` or type derived from `A`
 - `A(b)` will *convert* even in other cases
 - `A{x, y, z}` *forbids narrowing conversions*, while the new syntax `A(x, y, z)` *does allow narrowing*
 - a *temporary bound to a reference member of the aggregate via braces has its lifetime extended*, whereas no extension happens when the temporary is passed via round parentheses

```
struct A { int a; int&& r; };
int f();
int n = 10;
```

```
A a1{1, f()};           // lifetime is extended
A a2(1, f());          // dangling reference
A a3{1.0, 1};           // error: narrowing conversion
A a4(1.0, 1);           // dangling reference
A a5(1.0, std::move(n)); // OK
```

P1064 Allowing Virtual Function Calls in Constant Expressions

MOTIVATION

- Virtual function calls are currently *prohibited in constant expressions*
- The restriction is *unnecessary and artificial*

SOLUTION

- Remove the restriction

P1064 Allowing Virtual Function Calls in Constant Expressions

FAQ

- *Can a `constexpr` virtual function override a non-`constexpr` one?*
- Yes. This is required when, for instance, the programmer has no control over the base class, or when the base virtual function is pure

P1064 Allowing Virtual Function Calls in Constant Expressions

FAQ

- *Can a constexpr virtual function override a non-constexpr one?*
- Yes. This is required when, for instance, the programmer has no control over the base class, or when the base virtual function is pure
- *Can a non-constexpr virtual function override a constexpr one?*
- Yes

P1064 Allowing Virtual Function Calls in Constant Expressions

FAQ

- *Can a constexpr virtual function override a non-constexpr one?*
- Yes. This is required when, for instance, the programmer has no control over the base class, or when the base virtual function is pure
- *Can a non-constexpr virtual function override a constexpr one?*
- Yes
- *What happens when some overriders are constexpr and some are not?*
- The final overrider is selected, as usual, and if it's not **constexpr**, the expression is not a constant expression

P1330 Changing the active member of a union inside `constexpr`

- Allows to *change the active member of a union* during compile-time code

P1330 Changing the active member of a union inside `constexpr`

- Allows to *change the active member of a union* during compile-time code

EXAMPLE

```
union Foo {  
    int i;  
    float f;  
};
```

P1330 Changing the active member of a union inside `constexpr`

- Allows to *change the active member of a union* during compile-time code

EXAMPLE

```
union Foo {  
    int i;  
    float f;  
};
```

```
constexpr int use()  
{  
    Foo foo{};  
    foo.i = 3;  
    foo.f = 1.2f;  
    return 1;  
}  
  
static_assert(use());
```

P1002 Try-catch blocks in `constexpr` functions

- Allows writing *try/catch in constexpr* code
 - behave like no-ops when the function is evaluated as a constant expression
- It *does NOT allow to throw* exceptions at compile time

P1002 Try-catch blocks in `constexpr` functions

- Allows writing *try/catch in `constexpr`* code
 - behave like no-ops when the function is evaluated as a constant expression
- It *does NOT allow to throw* exceptions at compile time

EXAMPLE

```
constexpr int f(int x)
{
    try {
        return x + 1;
    }
    catch(...) {
        return 0;
    }
}
```

P1327 Allowing `dynamic_cast`, polymorphic `typeid` in Constant Expressions

- Allows writing `dynamic_cast` and `typeid` in compile-time expressions
- Not proposed earlier because of a lack of a motivating example

P1327 Allowing `dynamic_cast`, polymorphic `typeid` in Constant Expressions

- Allows writing `dynamic_cast` and `typeid` in compile-time expressions
- Not proposed earlier because of a lack of a motivating example

```
class error_category {  
// ...  
  
// new members (P1196, P1197, P1198)  
private:  
    uint64_t id_ = 0; // exposition only  
  
protected:  
    explicit constexpr error_category(uint64_t id) noexcept;  
  
public:  
    virtual const char* message(int ev, char* buffer,  
                                size_t len) const noexcept;  
    constexpr virtual bool failed(int ev) const noexcept;  
};
```

- ABI break :-(

P1327 Allowing `dynamic_cast`, polymorphic `typeid` in Constant Expressions

- Allows writing *dynamic_cast* and *typeid* in compile-time expressions
- Not proposed earlier because of a lack of a motivating example

```
class error_category {  
// ...  
  
// new members (P1196, P1197, P1198)  
private:  
    uint64_t id_ = 0; // exposition only  
  
protected:  
    explicit constexpr error_category(uint64_t id) noexcept;  
  
public:  
    virtual const char* message(int ev, char* buffer,  
                                size_t len) const noexcept;  
    constexpr virtual bool failed(int ev) const noexcept;  
};
```

- ABI break :-)

```
class error_category_v2 : public error_category {  
    uint64_t id_ = 0; // exposition only  
  
protected:  
    explicit constexpr error_category(uint64_t id) noexcept;  
  
public:  
    virtual const char* message(int ev, char* buffer,  
                                size_t len) const noexcept;  
    constexpr virtual bool failed(int ev) const noexcept;  
};
```

- `std::error_code` could use `dynamic_cast` to check whether it has been constructed from a "new" category, and adjust accordingly

P1006 Constexpr in std::pointer_traits

```
template<class T>
struct pointer_traits<T*> {
    using pointer = T*;
    using element_type = T;
    using difference_type = ptrdiff_t;
    template<class U> using rebind = U*;

    static constexpr pointer pointer_to(see below r) noexcept;
};
```

P1006 Constexpr in std::pointer_traits

```
template<class T>
struct pointer_traits<T*> {
    using pointer = T*;
    using element_type = T;
    using difference_type = ptrdiff_t;
    template<class U> using rebind = U*;

    static constexpr pointer pointer_to(see below r) noexcept;
};
```

- Enables *std::vector* to be *constexpr* - to be used for compile-time reflection

P1006 Constexpr in std::pointer_traits

```
template<class T>
struct pointer_traits<T*> {
    using pointer = T*;
    using element_type = T;
    using difference_type = ptrdiff_t;
    template<class U> using rebind = U*;

    static constexpr pointer pointer_to(see below r) noexcept;
};
```

- Enables ***std::vector* to be `constexpr`** - to be used for compile-time reflection
- *Limited only to partial specialization of std::pointer_traits<T*>*
 - primary class template cannot be made `constexpr` because it would influence all of the specializations
 - ***std::vector*** is about to be `constexpr` with ***std::allocator<T>*** only anyway

P1006 Constexpr in std::pointer_traits

```
template<class T>
struct pointer_traits<T*> {
    using pointer = T*;
    using element_type = T;
    using difference_type = ptrdiff_t;
    template<class U> using rebind = U*;

    static constexpr pointer pointer_to(see below r) noexcept;
};
```

- Enables ***std::vector* to be `constexpr`** - to be used for compile-time reflection
- *Limited only to partial specialization of std::pointer_traits<T*>*
 - primary class template cannot be made `constexpr` because it would influence all of the specializations
 - ***std::vector*** is about to be `constexpr` with ***std::allocator<T>*** only anyway
- *Forces user-provided specializations* of ***std::pointer_traits<T*>***, where **T** is a user-defined type, to abide by the added `constexpr` requirement

P1032 Misc constexpr bits

- Adds *constexpr to missing places* in the library
 - `std::pair`
 - `std::tuple`
 - `std::array`
 - `std::char_traits`
 - `std::basic_string_view`
 - `std::default_searcher`
 - `std::back_inserter`, `std::front_inserter`, `std::inserter`
 - `std::back_insert_iterator`, `std::front_insert_iterator`, `std::insert_iterator`

P1032 Misc constexpr bits

- Adds *constexpr to missing places* in the library
 - `std::pair`
 - `std::tuple`
 - `std::array`
 - `std::char_traits`
 - `std::basic_string_view`
 - `std::default_searcher`
 - `std::back_inserter`, `std::front_inserter`, `std::inserter`
 - `std::back_insert_iterator`, `std::front_insert_iterator`, `std::insert_iterator`

`constexpr` ALL the things! :-)

P0879 Constexpr for swap and swap related functions

MOTIVATION

```
constexpr std::array<char, 6> a { 'H', 'e', 'l', 'l', 'o' };
constexpr auto it = std::find(a.begin(), a.end(), 'H'); // ERROR
```

- Algorithms that use **swap** are not **constexpr**

P0879 Constexpr for swap and swap related functions

SOLUTION

- Apply `constexpr` to `swap` and all algorithms using it

AFFECTED ALGORITHMS

- `swap_ranges`
- `iter_swap`
- `reverse`
- `rotate`
- `partition`
- `sort`
- `partial_sort`
- `partial_sort_copy`
- `nth_element`
- `push_heap`
- `pop_heap`
- `make_heap`
- `sort_heap`
- `next_permutation`
- `prev_permutation`
- `swap`

P1023 `constexpr` comparison operators for `std::array`

MOTIVATION

- *Comparison operators for `std::array`* are specified in terms of `std::equals()` and `std::lexicographical_compare()`
- Those algorithms were made `constexpr` only recently

P1023 `constexpr` comparison operators for `std::array`

MOTIVATION

- *Comparison operators for `std::array`* are specified in terms of `std::equals()` and `std::lexicographical_compare()`
- Those algorithms were made `constexpr` only recently

SOLUTION

- Add `constexpr` to comparison operators in `std::array`

P1073 Immediate functions

- The `constexpr` specifier applied to a function or member function indicates that a *call to that function might be valid in a context requiring a constant-expression*
- It **does not require** that every such call *be a constant-expression*

P1073 Immediate functions

- The `constexpr` specifier applied to a function or member function indicates that a *call to that function might be valid in a context requiring a constant-expression*
- It **does not require** that every such call *be a constant-expression*
- The proposal allows writing functions that **are guaranteed to run at compile time** when called
- Non-constant result *should produce an error*
- Such a function is called an **immediate function**

P1073 Immediate functions

- The `constexpr` specifier applied to a function or member function indicates that a *call to that function might be valid in a context requiring a constant-expression*
- It **does not require** that every such call *be a constant-expression*
- The proposal allows writing functions that **are guaranteed to run at compile time** when called
- Non-constant result *should produce an error*
- Such a function is called an **immediate function**

```
constexpr int sqr(int n) { return n * n; }

constexpr int r = sqr(100); // OK
int x = 100;
int r2 = sqr(x); // Error: Call does not produce a constant
```

P1073 Immediate functions

- The `constexpr` specifier applied to a function or member function indicates that a *call to that function might be valid in a context requiring a constant-expression*
- It **does not require** that every such call *be a constant-expression*
- The proposal allows writing functions that **are guaranteed to run at compile time** when called
- Non-constant result *should produce an error*
- Such a function is called an **immediate function**

```
consteval int sqr(int n) { return n * n; }

constexpr int r = sqr(100); // OK
int x = 100;
int r2 = sqr(x); // Error: Call does not produce a constant
```

- It is *not allowed to form pointers* to `consteval` functions

P1073 Immediate functions

- The `constexpr` specifier applied to a function or member function indicates that a *call to that function might be valid in a context requiring a constant-expression*
- It **does not require** that every such call *be a constant-expression*
- The proposal allows writing functions that **are guaranteed to run at compile time** when called
- Non-constant result *should produce an error*
- Such a function is called an **immediate function**

```
consteval int sqr(int n) { return n * n; }

constexpr int r = sqr(100); // OK
int x = 100;
int r2 = sqr(x); // Error: Call does not produce a constant
```

- It is *not allowed to form pointers* to `consteval` functions
- A basic building block for reflection and meta-classes

P0892 explicit(bool)

MOTIVATION

```
std::pair<std::string, std::string> safe() {
    return {"meow", "purr"}; // ok
}

std::pair<std::vector<int>, std::vector<int>> unsafe() {
    return {11, 22}; // error
}
```

P0892 explicit(bool)

MOTIVATION

C++17

```
template<typename T1, typename T2>
struct pair {
    template<typename U1=T1, typename U2=T2,
             std::enable_if_t<
                 std::is_constructible_v<T1, U1> &&
                 std::is_constructible_v<T2, U2> &&
                 std::is_convertible_v<U1, T1> &&
                 std::is_convertible_v<U2, T2>
             , int> = 0>
    constexpr pair(U1&&, U2&& );
    template<typename U1=T1, typename U2=T2,
             std::enable_if_t<
                 std::is_constructible_v<T1, U1> &&
                 std::is_constructible_v<T2, U2> &&
                 !(std::is_convertible_v<U1, T1> &&
                     std::is_convertible_v<U2, T2>)
             , int> = 0>
    explicit constexpr pair(U1&&, U2&& );
};
```

P0892 explicit(bool)

MOTIVATION

C++17

```
template<typename T1, typename T2>
struct pair {
    template<typename U1=T1, typename U2=T2,
             std::enable_if_t<
                 std::is_constructible_v<T1, U1> &&
                 std::is_constructible_v<T2, U2> &&
                 std::is_convertible_v<U1, T1> &&
                 std::is_convertible_v<U2, T2>
             , int> = 0>
        constexpr pair(U1&&, U2&& );
    template<typename U1=T1, typename U2=T2,
             std::enable_if_t<
                 std::is_constructible_v<T1, U1> &&
                 std::is_constructible_v<T2, U2> &&
                 !(std::is_convertible_v<U1, T1> &&
                     std::is_convertible_v<U2, T2>)
             , int> = 0>
        explicit constexpr pair(U1&&, U2&& );
};
```

WITH CONCEPTS

```
template<typename T1, typename T2>
struct pair {
    template<typename U1=T1, typename U2=T2>
        requires std::is_constructible_v<T1, U1> &&
                 std::is_constructible_v<T2, U2> &&
                 std::is_convertible_v<U1, T1> &&
                 std::is_convertible_v<U2, T2>
        constexpr pair(U1&&, U2&& );
    template<typename U1=T1, typename U2=T2>
        requires std::is_constructible_v<T1, U1> &&
                 std::is_constructible_v<T2, U2>
        explicit constexpr pair(U1&&, U2&& );
};
```

P0892 explicit(bool)

SOLUTION

C++17

```
template<typename T1, typename T2>
struct pair {
    template<typename U1=T1, typename U2=T2,
             std::enable_if_t<
                 std::is_constructible_v<T1, U1> &&
                 std::is_constructible_v<T2, U2> &&
                 std::is_convertible_v<U1, T1> &&
                 std::is_convertible_v<U2, T2>
             , int> = 0>
    constexpr pair(U1&&, U2&& );
    template<typename U1=T1, typename U2=T2,
             std::enable_if_t<
                 std::is_constructible_v<T1, U1> &&
                 std::is_constructible_v<T2, U2> &&
                 !(std::is_convertible_v<U1, T1> &&
                     std::is_convertible_v<U2, T2>)
             , int> = 0>
    explicit constexpr pair(U1&&, U2&& );
};
```

P0892 explicit(bool)

SOLUTION

C++17

```
template<typename T1, typename T2>
struct pair {
    template<typename U1=T1, typename U2=T2,
        std::enable_if_t<
            std::is_constructible_v<T1, U1> &&
            std::is_constructible_v<T2, U2> &&
            std::is_convertible_v<U1, T1> &&
            std::is_convertible_v<U2, T2>
        , int> = 0>
    constexpr pair(U1&&, U2&& );
    template<typename U1=T1, typename U2=T2,
        std::enable_if_t<
            std::is_constructible_v<T1, U1> &&
            std::is_constructible_v<T2, U2> &&
            !(std::is_convertible_v<U1, T1> &&
                std::is_convertible_v<U2, T2>)
        , int> = 0>
    explicit constexpr pair(U1&&, U2&& );
};
```

C++17 + EXPLICIT(BOOL)

```
template <typename T1, typename T2>
struct pair {
    template<typename U1=T1, typename U2=T2,
        std::enable_if_t<
            std::is_constructible_v<T1, U1> &&
            std::is_constructible_v<T2, U2>
        , int> = 0>
    explicit(!std::is_convertible_v<U1, T1> ||
        std::is_convertible_v<U2, T2>)
    constexpr pair(U1&&, U2&& );
};
```

P0892 explicit(bool)

SOLUTION

WITH CONCEPTS

```
template<typename T1, typename T2>
struct pair {
    template<typename U1=T1, typename U2=T2>
        requires std::is_constructible_v<T1, U1> &&
            std::is_constructible_v<T2, U2> &&
            std::is_convertible_v<U1, T1> &&
            std::is_convertible_v<U2, T2>
    constexpr pair(U1&&, U2&&);

    template<typename U1=T1, typename U2=T2>
        requires std::is_constructible_v<T1, U1> &&
            std::is_constructible_v<T2, U2>
    explicit constexpr pair(U1&&, U2&&);

};
```

P0892 explicit(bool)

SOLUTION

WITH CONCEPTS

```
template<typename T1, typename T2>
struct pair {
    template<typename U1=T1, typename U2=T2>
        requires std::is_constructible_v<T1, U1> &&
        std::is_constructible_v<T2, U2> &&
        std::is_convertible_v<U1, T1> &&
        std::is_convertible_v<U2, T2>
    constexpr pair(U1&&, U2&& );
    template<typename U1=T1, typename U2=T2>
        requires std::is_constructible_v<T1, U1> &&
        std::is_constructible_v<T2, U2>
    explicit constexpr pair(U1&&, U2&& );
};
```

WITH CONCEPTS + EXPLICIT(BOOL)

```
template<typename T1, typename T2>
struct pair {
    template<typename U1=T1, typename U2=T2>
        requires std::is_constructible_v<T1, U1> &&
        std::is_constructible_v<T2, U2>
        explicit(!std::is_convertible_v<U1, T1> ||
                 !std::is_convertible_v<U2, T2>)
    constexpr pair(U1&&, U2&& );
};
```

P1236 Alternative Wording for P0907R4 Signed Integers are Two's Complement

MOTIVATION

- C11 integer types allows three representations for signed integral types
 - Signed magnitude
 - Ones' complement
 - Two's complement
- C++ goes further than C and only requires that "the representations of integral types shall define values by use of a pure binary numeration system"
- All known modern computers are two's complement machines

P1236 Alternative Wording for P0907R4 Signed Integers are Two's Complement

SOLUTION

- *bool* is represented as 0 for *false* and 1 for *true* (only has value bits, no padding bits)
- **Signed integers are two's complement**
- None of the integral types have extraordinary values
- *Conversion from signed to unsigned is always well-defined*
- *Left-shift on signed integer* types produces *the same results as left-shift* on the corresponding *unsigned integer* type
- *Right-shift* is an arithmetic right shift which *performs sign-extension*
- The *range of enumerations* without a fixed underlying type is simplified because of two's complement
- **has_unique_object_representations<T>** is **true** for **bool** and signed integer types, in addition to unsigned integer types and others before

P0482 `char8_t`: A type for UTF-8 characters and strings

MOTIVATION

- C++11 introduced support for `UTF-8`, `UTF-16`, and `UTF-32` encoded string literals
 - `UTF-32` code units are hold in a new `char32_t` type
 - `UTF-16` code units are hold in a new `char16_t` type
 - `UTF-8` string literals are defined in terms of the `char` type
 - used also for the code unit type of *ordinary character and string literals*

P0482 `char8_t`: A type for UTF-8 characters and strings

MOTIVATION

- C++11 introduced support for *UTF-8*, *UTF-16*, and *UTF-32* encoded string literals
 - *UTF-32* code units are hold in a new `char32_t` type
 - *UTF-16* code units are hold in a new `char16_t` type
 - *UTF-8* string literals are defined in terms of the `char` type
 - used also for the code unit type of *ordinary character and string literals*
- Lack of a distinct type for UTF-8 encoded character and string literals *prevents the use of overloading and template specialization* in interfaces designed for interoperability with encoded text

P0482 `char8_t`: A type for UTF-8 characters and strings

MOTIVATION

- C++11 introduced support for *UTF-8*, *UTF-16*, and *UTF-32* encoded string literals
 - *UTF-32* code units are held in a new `char32_t` type
 - *UTF-16* code units are held in a new `char16_t` type
 - *UTF-8* string literals are defined in terms of the `char` type
 - used also for the code unit type of *ordinary character and string literals*
- Lack of a distinct type for *UTF-8* encoded character and string literals *prevents the use of overloading and template specialization* in interfaces designed for interoperability with encoded text
- Whether `char` is a signed or unsigned type is implementation defined
 - implementations that use an *8-bit signed char are at a disadvantage* with respect to working with *UTF-8* encoded text due to the necessity of having to rely on *conversions to unsigned types* in order to correctly process leading and continuation code units of multi-byte encoded code points

P0482 `char8_t`: A type for UTF-8 characters and strings

- `\u0123` (latin small letter g with cedilla (U+0123))

	// Encoding:	Code unit type:	Code unit values:
<code>u8"\u0123"</code>	// UTF-8	<code>const char[]</code>	<code>0xC4 0xA3 0x00</code>
<code>u"\u0123"</code>	// UTF-16	<code>const char16_t[]</code>	<code>0x0123 0x0000</code>
<code>U"\u0123"</code>	// UTF-32	<code>const char32_t[]</code>	<code>0x00000123 0x00000000</code>
<code>"\u0123"</code>	// ???	<code>const char[]</code>	???
<code>L"\u0123"</code>	// ???	<code>const wchar_t[]</code>	???

P0482 `char8_t`: A type for UTF-8 characters and strings

- `\u0123` (latin small letter g with cedilla (U+0123))

	// Encoding:	Code unit type:	Code unit values:
<code>u8"\u0123"</code>	// UTF-8	<code>const char[]</code>	<code>0xC4 0xA3 0x00</code>
<code>u"\u0123"</code>	// UTF-16	<code>const char16_t[]</code>	<code>0x0123 0x0000</code>
<code>U"\u0123"</code>	// UTF-32	<code>const char32_t[]</code>	<code>0x00000123 0x00000000</code>
<code>"\u0123"</code>	// ???	<code>const char[]</code>	???
<code>L"\u0123"</code>	// ???	<code>const wchar_t[]</code>	???

- Differentiation via function overloading

```
void do_x(const char*);  
void do_x_utf8(const char*);  
void do_x(const wchar_t*);  
void do_x(const char16_t*);  
void do_x(const char32_t*);
```

P0482 `char8_t`: A type for UTF-8 characters and strings

STD::FILESYSTEM::PATH

```
template<class Source>
path(const Source& source);

template<class InputIterator>
path(InputIterator first, InputIterator last);
```

- `char32_t` => UTF-32, `char16_t` => UTF-16, `char & wchar_t` => implementation defined encoding
- Not possible to construct a path object from UTF-8 encoded text using these constructors

P0482 `char8_t`: A type for UTF-8 characters and strings

STD::FILESYSTEM::PATH

```
template<class Source>
path(const Source& source);

template<class InputIterator>
path(InputIterator first, InputIterator last);
```

- `char32_t` => UTF-32, `char16_t` => UTF-16, `char & wchar_t` => implementation defined encoding
- Not possible to construct a path object from UTF-8 encoded text using these constructors

```
template<class Source>
path u8path(const Source& source);

template<class InputIterator>
path u8path(InputIterator first, InputIterator last);
```

- Factory functions are not provided for other encodings

P0482 `char8_t`: A type for UTF-8 characters and strings

CODECVT

```
codecvt<char, char, mbstate_t> // performs no conversions  
codecvt<wchar_t, char, mbstate_t> // converts between the implementation defined encodings  
codecvt<char16_t, char, mbstate_t> // converts between the UTF-16 and the UTF-8 encodings  
codecvt<char32_t, char, mbstate_t> // converts between the UTF-32 and the UTF-8 encodings
```

- Specializations are *not currently specified for conversion between the implementation defined narrow and wide encodings and any of the UTF-8, UTF-16, or UTF-32 encodings*
- If support for such conversions were to be added, the desired interfaces are already taken

P0482 `char8_t`: A type for UTF-8 characters and strings

SOLUTION: CORE LANGUAGE

- A new fundamental type named `char8_t`
 - same signedness, size, alignment, and integer conversion rank as `unsigned char`
 - does not alias with any other type
- The type of *UTF-8 string literals* is changed from array of `const char` to array of `const char8_t`
- The type of *UTF-8 character literals* is changed from `char` to `char8_t`
- New `char8_t` based signatures for *user-defined literal operators*

P0482 `char8_t`: A type for UTF-8 characters and strings

SOLUTION: STANDARD LIBRARY

- New `char8_t` based *specializations* of `atomic`, `numeric_limits`, `hash`, `char_traits`, `basic_string`, and `basic_string_view`
- New `u8streampos`, `u8string`, `u8string_view` *type aliases*
- New `operator ""s` and `operator ""sv` `char8_t` based *overloads for UTF-8 literals*
- New `char8_t` based *specializations of codecvt* and *codecvt_byname*
 - the existing `char` based specializations are deprecated
- The *return type of the u8string and generic_u8string* member functions of the filesystem `path` class are changed from `string` to `u8string`
- Filesystem `path` objects may now be constructed with UTF-8 strings *using existing constructors*
 - the existing `u8path` factory functions are deprecated

P0482 `char8_t`: A type for UTF-8 characters and strings

CORE LANGUAGE BACKWARD COMPATIBILITY

- Initialization

```
char ca[] = u8"text"; // C++17: OK  
                      // C++20: Ill-formed
```

```
char8_t c8a[] = "text"; // C++17: N/A (char8_t is not a type specifier)  
                      // C++20: Ill-formed
```

P0482 `char8_t`: A type for UTF-8 characters and strings

CORE LANGUAGE BACKWARD COMPATIBILITY

- Initialization

```
char ca[] = u8"text";    // C++17: OK
                           // C++20: Ill-formed

char8_t c8a[] = "text"; // C++17: N/A (char8_t is not a type specifier)
                           // C++20: Ill-formed
```

```
char c = u8'c';          // C++17: OK
                           // C++20: OK (no change from C++17)

char8_t c8 = 'c';        // C++17: N/A (char8_t is not a type specifier)
                           // C++20: OK; c8 is assigned the value of the 'c'
                           //           character in the execution character set
```

P0482 `char8_t`: A type for UTF-8 characters and strings

CORE LANGUAGE BACKWARD COMPATIBILITY

- Implicit conversions

```
const char (&u8r)[] = u8"text"; // C++17: OK
                                // C++20: Ill-formed

const char *u8p = u8"text";    // C++17: OK
                                // C++20: Ill-formed
```

P0482 `char8_t`: A type for UTF-8 characters and strings

CORE LANGUAGE BACKWARD COMPATIBILITY

- Type deduction

```
template<typename T1, typename T2>
void ft(T1, T2);

ft(u8"text", u8'c'); // C++17: T1 deduced to const char*, T2 deduced to char
                      // C++20: T1 deduced to const char8_t*, T2 deduced to char8_t

auto u8p = u8"text"; // C++17: Type deduced to const char*
                      // C++20: Type deduced to const char8_t*

auto u8c = u8'c';   // C++17: Type deduced to char
                      // C++20: Type deduced to char8_t
```

P0482 `char8_t`: A type for UTF-8 characters and strings

CORE LANGUAGE BACKWARD COMPATIBILITY

- Overload resolution

```
template<typename T> void f(const T*);  
void f(const char*);  
  
f(u8"text");           // C++17: Calls f(const char*)  
                      // C++20: Calls f<char8_t>(const char8_t*)
```

P0482 `char8_t`: A type for UTF-8 characters and strings

CORE LANGUAGE BACKWARD COMPATIBILITY

- Overload resolution

```
template<typename T> void f(const T*);  
void f(const char*);  
  
f(u8"text");           // C++17: Calls f(const char*)  
                      // C++20: Calls f<char8_t>(const char8_t*)
```

```
void f(const char*);  
f(u8"text");           // C++17: OK  
                      // C++20: Ill-formed; no matching function found  
  
int operator ""_udl(const char*, size_t);  
auto x = u8"text"_udl;      // C++17: OK  
                           // C++20: Ill-formed; no matching literal operator found
```

P0482 `char8_t`: A type for UTF-8 characters and strings

CORE LANGUAGE BACKWARD COMPATIBILITY

- Template specialization

```
template<typename T>
struct ct { static constexpr bool value = false; };

template<>
struct ct<char> { static constexpr bool value = true; };

template<typename T>
bool ft(const T*) { return ct<T>::value; }

ft(u8"text");           // C++17: returns true
                      // C++20: returns false
```

P0482 `char8_t`: A type for UTF-8 characters and strings

STANDARD LIBRARY BACKWARD COMPATIBILITY

- Return type of `path::u8string` and `path::generic_u8string`

```
void f(std::filesystem::path p)
{
    std::string s;
    s = p.u8string(); // C++17: OK
                      // C++20: ill-formed
}
```

P0482 `char8_t`: A type for UTF-8 characters and strings

STANDARD LIBRARY BACKWARD COMPATIBILITY

- Return type of `path::u8string` and `path::generic_u8string`

```
void f(std::filesystem::path p)
{
    std::string s;
    s = p.u8string(); // C++17: OK
                      // C++20: ill-formed
}
```

- Return type of `operator ""s` and `operator ""sv`

```
std::string s;
s = u8"text"s;    // C++17: OK
                  // C++20: ill-formed
s = u8"text"sv;  // C++17: OK
                  // C++20: ill-formed
```

P1094 Nested Inline Namespaces

C++17

```
namespace std::experimental {
    inline namespace parallelism_v2 {
        namespace execution {
            // 5.7, Unsequenced execution policy
            class unsequenced_policy;

            // 5.8, Vector execution policy
            class vector_policy;

            // 5.10, execution policy objects
            inline constexpr sequenced_policy seq{ unspecified };
            inline constexpr parallel_policy par{ unspecified };

        }
    }
}
```

P1094 Nested Inline Namespaces

C++20

```
namespace std::experimental::inline_parallelism_v2::execution {
    // 5.7, Unsequenced execution policy
    class unsequenced_policy;

    // 5.8, Vector execution policy
    class vector_policy;

    // 5.10, execution policy objects
    inline constexpr sequenced_policy seq{ unspecified };
    inline constexpr parallel_policy par{ unspecified };
}
```

P1091 Extending structured bindings to be more like variable declarations

MOTIVATION

- Structure binding is not a variable declaration
- Bindings do not have any linkage because they're just names
- A lot of *restrictions on structured bindings compared to variable declarations*
 - not being able to mark them **static**
 - not being able to mark them **constexpr**
 - bindings cannot be captured by lambda expression
 - not able to apply **[[maybe_unused]]**

P1091 Extending structured bindings to be more like variable declarations

SOLUTION

- Linkage - external
- Make `extern`, `static`, `thread_local`, `inline`, and `constexpr` work
- Allow lambda captures to refer to structured bindings
- Allow `[[maybe_unused]]` usage in a structured binding declaration

P0806 Deprecate implicit capture of **this** via [=]

C++17

```
struct Foo {  
    int n = 0;  
    void f(int a) {  
        g([=](int k) { return n + a * k; });  
        g([=, *this](int k) { return n + a * k; });  
        g([&, a](int k) { n += a * k; });  
    }  
};
```

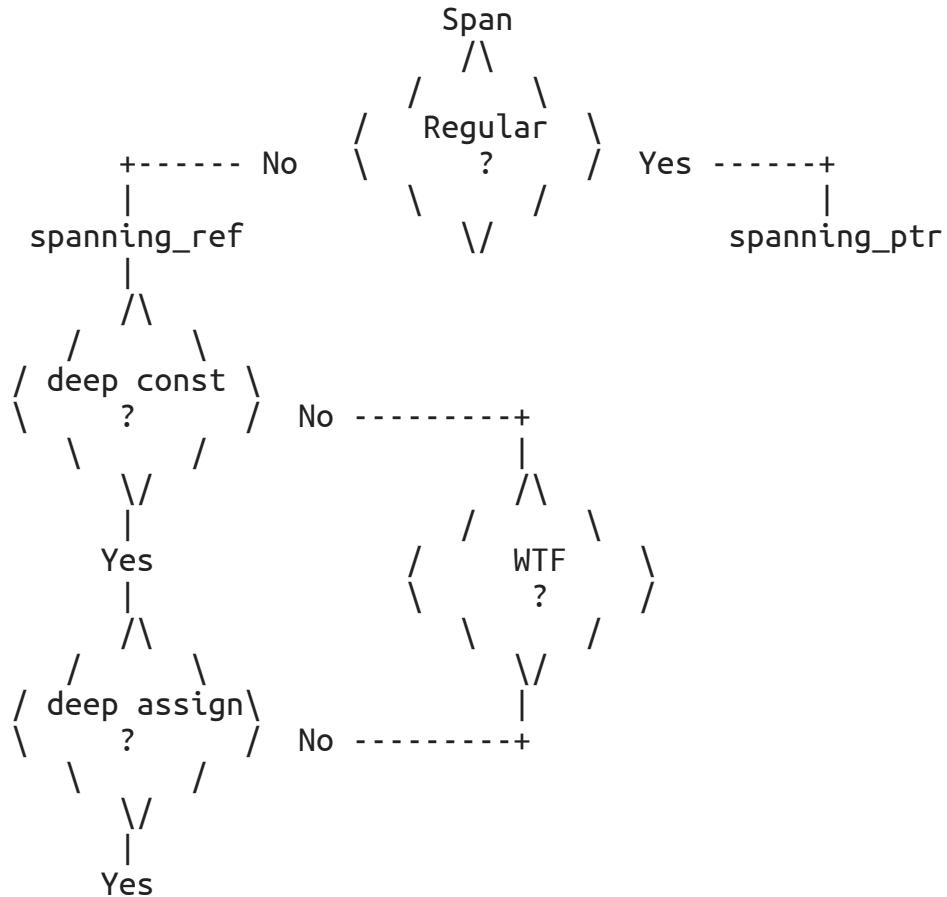
C++20

```
struct Foo {  
    int n = 0;  
    void f(int a) {  
        g([=, this](int k) { return n + a * k; });  
        g([=, *this](int k) { return n + a * k; });  
        g([&, a](int k) { n += a * k; });  
    }  
};
```

Capturing of **this** is deprecated only for [=], [&] still captures **this**

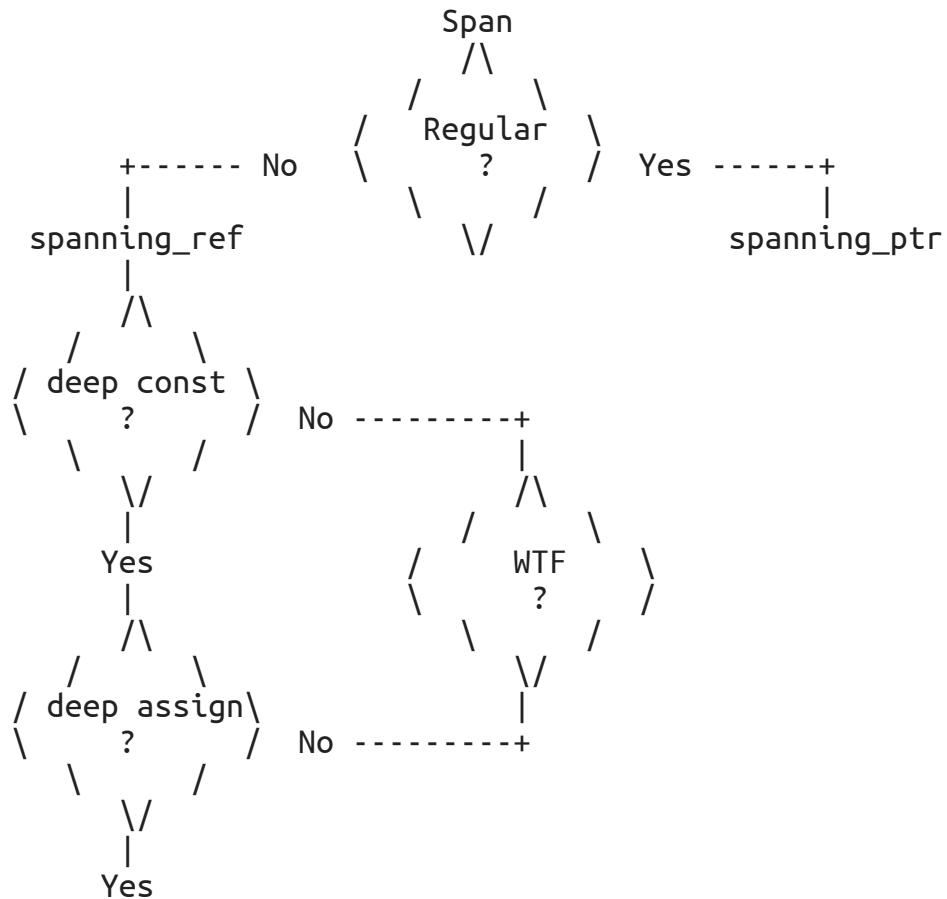
P1085 Should Span be Regular?

MOTIVATION



P1085 Should Span be Regular?

MOTIVATION



Copy or copy not; there is no shallow.

- Master Yoda

- **int** is Regular
- Pointers are Regular
- **span** currently is not

P1085 Should Span be Regular?

- `span` is currently *not Regular*
 - copy is "shallow" (only pointer and length are copied)
 - `operator==` is "deep" (`std::equal`)

P1085 Should Span be Regular?

- `span` is currently *not Regular*
 - copy is "shallow" (only pointer and length are copied)
 - `operator==` is "deep" (`std::equal`)
- This matches `string_view`, however *`string_view` can't modify the elements* it points at
 - the shallow copy of `string_view` can be thought of as similar to a copy-on-write optimization

P1085 Should Span be Regular?

- `span` is currently *not Regular*
 - copy is "shallow" (only pointer and length are copied)
 - `operator==` is "deep" (`std::equal`)
- This matches `string_view`, however *`string_view` can't modify the elements* it points at
 - the shallow copy of `string_view` can be thought of as similar to a copy-on-write optimization
- *May not work as expected compared to most types*, when a novice copy/pastes code patterns and then applies them to `span`, or *when using `span` in a generic template*

P1085 Should Span be Regular?

```
void read_only(const T & x);

void f()
{
    T tmp = x;
    read_only(x);
    assert(tmp == x); // can easily fail for span
}
```

P1085 Should Span be Regular?

```
void read_only(const T & x);

void f()
{
    T tmp = x;
    read_only(x);
    assert(tmp == x); // can easily fail for span
}
```

```
void g()
{
    const span const_sp = some_span();
    span cp = const_sp; // makes a "copy", and is non-const
    cp[1] = 17; // this also changes value of const_sp
}
```

P1085 Should Span be Regular?

SOLUTION

- Remove all the comparison operators from `span`

P1024 Usability Enhancements for std::span

- Add **front()** and **back()** member functions
- Mark **empty()** as **[[nodiscard]]**
- Remove **operator()** and just leave **operator[]**
- Structured bindings support for **fixed-size** spans (**span<T, N>** as a drop-in replacement for **T (&)[N]** or **std::array<T, N>**)

P1227 Signed `ssize()` functions, unsigned `size()` functions (Revision 2)

SIZE_TYPE SHOULD BE A SIGNED INTEGER

P1227 Signed `ssize()` functions, unsigned `size()` functions (Revision 2)

SIZE_TYPE SHOULD BE A SIGNED INTEGER

```
class MyMessageHeader {};

void handleMessage(span<const char> message)
{
    // Warning: Comparison of signed and unsigned types
    if(message.size() >= sizeof(MyMessageHeader)) {
        const MyMessageHeader* hdr = reinterpret_cast<const MyMessageHeader*>(message.data());
    }
}
```

P1227 Signed `ssize()` functions, unsigned `size()` functions (Revision 2)

SIZE_TYPE SHOULD BE A SIGNED INTEGER

```
class MyMessageHeader {};

void handleMessage(span<const char> message)
{
    // Warning: Comparison of signed and unsigned types
    if(message.size() >= sizeof(MyMessageHeader)) {
        const MyMessageHeader* hdr = reinterpret_cast<const MyMessageHeader*>(message.data());
    }
}
```

SIZE_TYPE SHOULD BE AN UNSIGNED INTEGER

P1227 Signed `ssize()` functions, unsigned `size()` functions (Revision 2)

SIZE_TYPE SHOULD BE A SIGNED INTEGER

```
class MyMessageHeader {};

void handleMessage(span<const char> message)
{
    // Warning: Comparison of signed and unsigned types
    if(message.size() >= sizeof(MyMessageHeader)) {
        const MyMessageHeader* hdr = reinterpret_cast<const MyMessageHeader*>(message.data());
    }
}
```

SIZE_TYPE SHOULD BE AN UNSIGNED INTEGER

```
template <typename T>
bool has_repeated_values(const T& container)
{
    // Warning: Comparison of signed and unsigned types
    for(int i = 0; i < container.size() - 1; ++i) {
        if(container[i] == container[i + 1]) return true;
    }
    return false;
}
```

P1227 Signed `ssize()` functions, unsigned `size()` functions (Revision 2)

SOLUTION

- Use unsigned `size()` for `std::span` similarly to all other containers from the C++ Standard Library
- Allow using signed sizes and indexes via non-member `ssize()` function to get the benefits of signedness

```
template <typename T>
bool has_repeated_values(const T& container)
{
    using std::ssize;
    for(ptrdiff_t i = 0; i < ssize(container) - 1; ++i) {
        if(container[i] == container[i + 1]) return true;
    }
    return false;
}
```

P0458 Checking for Existence of an Element in Associative Containers

MOTIVATION

- The common idiom for checking whether an element exists in an associative container

```
if(some_set.find(element) != some_set.end()) {  
    // ...  
}
```

- Excessive boilerplate
- Not clearly expresses the intent
- Not obvious to beginners
- Using **count()** member function is suboptimal for **std::multiset** and **std::multimap**

P0458 Checking for Existence of an Element in Associative Containers

SOLUTION

```
if(some_set.contains(element)) {  
    // ...  
}
```

- Added to
 - `std::map`
 - `std::multimap`
 - `std::set`
 - `std::multiset`
 - `std::unordered_map`
 - `std::unordered_multimap`
 - `std::unordered_set`
 - `std::unordered_multiset`

P0919 Heterogeneous lookup for unordered containers

C++17

```
std::unordered_map<std::string, int> map = /* ... */;
auto it1 = map.find("abc");
auto it2 = map.find("def"sv);
```

P0919 Heterogeneous lookup for unordered containers

C++17

```
std::unordered_map<std::string, int> map = /* ... */;
auto it1 = map.find("abc");
auto it2 = map.find("def"sv);
```

C++20

```
struct string_hash {
    using transparent_key_equal = std::equal_to<>; // Pred to use
    using hash_type = std::hash<std::string_view>; // just a helper local type
    size_t operator()(std::string_view txt) const { return hash_type{}(txt); }
    size_t operator()(const std::string& txt) const { return hash_type{}(txt); }
    size_t operator()(const char* txt) const { return hash_type{}(txt); }
};

std::unordered_map<std::string, int, string_hash> map = /* ... */;
map.find("abc");
map.find("def"sv);
```

P0920 Precalculated hash values in lookup

```
std::array<std::unordered_map<std::string, int>, array_size> maps;

void update(const std::string& user)
{
    const auto hash = maps.front().hash_function()(user);
    for(auto& m : maps) {
        auto it = m.find(user, hash);
        // ...
    }
}
```

P1209 Adopt Consistent Container Erasure from Library Fundamentals 2 for C++20

```
template <class T, class A, class Predicate>
void erase_if(vector<T, A>& c, Predicate pred);
```

```
template <class T, class A, class U>
void erase(vector<T, A>& c, const U& value);
```

- Those and similar overloads added for *every container* in the C++ Standard Library

P1001 Target Vectorization Policies from Parallelism V2 TS to C++20

- Adds a *new execution policy* type indicating that a parallel algorithm's execution **may be vectorized** on one thread of execution

```
namespace std::execution {
    class unsequenced_policy;
    inline constexpr unsequenced_policy unseq{ unspecified };
}
```

- If the invocation of an element access function with **unseq** *exists via an uncaught exception, terminate() shall be called*

P0769 Add shift to <algorithm>

MOTIVATION

- *No easy way to shift elements in a range* which is an important use case in time series analysis algorithms used in scientific and financial applications

P0769 Add shift to <algorithm>

MOTIVATION

- *No easy way to shift elements in a range* which is an important use case in time series analysis algorithms used in scientific and financial applications
- Implementation which uses `std::move` to implement `shift_left` for forward iterators and `std::move_backward` to implement `shift_right` for bidirectional iterators *is suboptimal since elements are guaranteed to be moved within the same range*, not between two different ranges

P0769 Add shift to <algorithm>

MOTIVATION

- *No easy way to shift elements in a range* which is an important use case in time series analysis algorithms used in scientific and financial applications
- Implementation which uses `std::move` to implement `shift_left` for forward iterators and `std::move_backward` to implement `shift_right` for bidirectional iterators *is suboptimal since elements are guaranteed to be moved within the same range*, not between two different ranges
- Comparing to `std::rotate`, shifting just the desired elements would allow for both a *more efficient implementation and clearer semantics* in case rotation is not needed

P0769 Add shift to <algorithm>

SOLUTION

```
template<class ForwardIterator>
constexpr ForwardIterator shift_left(ForwardIterator first, ForwardIterator last,
                                    typename iterator_traits<ForwardIterator>::difference_type n);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator shift_left(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                          typename iterator_traits<ForwardIterator>::difference_type n);

template<class ForwardIterator>
ForwardIterator shift_right(ForwardIterator first, ForwardIterator last,
                           typename iterator_traits<ForwardIterator>::difference_type n);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator shift_right(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                           typename iterator_traits<ForwardIterator>::difference_type n);
```

P0646 Improving the Return Value of Erase-Like Algorithms

MOTIVATION

```
template<class Key, class T, class Compare = std::less<Key>,
         class Allocator = std::allocator<std::pair<const Key, T>>>
class map {
public:
    size_type erase(const key_type& key); // Removes all elements with the key equivalent to key
    // ...
};
```

```
template<class T, class Allocator = std::allocator<T>>
class list {
public:
    void remove(const T& value);           // Removes all elements satisfying specific criteria
    // ..
};
```

P0646 Improving the Return Value of Erase-Like Algorithms

SOLUTION

- *Change the return type* of the `remove()`, `remove_if()`, and `unique()` members of `forward_list` and `list` from `void` to `container::size` type
- Above functions *return the number of elements removed*

P0556 Integral power-of-2 operations

- Adds free functions for operations related to integral powers of 2 on all *unsigned integer* types
- Added to `<bit>` header

P0556 Integral power-of-2 operations

- Adds free functions for operations related to integral powers of 2 on all *unsigned integer* types
- Added to `<bit>` header

```
template<class T>
constexpr bool ispow2(T x) noexcept;
```

- **true** if **x** is an integral power of two; **false** otherwise

```
template <class T>
constexpr T ceil2(T x) noexcept;
```

- The minimal value **y** such that **ispow2(y)** is true and **y >= x**; if **y** is not representable as a value of type **T**, the result is an unspecified value

```
template <class T>
constexpr T floor2(T x) noexcept;
```

- If **x == 0, 0**; otherwise the maximal value **y** such that **ispow2(y)** is **true** and **y <= x**

```
template <class T>
constexpr T log2p1(T x) noexcept;
```

- If **x == 0, 0**; otherwise one plus the base-2 logarithm of **x**, with any fractional part discarded

P0476 Bit-casting object representations

```
namespace std {  
    template<typename To, typename From>  
        constexpr To bit_cast(const From& from) noexcept;  
}
```

MOTIVATION

- Interpreting objects of one type as another (keep the same bits, but obtain an object of a different type) using `reinterpret_cast` or `union` runs afoul of *type-aliasing* rules
- Proper implementation use `aligned_storage` with `memcpy`, avoiding alignment pitfalls and allowing them to bit-cast non-default-constructible types

P0476 Bit-casting object representations

```
namespace std {  
    template<typename To, typename From>  
        constexpr To bit_cast(const From& from) noexcept;  
}
```

MOTIVATION

- Interpreting objects of one type as another (keep the same bits, but obtain an object of a different type) using `reinterpret_cast` or `union` runs afoul of *type-aliasing* rules
- Proper implementation use `aligned_storage` with `memcpy`, avoiding alignment pitfalls and allowing them to bit-cast non-default-constructible types

Bikeshedding included the names like `bit_cast`, `memory_cast` and...
`pod_cast` ;-)

P0476 Bit-casting object representations

```
namespace std {  
    template<typename To, typename From>  
        constexpr To bit_cast(const From& from) noexcept;  
}
```

SOLUTION

- Does not participate in overload resolution unless
 - `sizeof(To) == sizeof(From)`
 - `std::is_trivially_copyable_v<To>`
 - `std::is_trivially_copyable_v<From>`
- Copies bits of an object in a consistent and simple manner
- Adds `std::bit_cast` in `<bit>` header for trivially-copyable objects
- Similar to `memcpy` but safer and can run at compile-time (`constexpr`)

P0528 The Curious Case of Padding Bits, Featuring Atomic Compare-and-Exchange

MOTIVATION

- Currently the effect of `atomic_compare_exchange_strong` is defined in terms of `object representation` (contents of the memory)

```
if(memcmp(object, expected, sizeof(*object)) == 0)
    memcpy(object, &desired, sizeof(*object));
else
    memcpy(expected, object, sizeof(*object));
```

P0528 The Curious Case of Padding Bits, Featuring Atomic Compare-and-Exchange

MOTIVATION

- Currently the effect of `atomic_compare_exchange_strong` is defined in terms of **object representation** (contents of the memory)

```
if(memcmp(object, expected, sizeof(*object)) == 0)
    memcpy(object, &desired, sizeof(*object));
else
    memcpy(expected, object, sizeof(*object));
```

- Following code may not compare equal because of different values in padding bits

```
struct padded {
    char c = 0x42; // padded
    unsigned v = 0xC0DEFEFE;
};
atomic<padded> pad = ATOMIC_VAR_INIT({});
```

```
bool compare()
{
    padded p1, p2{0, 0};
    return pad.compare_exchange_strong(p1, p2);
}
```

P0528 The Curious Case of Padding Bits, Featuring Atomic Compare-and-Exchange

SOLUTION

- Change `atomic_compare_exchange_strong` to be defined in terms of `value representation`
- *Padding bits* that never participate in the object's value representation *are ignored*
- For a *union with bits that participate* in the value representation of some members but not others, compare-and-exchange *might always fail*

```
union pony {
    double celestia = 0.;
    short luna; // padded
};
atomic<pony> princesses = ATOMIC_VAR_INIT({});

bool party(pony desired) {
    pony expected;
    return princesses.compare_exchange_strong(expected, desired);
}
```

P0019 Atomic Ref

MOTIVATION

Extension to the atomic operations library to *allow atomic operations to apply to non-atomic objects*

P0019 Atomic Ref

SOLUTION

```
template<class T>
struct atomic_ref<T*>;
```

- *All atomic operations* performed through an atomic reference on a referenced non-atomic object *are atomic with respect to* any other atomic reference that references *the same object*
- Atomic reference objects are *not lock free* and *not address free*
- *T must be trivially copyable*
- Interface similar to **std::atomic<T>**
- *Specializations for integral and floating-point types*
- *Partial specialization for pointers*

P0935 Eradicating unnecessarily explicit default constructors from the standard library

MOTIVATION

- *Explicit default constructors block copy-list-initialization from {}*
- Most explicit default constructors in the library appear to arise out of *historical accident* rather than intentional design
 - they are *constructors taking one or more parameters*, all of which *have default arguments*

```
struct tagged_queue {  
    std::string tag;  
    std::queue<int> queue;  
};  
tagged_queue q = { "queue1" }; // error
```

P0935 Eradicating unnecessarily explicit default constructors from the standard library

SOLUTION

- Fix default construction of
 - container adapters (`queue`, `priority_queue`, `stack`)
 - random number engines
 - random number distributions
 - `basic_stringbuf`
 - `basic_istringstream`
 - `basic_ostringstream`
 - `basic_stringstream`
 - `match_results`
 - `strstreambuf`
 - `wstring_convert`
 - `wbuffer_convert`

P0608 A sane variant converting constructor

MOTIVATION

```
variant<string, bool> x = "abc";           // holds bool
```

P0608 A sane variant converting constructor

MOTIVATION

```
variant<string, bool> x = "abc";           // holds bool
```

```
variant<char, optional<char16_t>> x = u'\u2043'; // holds char = 'C'  
double d = 3.14;  
variant<int, reference_wrapper<double>> y = d;    // holds int = 3
```

P0608 A sane variant converting constructor

MOTIVATION

```
using T = variant<float, int>;
T v;
v = 0;    // switches to int
```

P0608 A sane variant converting constructor

MOTIVATION

```
using T = variant<float, int>;
T v;
v = 0;      // switches to int
```

```
using T = variant<float, long>;
T v;
v = 0;      // error
```

P0608 A sane variant converting constructor

MOTIVATION

```
using T = variant<float, int>;
T v;
v = 0;      // switches to int
```

```
using T = variant<float, long>;
T v;
v = 0;      // error
```

```
using T = variant<float, big_int<256>>;
T v;
v = 0;      // holds 0.f
```

P0608 A sane variant converting constructor

SOLUTION

- Constrains the **variant** converting constructor and the converting assignment operator to *prevent narrowing conversions and conversions to bool*

P0655 visit<R>: Explicit Return Type for visit

MOTIVATION

- Variant visitation requires invocation of *all combinations of alternatives to result in the same type*
- This type is *deduced as the visitation return type*

```
struct process {
    template<typename I>
    auto operator()(I i) -> O<I> { /* ... */ };
};

std::variant<I1, I2> input = /* ... */;

auto output = std::visit(process{}, input); // ERROR
```

P0655 visit<R>: Explicit Return Type for visit

SOLUTION

```
struct process {
    template<typename I>
    auto operator()(I i) -> 0<I> { /* ... */ };
};

std::variant<I1, I2> input = /* ... */;

// mapping from a variant of inputs to a variant of results
auto output = std::visit<std::variant<0<I1>, 0<I2>>>(process{}, input);

// coercing different results to a common type
auto result = std::visit<std::common_type_t<0<I1>, 0<I2>>>(process{}, input);

// visiting a variant for the side-effects, discarding results:
std::visit<void>(process{}, input);
```

P1007 `std::assume_aligned`

MOTIVATION

- It is possible to have *data allocated at an over-aligned memory address*
 - i.e. obtained from the `std::align_val_t` version of `operator new`, or from `std::align`

P1007 std::assume_aligned

MOTIVATION

- It is possible to have *data allocated at an over-aligned memory address*
 - i.e. obtained from the `std::align_val_t` version of `operator new`, or from `std::align`

```
void mult(float* x, int size, float factor)
{
    for(int i = 0; i < size; ++i)
        x[i] *= factor;
}
```

- Often, the compiler will auto-vectorise a loop over a buffer with contiguous data, generating SIMD instructions
 - *prolog*
 - fast vectorized SIMD loop
 - *epilog*

P1007 std::assume_aligned

SOLUTION

```
template<size_t N, class T>
[[nodiscard]] constexpr T* assume_aligned(T* ptr);
```

- Takes a pointer and *returns it unchanged*
- Allows the *compiler to assume that the pointer returned is aligned* to at least **N** bytes

P1007 std::assume_aligned

SOLUTION

```
template<size_t N, class T>
[[nodiscard]] constexpr T* assume_aligned(T* ptr);
```

- Takes a pointer and *returns it unchanged*
- Allows the *compiler to assume that the pointer returned is aligned* to at least **N** bytes
- If the pointer passed in is *not aligned to at least N bytes*, calling **assume_aligned** results in *undefined behaviour*
- If **N** is *not an alignment requirement* (a power of two), the program is *ill-formed*

P1007 std::assume_aligned

SOLUTION

```
template<size_t N, class T>
[[nodiscard]] constexpr T* assume_aligned(T* ptr);
```

- Takes a pointer and *returns it unchanged*
- Allows the *compiler to assume that the pointer returned is aligned* to at least **N** bytes
- If the pointer passed in is *not aligned to at least N bytes*, calling **assume_aligned** results in *undefined behaviour*
- If **N** is *not an alignment requirement* (a power of two), the program is *ill-formed*
- **Allows the compiler to generate better-optimised code**

P1007 std::assume_aligned

C++17

```
void mult(float* x, int size, float factor)
{
    for(int i = 0; i < size; ++i)
        x[i] *= factor;
}
```

C++20

```
void mult(float* x, int size, float factor)
{
    float* ax = std::assume_aligned<64>(x);
    for(int i = 0; i < size; ++i)
        ax[i] *= factor;
}
```

P1020 Smart pointer creation with default initialization

MOTIVATION

```
int* foo(size_t size)
{
    int* ptr = new int[size];
    for(size_t i=0; i<size; i++)
        ptr[i] = i;
    return ptr;
}
```

```
std::unique_ptr<int[]> foo(size_t size)
{
    auto ptr = std::make_unique<int[]>(size);
    for(size_t i=0; i<size; i++)
        ptr[i] = i;
    return ptr;
}
```

P1020 Smart pointer creation with default initialization

MOTIVATION

```
int* foo(size_t size)
{
    int* ptr = new int[size];
    for(size_t i=0; i<size; i++)
        ptr[i] = i;
    return ptr;
}
```

```
std::unique_ptr<int[]> foo(size_t size)
{
    auto ptr = std::make_unique<int[]>(size);
    for(size_t i=0; i<size; i++)
        ptr[i] = i;
    return ptr;
}
```

- *Value initialization* performed by `make_unique`, `make_shared`, and `allocate_shared` is redundant and hurts performance

P1020 Smart pointer creation with default initialization

SOLUTION

```
template<class T> unique_ptr<T> make_unique_default_init(); //T is not array  
template<class T> unique_ptr<T> make_unique_default_init(size_t n); //T is U[]
```

```
template<class T> shared_ptr<T> make_shared_default_init(); //T is not array  
template<class T> shared_ptr<T> make_shared_default_init(size_t N); //T is U[]
```

```
template<class T> shared_ptr<T> allocate_shared_default_init(const A& a); //T is not array  
template<class T> shared_ptr<T> allocate_shared_default_init(const A& a, size_t N); //T is U[]
```

- Perform *default initialization* on allocated objects

P0487 Fixing operator>>(basic_istream&, CharT*) (LWG 2499)

MOTIVATION

```
char* buffer = get_buffer();
std::cin >> buffer;
```

- `operator>>(basic_istream&, CharT*)` overload does not protect against buffer overflow

P0487 Fixing operator>>(basic_istream&, CharT*) (LWG 2499)

MOTIVATION

```
char* buffer = get_buffer();
std::cin >> buffer;
```

- **operator>>(basic_istream&, CharT*)** overload does not protect against buffer overflow

SOLUTION

- Remove **CharT*** overloads and *replace with bounded array ones*

```
template<class charT, class traits, size_t N>
basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& in, charT(&)[N]);
template<class traits, size_t N>
basic_istream<char, traits>& operator>>(basic_istream<char, traits>& in, unsigned char(&)[N]);
template<class traits, size_t N>
basic_istream<char, traits>& operator>>(basic_istream<char, traits>& in, signed char(&)[N]);
```

P0356 Simplified partial function application

- *Replacement* for existing `std::bind`
- `std::bind_front` applies first arguments of the function
- `bind_front(f, bound_args...)(call_args...)` is equivalent to `std::invoke(f, bound_args..., call_args....)`
- *Comparing* to `std::bind()`
 - support passing *variable number of arguments*
 - does *not allow arbitrary reordering, duplication, or removal* of the arguments

P0356 Simplified partial function application

PASSING ARGUMENTS

```
struct Strategy { double process(std::string, std::string, double, double); };
std::unique_ptr<Strategy> createStrategy();
```

P0356 Simplified partial function application

PASSING ARGUMENTS

```
struct Strategy { double process(std::string, std::string, double, double); };
std::unique_ptr<Strategy> createStrategy();
```

- Lambda expression usage is quite *verbose*

```
auto f1 = [s = createStrategy()](auto&&... args) { return s->process(std::forward<decltype(args)>(args)...); };
```

P0356 Simplified partial function application

PASSING ARGUMENTS

```
struct Strategy { double process(std::string, std::string, double, double); };
std::unique_ptr<Strategy> createStrategy();
```

- Lambda expression usage is quite *verbose*

```
auto f1 = [s = createStrategy()](auto&&... args) { return s->process(std::forward<decltype(args)>(args)...); };
```

- **std::bind** does *not allow variadic list* of arguments

```
auto f2 = std::bind(&Strategy::process, createStrategy(), _1, _2, _3, _4);
```

P0356 Simplified partial function application

PASSING ARGUMENTS

```
struct Strategy { double process(std::string, std::string, double, double); };
std::unique_ptr<Strategy> createStrategy();
```

- Lambda expression usage is quite *verbose*

```
auto f1 = [s = createStrategy()](auto&&... args) { return s->process(std::forward<decltype(args)>(args)...); };
```

- **std::bind** does *not allow variadic list* of arguments

```
auto f2 = std::bind(&Strategy::process, createStrategy(), _1, _2, _3, _4);
```

- With **bind_front** *all arguments provided on the call side are forwarded to the callable*

```
auto f3 = std::bind_front(&Strategy::process, createStrategy());
```

P0356 Simplified partial function application

PROPAGATING MUTABILITY

```
auto f1 = [s = Strategy{}](auto&&... args) { return s.process(std::forward<decltype(args)>(args)...); };
auto f2 = std::bind(&Strategy::process, Strategy{}, _1, _2, _3, _4);
auto f3 = std::bind_front(&Strategy::process, Strategy{});
```

P0356 Simplified partial function application

PROPAGATING MUTABILITY

```
auto f1 = [s = Strategy{}](auto&&... args) { return s.process(std::forward<decltype(args)>(args)...); };
auto f2 = std::bind(&Strategy::process, Strategy{}, _1, _2, _3, _4);
auto f3 = std::bind_front(&Strategy::process, Strategy{});
```

```
auto f = ...;
```

- Shall invoke **process()** member function *on a mutable object*

P0356 Simplified partial function application

PROPAGATING MUTABILITY

```
auto f1 = [s = Strategy{}](auto&&... args) { return s.process(std::forward<decltype(args)>(args)...); };
auto f2 = std::bind(&Strategy::process, Strategy{}, _1, _2, _3, _4);
auto f3 = std::bind_front(&Strategy::process, Strategy{});
```

```
auto f = ...;
```

- Shall invoke **process()** member function *on a mutable* object

```
const auto f = ...;
```

- Shall invoke process method *on an immutable/const* object
 - *true* for **std::bind** and **std::bind_front**
 - *not possible* to achieve with a lambda expression (either mutable or not)

P0356 Simplified partial function application

PRESERVING RETURN TYPE

```
struct Mapper {  
    auto operator()(int i, int j) -> std::string& { return mapping_[{i, j}]; }  
    auto operator()(int i, int j) const -> std::string& const { return mapping_[{i, j}]; }  
private:  
    std::map<std::pair<int, int>, std::string> mapping_;  
};
```

P0356 Simplified partial function application

PRESERVING RETURN TYPE

```
struct Mapper {  
    auto operator()(int i, int j) -> std::string& { return mapping_[{i, j}]; }  
    auto operator()(int i, int j) const -> std::string& const { return mapping_[{i, j}]; }  
private:  
    std::map<std::pair<int, int>, std::string> mapping_;  
};
```

```
auto      f1 = std::bind(Mapper{}, 10, _1);  
auto const f2 = std::bind(Mapper{}, 10, _1);  
auto      f3 = std::bind_front(Mapper{}, 10);  
auto const f4 = std::bind_front(Mapper{}, 10);  
auto      f5 = [m = Mapper{}](int i) mutable -> std::string& { return m(10, i); };  
auto const f6 = [m = Mapper{}](int i) -> std::string const& { return m(10, i); };
```

- *Explicit return type* needed for lambda expressions because they would otherwise return **std::string** by value

P0356 Simplified partial function application

PRESERVING RETURN TYPE

```
struct Mapper {  
    auto operator()(int i, int j) -> std::string& { return mapping_[{i, j}]; }  
    auto operator()(int i, int j) const -> std::string& const { return mapping_[{i, j}]; }  
private:  
    std::map<std::pair<int, int>, std::string> mapping_;  
};
```

```
auto      f1 = std::bind(Mapper{}, 10, _1);  
auto const f2 = std::bind(Mapper{}, 10, _1);  
auto      f3 = std::bind_front(Mapper{}, 10);  
auto const f4 = std::bind_front(Mapper{}, 10);  
auto      f5 = [m = Mapper{}](int i) mutable -> std::string& { return m(10, i); };  
auto const f6 = [m = Mapper{}](int i) -> std::string const& { return m(10, i); };
```

- *Explicit return type* needed for lambda expressions because they would otherwise return **std::string** by value

```
auto      f5 = [m = Mapper{}](int i) mutable -> decltype(auto) { return m(i, 10); };  
auto const f6 = [m = Mapper{}](int i) -> decltype(auto) { return m(i, 10); };
```

P0356 Simplified partial function application

PRESERVING VALUE CATEGORY

```
struct CachedFunc {  
    const std::string& operator()(int i, int j) &;  
private:  
    using key_type = std::pair<int, int>;  
    std::map<key_type, std::string> _cache;  
};
```

- Use of **CachedFunc** *makes sense only* in situation when it is *invoked multiple times*

P0356 Simplified partial function application

PRESERVING VALUE CATEGORY

```
struct CachedFunc {  
    const std::string& operator()(int i, int j) &;  
private:  
    using key_type = std::pair<int, int>;  
    std::map<key_type, std::string> _cache;  
};
```

- Use of **CachedFunc** *makes sense only* in situation when it is *invoked multiple times*

```
auto f1 = [cache = CachedFunc{}](int j) mutable -> std::string& { return cache(10, j); };  
auto f2 = std::bind(CachedFunc{}, 10, _1);  
auto f3 = std::bind_front(CachedFunc{}, 10);
```

- Only *f3 fails to compile*

P0356 Simplified partial function application

SUPPORTING ONE-SHOT INVOCATION

```
struct CallableOnce {  
    void operator()(int) &&;  
};
```

P0356 Simplified partial function application

SUPPORTING ONE-SHOT INVOCATION

```
struct CallableOnce {  
    void operator()(int) &&;  
};
```

```
auto make_bind(int i)      { return std::bind(CallableOnce{}, i); }  
auto make_lambda(int i)    { return [f = CallableOnce{}, i] { return f(i); }; }  
auto make_bind_front(int i) { return std::bind_front(CallableOnce{}, i); }
```

- Only `make_bind_front()` will compile

P0356 Simplified partial function application

SUPPORTING ONE-SHOT INVOCATION

```
struct CallableOnce {  
    void operator()(int) &&;  
};
```

```
auto make_bind(int i)      { return std::bind(CallableOnce{}, i); }  
auto make_lambda(int i)    { return [f = CallableOnce{}, i] { return f(i); }; }  
auto make_bind_front(int i) { return std::bind_front(CallableOnce{}, i); }
```

- Only `make_bind_front()` will compile

```
[f = CallableOnce{}, i] { return std::move(f)(i); }
```

- Workaround* for lambda
 - forcing calls on rvalue of `CallableOnce`, even if lvalue functor is invoked
 - multiple calls may be performed on single instance of `CallableOnce` class

P0591 Utility functions to implement uses-allocator construction

```
template<class T, class Alloc>
struct uses_allocator;
```

- Automatically *detects whether T has a nested allocator_type* that is convertible from **Alloc**

P0591 Utility functions to implement uses-allocator construction

```
template<class T, class Alloc>
struct uses_allocator;
```

- Automatically *detects whether T has a nested allocator_type* that is convertible from **Alloc**
- A program may *specialize this template* to derive from **true_type** for a program-defined type **T** that *does not have a nested allocator_type* but nonetheless *can be constructed with an allocator* where either
 - the first argument of a constructor has type **allocator_arg_t** and the second argument has type **Alloc**
 - the last argument of a constructor has type **Alloc**

P0591 Utility functions to implement uses-allocator construction

```
template<class T, class Alloc>
struct uses_allocator;
```

- Automatically *detects whether T has a nested allocator_type* that is convertible from **Alloc**
- A program may *specialize this template* to derive from **true_type** for a program-defined type **T** that *does not have a nested allocator_type* but nonetheless *can be constructed with an allocator* where either
 - the first argument of a constructor has type **allocator_arg_t** and the second argument has type **Alloc**
 - the last argument of a constructor has type **Alloc**
- *Used by polymorphic_allocator and scoped_allocator_adaptor already*

P0591 Utility functions to implement uses-allocator construction

MOTIVATION

- What about the case of constructing a `std::pair` where one or both members can use `Alloc`?

P0591 Utility functions to implement uses-allocator construction

MOTIVATION

- What about the case of constructing a `std::pair` where one or both members can use `Alloc`?
- What about some `Wrapper` type?

```
template<typename T, class Alloc = std::allocator<T>>
class Wrapper {
    Alloc m_alloc;
    T m_data;
    // ...
public:
    using allocator_type = Alloc;
    Wrapper(const T& v, const allocator_type& alloc = {});
    // ...
};
```

P0591 Utility functions to implement uses-allocator construction

MOTIVATION

- What about the case of constructing a `std::pair` where one or both members can use `Alloc`?
- What about some `Wrapper` type?

```
template<typename T, class Alloc = std::allocator<T>>
class Wrapper {
    Alloc m_alloc;
    T m_data;
    // ...
public:
    using allocator_type = Alloc;
    Wrapper(const T& v, const allocator_type& alloc = {});
    // ...
};
```

- Intended to work with many types for `T`, including types that
 - don't use an allocator at all
 - types that take an allocator on construction using the `allocator_arg` protocol
 - types that take an allocator on construction as a trailing argument

P0591 Utility functions to implement uses-allocator construction

SOLUTION

- *Introduction of make_obj_using_allocator* function template to construct the **m_data** member using the supplied allocator

```
template<typename T, class Alloc>
Wrapper<T,Alloc>::Wrapper(const T& v, const allocator_type& alloc)
    : m_alloc(alloc)
    , m_data(make_obj_using_allocator<T>(alloc, v))
    , // ...
{ /* ... */ }
```

P0591 Utility functions to implement uses-allocator construction

```
template<class T, class Alloc, class... Args>
auto uses_allocator_construction_args(const Alloc& alloc, Args&&... args) -> see below;

template<class T, class Alloc, class Tuple1, class Tuple2>
auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_t, Tuple1&& x, Tuple2&& y) -> see below;

template<class T>
auto uses_allocator_construction_args(const Alloc& alloc) -> see below;

template<class T, class Alloc, class U, class V>
auto uses_allocator_construction_args(const Alloc& alloc, U&& u, V&& v) -> see below;

template<class T, class Alloc, class U, class V>
auto uses_allocator_construction_args(const Alloc& alloc, const pair<U,V>& pr) -> see below;

template<class T, class Alloc, class U, class V>
auto uses_allocator_construction_args(const Alloc& alloc, pair<U,V>&& pr) -> see below;
```

- Produces (as a **tuple**) a new argument list matching one of the above conventions
- Overloads are provided that treat specializations of **std::pair** such that uses-allocator construction is applied individually to the first and second data members

P0591 Utility functions to implement uses-allocator construction

```
template<class T, class Alloc, class... Args>
T make_obj_using_allocator(const Alloc& alloc, Args&&... args)
{
    return make_from_tuple<T>(uses_allocator_construction_args<T>(alloc, std::forward<Args>(args)...));
}
```

```
template<class T, class Alloc, class... Args>
T* uninitialized_construct_using_allocator(T* p, const Alloc& alloc, Args&&... args)
{
    return ::new(static_cast<void*>(p)) T(make_obj_using_allocator<T>(alloc, std::forward<Args>(args)...));
}
```

- Apply the modified constructor arguments to construct an object of type **T** as a return value or in-place, respectively

P0591 Utility functions to implement uses-allocator construction

```
template<class T, class Alloc, class... Args>
T make_obj_using_allocator(const Alloc& alloc, Args&&... args)
{
    return make_from_tuple<T>(uses_allocator_construction_args<T>(alloc, std::forward<Args>(args)...));
}
```

```
template<class T, class Alloc, class... Args>
T* uninitialized_construct_using_allocator(T* p, const Alloc& alloc, Args&&... args)
{
    return ::new(static_cast<void*>(p)) T(make_obj_using_allocator<T>(alloc, std::forward<Args>(args)...));
}
```

- Apply the modified constructor arguments to construct an object of type **T** as a return value or in-place, respectively
- Remove 5 out of 6 **construct** member functions (leaving only the first generic case) of
 - **polymorphic_allocator**
 - **scoped_allocator_adaptor**
- Make the first ones use new **uses_allocator_construction_args** facilities

P0339 `polymorphic_allocator`<T> as a vocabulary type

MOTIVATION

- The `pmr::memory_resource` type provides a way to control the memory allocation for an object without affecting its compile-time type by accept a pointer to `pmr::memory_resource` in a class constructor
- The `pmr::polymorphic_allocator<T>` adaptor class allows memory resources to be used in all places where allocators are used in the standard
- However, for many classes the `T` parameter does not make sense

P0339 polymorphic_allocator<> as a vocabulary type

MOTIVATION

```
class IntVec {  
    std::size_t size_ = 0;  
    std::size_t capacity_;  
    int* data_;  
public:  
    IntVec(std::size_t capacity):  
        capacity_(capacity),  
        data_(new int[capacity]) {}  
    // ...  
};
```

- No control over allocation

P0339 polymorphic_allocator<> as a vocabulary type

MOTIVATION

```
template<class Alloc = std::allocator<int>>
class IntVec {
public:
    using allocator_type = Alloc;
private:
    using a_traits = std::allocator_traits<Alloc>;
    std::size_t size_ = 0;
    std::size_t capacity_;
    Alloc alloc_;
    int* data_;
public:
    IntVec(std::size_t capacity, Alloc alloc = {}):
        capacity_(capacity),
        alloc_(alloc),
        data_(a_traits::allocate(alloc_, capacity_)) {}
    // ...
};
```

- Forces class template
- Requires **std::allocator_traits** usage

P0339 `polymorphic_allocator`<> as a vocabulary type

MOTIVATION

```
class IntVec {
    std::size_t size_ = 0;
    std::size_t capacity_;
    std::pmr::memory_resource* memrsrc_;
    int* data_;
public:
    IntVec(std::size_t capacity,
            std::pmr::memory_resource* memrsrc =
                std::pmr::get_default_resource()):
        capacity_(capacity),
        memrsrc_(&memrsrc),
        data_(memrsrc_->allocate(capacity_ * sizeof(int),
                                    alignof(int))) {}
    // ...
};
```

- Does not conform to the **Allocator** concept
 - doesn't fit into the facilities designed for allocators (i.e. uses-allocator construction)
- Lack of reasonable value-initialization
 - the programmer must explicitly call `std::pmr::get_default_resource()`
- Danger of null pointers
- Inadvertent reseating of the memory resource
 - move- and copy-assignment don't have to move or copy the allocator or memory resource

P0339 `polymorphic_allocator` as a vocabulary type

MOTIVATION

```
class IntVec {  
public:  
    using allocator_type =  
        std::pmr::polymorphic_allocator<int>;  
private:  
    std::size_t size_ = 0;  
    std::size_t capacity_;  
    allocator_type alloc_;  
    int* data_;  
public:  
    IntVec(std::size_t capacity, allocator_type alloc = {}):  
        capacity_(capacity),  
        alloc_(alloc),  
        data_(alloc.allocate(capacity_)) {}  
    // ...  
};
```

- Plays nicely in the world of uses-allocator construction
- Value-initializing the allocator causes the default memory resource to be used
- A `polymorphic_allocator` is not a pointer and cannot be null
- The assignment operators for `polymorphic_allocator` are deleted
 - the problem of accidentally reseating the allocator does not exist

P0339 `polymorphic_allocator`<> as a vocabulary type

MOTIVATION

```
class IntVec {  
public:  
    using allocator_type =  
        std::pmr::polymorphic_allocator<int>;  
private:  
    std::size_t size_ = 0;  
    std::size_t capacity_;  
    allocator_type alloc_;  
    int* data_;  
public:  
    IntVec(std::size_t capacity, allocator_type alloc = {}):  
        capacity_(capacity),  
        alloc_(alloc),  
        data_(alloc.allocate(capacity_)) {}  
    // ...  
};
```

- `polymorphic_allocator` is a template, when what is desired is a non-template vocabulary type
- In order to allocate objects of different types, it is necessary to rebind the allocator
 - a step backwards from direct use of `memory_resource` which does not require rebinding

P0339 polymorphic_allocator<> as a vocabulary type

SOLUTION

```
template<class Tp = byte>
class polymorphic_allocator {
public:
    void* allocate_bytes(size_t nbytes,
                         size_t alignment = alignof(max_align_t));
    void deallocate_bytes(void* p, size_t nbytes,
                         size_t alignment = alignof(max_align_t));

    template<class T>
    T* allocate_object(size_t n = 1);

    template<class T>
    void deallocate_object(T* p, size_t n = 1);

    template<class T, class... CtorArgs>
    T* new_object(CtorArgs&&... ctor_args);

    template<class T>
    void delete_object(T* p);
};

//...
```

- **polymorphic_allocator<>** is a class and not a class template
- It can allocate objects of any type without needing to use rebind
- It can allocate objects on any desired alignment boundary
- It provides member functions to allocate and construct objects in one step
- It provides a good alternative to type erasure for types that don't have an allocator template argument (i.e. **std::function**)

P0811 Well-behaved interpolation for numbers and pointers

MOTIVATION

- The simple problem of computing a value between two other values is surprisingly subtle in general

P0811 Well-behaved interpolation for numbers and pointers

MOTIVATION

- The simple problem of computing a value between two other values is surprisingly subtle in general

```
(a + b) / 2
```

- Can cause overflow for signed or unsigned integers as well as for floating-point values

P0811 Well-behaved interpolation for numbers and pointers

MOTIVATION

- The simple problem of computing a value between two other values is surprisingly subtle in general

```
(a + b) / 2
```

- Can cause overflow for signed or unsigned integers as well as for floating-point values

```
a + (b - a) / 2
```

- Works for signed integers with the same sign (even if $b < a$), but can overflow if they have different signs

P0811 Well-behaved interpolation for numbers and pointers

MOTIVATION

- The simple problem of computing a value between two other values is surprisingly subtle in general

```
(a + b) / 2
```

- Can cause overflow for signed or unsigned integers as well as for floating-point values

```
a + (b - a) / 2
```

- Works for signed integers with the same sign (even if $b < a$), but can overflow if they have different signs

```
a / 2 + b / 2
```

- For floating-point types prevents overflow but is not correctly rounded for subnormal inputs

P0811 Well-behaved interpolation for numbers and pointers

SOLUTION

- New library functions to compute
 - the midpoint between two integer, floating-point, or pointer values

```
template<class T>
constexpr T midpoint(T a, T b) noexcept;
template<class T>
constexpr T* midpoint(T* a, T* b);
```

P0811 Well-behaved interpolation for numbers and pointers

SOLUTION

- New library functions to compute
 - the midpoint between two integer, floating-point, or pointer values

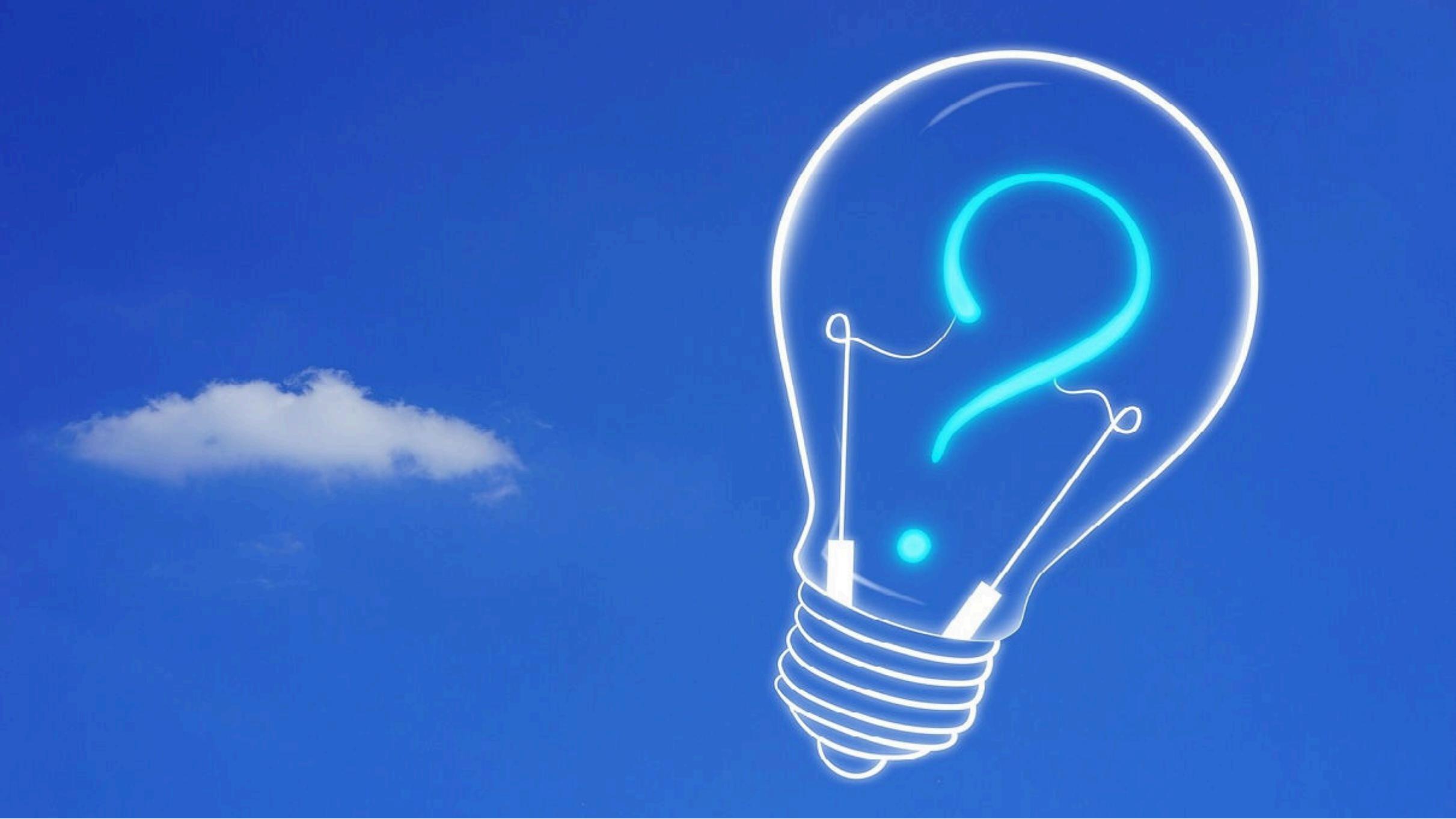
```
template<class T>
constexpr T midpoint(T a, T b) noexcept;
template<class T>
constexpr T* midpoint(T* a, T* b);
```

- a more general routine that interpolates (or extrapolates) between two floating-point values

```
constexpr float lerp(float a, float b, float t);
constexpr double lerp(double a, double b, double t);
constexpr long double lerp(long double a, long double b, long double t);
```

Next meetings

DATE	PLACE	SUBJECT
15-20 Jul 2019	Cologne, Germany	CWG+LWG: Complete CD wording EWG+LEWG: Working on C++23 features + CWG/LWG design clarification questions C++20 draft wording is feature complete, start CD ballot
04-09 Nov 2019	Belfast, Northern Ireland	CD ballot comment resolution
10-15 Feb 2020	Prague, Czech Republic	CD ballot comment resolution, C++20 technically finalized, start DIS ballot
01-06 Jun 2020	Bulgaria	First meeting of C++23
Nov 2020	New York, NY, USA	
22-27 Feb 2021	Kona, HI, USA	



CAUTION
Programming
is addictive
(and too much fun)