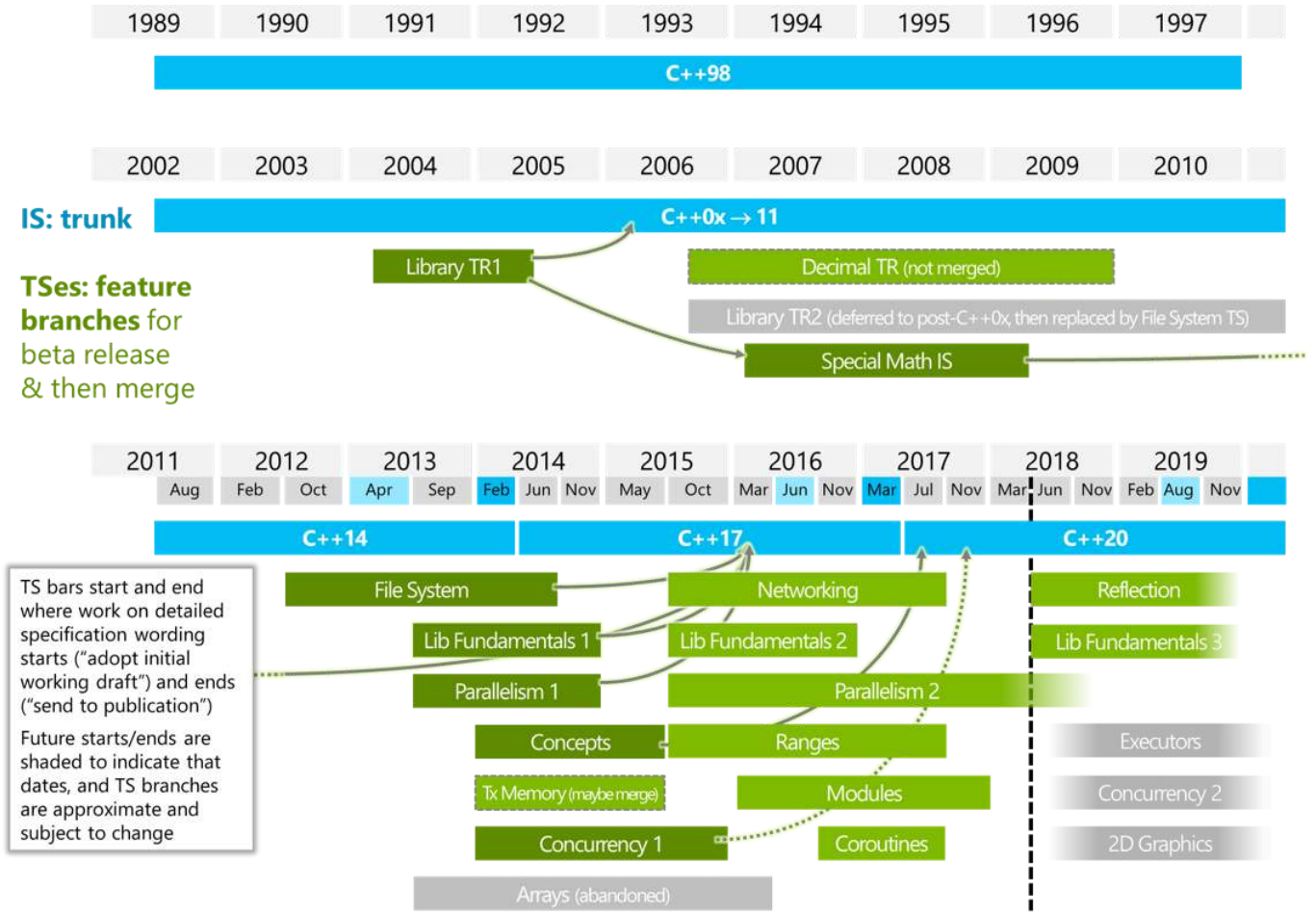
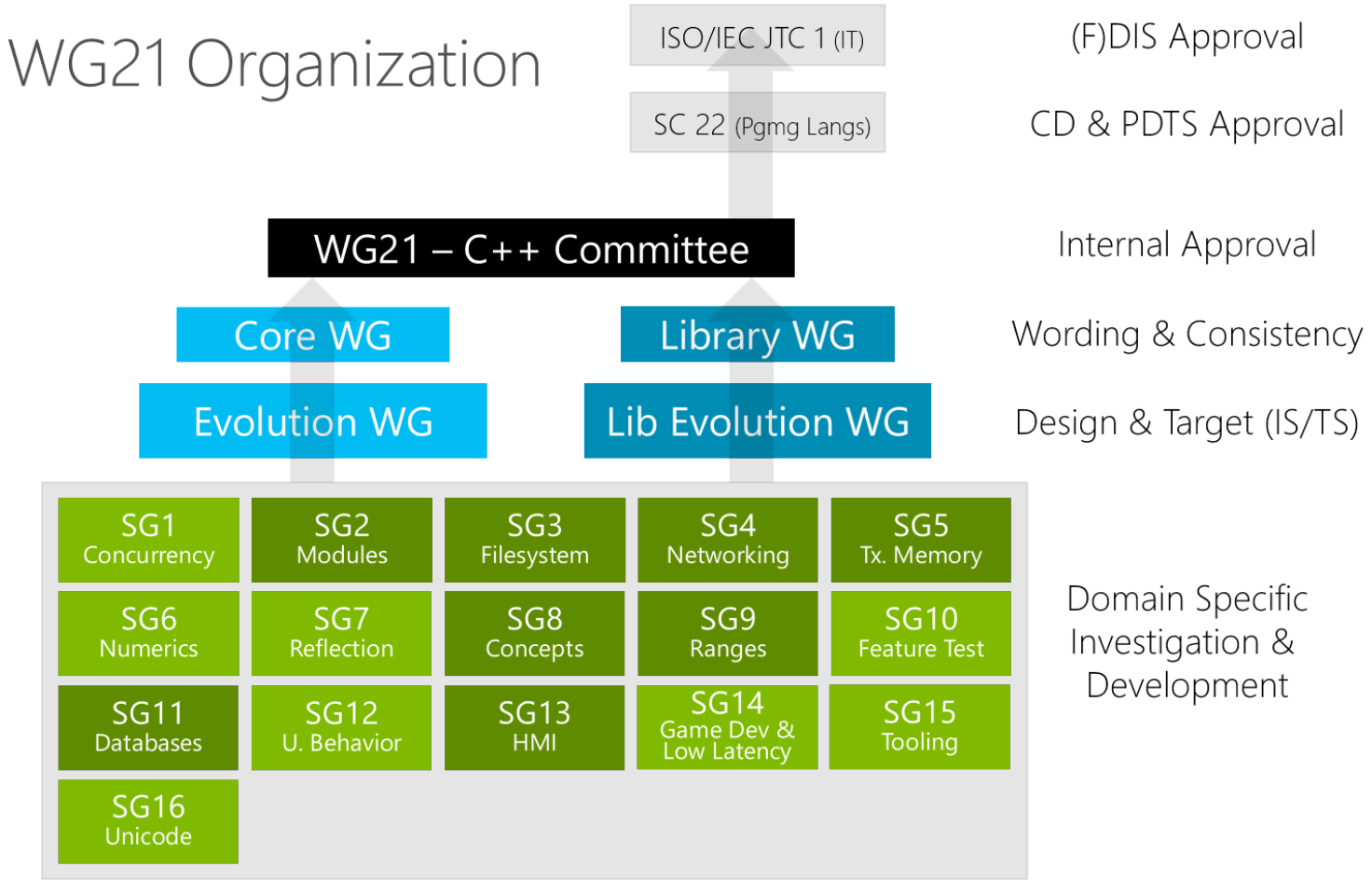

epam
BEYOND C++17
Mateusz Pusz
May 7, 2018

C++ TIMELINE


1989
1990
1991
1992
1993
1994
1995
1996
1997
C++98
2002
2003
2004
2005
2006
2007
2008
2009
2010
IS: trunk
C++0x → 11
Library TR1
Decimal TR (not merged)
Library TR2 (deferred to post-C++0x, then replaced by File System TS)
Special Math IS
TSes: feature branches for beta release & then merge
2011
2012
2013
2014
2015
2016
2017
2018
2019
Aug
Feb
Oct
Apr
Sep
Feb
Jun
Nov
May
Oct
Mar
Jun
Nov
Mar
Jul
Nov
Mar
Jun
Nov
Feb
Aug
Nov
C++14
C++17
C++20
TS bars start and end where work on detailed specification wording starts ("adopt initial working draft") and ends ("send to publication")
Future starts/ends are shaded to indicate that dates, and TS branches are approximate and subject to change
File System
Networking
Reflection
Lib Fundamentals 1
Lib Fundamentals 2
Lib Fundamentals 3
Parallelism 1
Parallelism 2
Concepts
Ranges
Executors
Tx Memory (maybe merge)
Modules
Concurrency 2
Concurrency 1
Coroutines
2D Graphics
Arrays (abandoned)

ISO C++ COMMITTEE STRUCTURE

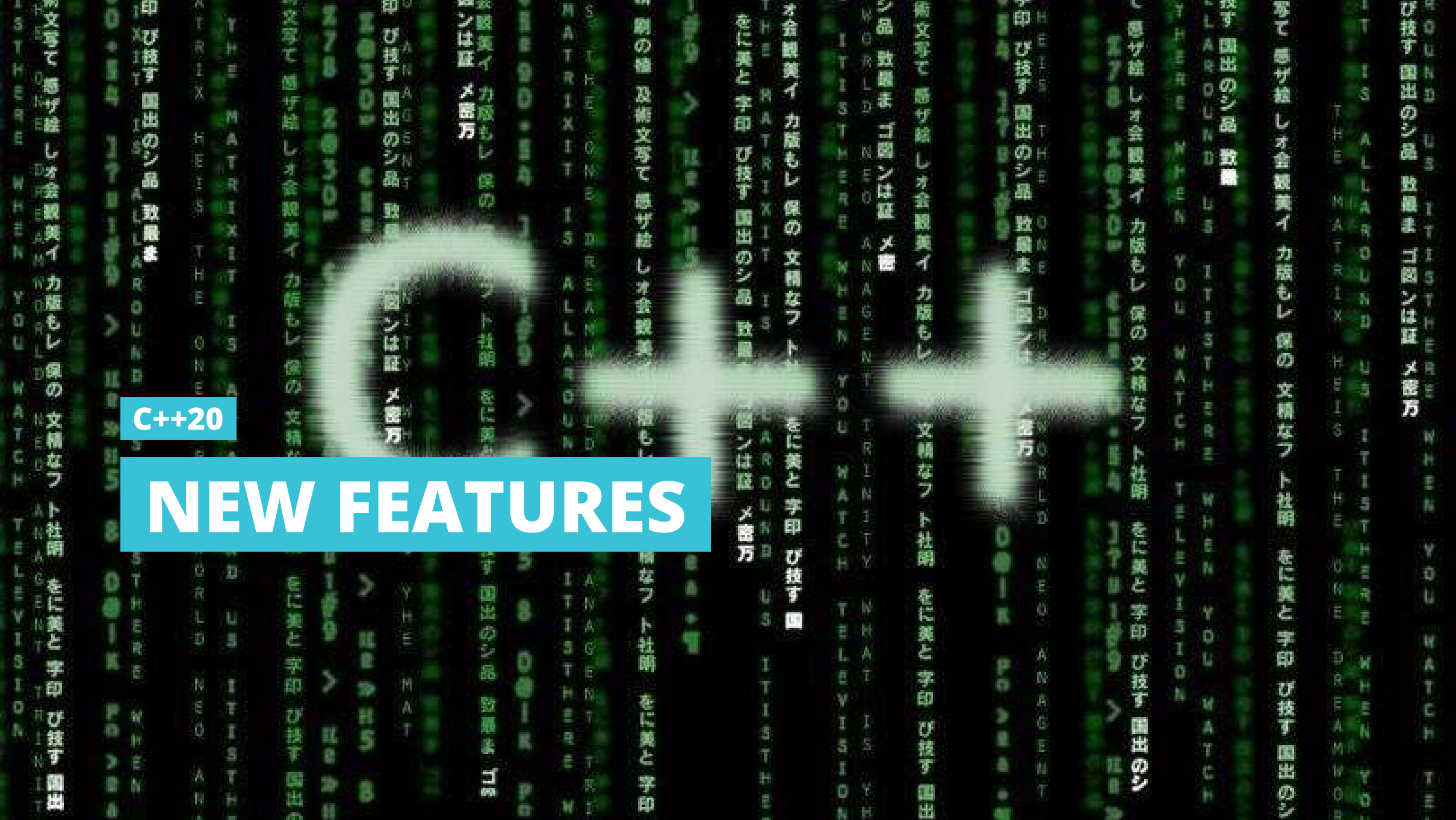


MAJOR FEATURES STATUS

	DEPENDS ON	CURRENT TARGET (ESTIMATED)
Coroutines		C++20
Contracts		C++20
Ranges		Core concepts in C++20 Rest in C++20 or 23
Modules		Core concepts in C++20 Rest in (TBD) focusing on a bridge from header files
Reflection		TS in C++20 timeframe; IS in C++23
Executors		TS in C++20 timeframe; IS in C++23
Networking TS	Executors	IS in C++23
future.then, async2	Executors	IS in C++23

FINDING A PAPER - HTTPS://WG21.LINK

- Usage info
  - wg21.link
- Get paper
  - wg21.link/nXXXX
  - wg21.link/pXXXX - latest version (e.g. wg21.link/p0463)
  - wg21.link/pXXXXrX
- Get working draft
  - wg21.link/standard
  - wg21.link/concepts
  - wg21.link/coroutines
  - wg21.link/modules
  - wg21.link/networking
  - wg21.link/ranges


C++20
NEW FEATURES

CONCEPTS

- Concepts TS standardised 2 years ago
- No consensus on merging to IS as is
- Consensus reached by postponing the merge of
  - introducer syntax
  - terse/natural syntax
- Small changes approved
  - removed **`bool`** from concept syntax
  - removed function concepts
- P0734 **C++ extensions for Concepts** merged with IS

C++ CONCEPTS IN ACTION

ACCEPTED FEATURES

Concept definition

```
template<class T>
concept Sortable { /* ... */ }
```

Original template notation

```
template<typename T>
  requires Sortable<T>
void sort(T&);
```

The shorthand notation

```
template<Sortable T>
void sort(T&);
```

C++ CONCEPTS IN ACTION

ACCEPTED FEATURES

Concept definition

```
template<class T>
concept Sortable { /* ... */ }
```

Original template notation

```
template<typename T>
  requires Sortable<T>
void sort(T&);
```

The shorthand notation

```
template<Sortable T>
void sort(T&);
```

NOT ACCEPTED FEATURES

The terse/natural notation

```
void sort(Sortable&);
// Not merged to IS
```

The concept introducer notation

```
Sortable{Seq} void sort(Seq&);
// Not merged to IS
```

CONSTRAINTS AND CONCEPTS

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

CONSTRAINTS AND CONCEPTS

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

```
void foo()
{
  f("abc"s);    // OK
  std::mutex mtx;
  f(mtx);       // Error
}
```

CONSTRAINTS AND CONCEPTS

```
<source>: In instantiation of 'void f(T&&) [with T = std::mutex&]':
28 : <source>:28:8:   required from here
15 : <source>:15:8: error: no match for 'operator==' (operand types are 'std::mutex' and 'std::mutex')
   if(t == other) {
      ~~^~~~~~~~
In file included from /opt/compiler-explorer/gcc-7.2.0/include/c++/7.2.0/mutex:42:0,
                 from <source>:2:
/opt/compiler-explorer/gcc-7.2.0/include/c++/7.2.0/system_error:311:3: note: candidate: bool std::operator==(const std::error_cond
   operator==(const error_condition& __lhs,
   ^~~~~~~~
/opt/compiler-explorer/gcc-7.2.0/include/c++/7.2.0/system_error:311:3: note:   no known conversion for argument 1 from 'std::mutex
/opt/compiler-explorer/gcc-7.2.0/include/c++/7.2.0/system_error:304:3: note: candidate: bool std::operator==(const std::error_cond
   operator==(const error_condition& __lhs, const error_code& __rhs) noexcept
   ^~~~~~~~

... 290 lines more ...

In file included from /opt/compiler-explorer/gcc-7.2.0/include/c++/7.2.0/bits/stl_algobase.h:64:0,
                 from /opt/compiler-explorer/gcc-7.2.0/include/c++/7.2.0/memory:62,
                 from <source>:1:
/opt/compiler-explorer/gcc-7.2.0/include/c++/7.2.0/bits/stl_pair.h:443:5: note: candidate: template<class _T1, class _T2> constexp
     operator==(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
     ^~~~~~~~
/opt/compiler-explorer/gcc-7.2.0/include/c++/7.2.0/bits/stl_pair.h:443:5: note:   template argument deduction/substitution failed
15 : <source>:15:8: note:   'std::mutex' is not derived from 'const std::pair<_T1, _T2>'
   if(t == other) {
      ~~^~~~~~~~
Compiler exited with result code 1
```

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
}
```

CONSTRAINTS AND CONCEPTS

```
<source>: In function 'void foo()':
28 : <source>:28:8: error: cannot call function 'void f(T&&) [with T = std::mutex&]'
   f(mtx);       // Error: not EqualityComparable
        ^
12 : <source>:12:6: note:   constraints not satisfied
 void f(T&& t)
      ^
6 : <source>:6:14: note: within 'template<class T> concept const bool EqualityComparable<T> [with T = std::mutex&]'
 concept bool EqualityComparable = requires(T a, T b) {
              ^~~~~~~~~~~~~~~~~
6 : <source>:6:14: note:     with 'std::mutex& a'
6 : <source>:6:14: note:     with 'std::mutex& b'
6 : <source>:6:14: note: the required expression '(a == b)' would be ill-formed
Compiler exited with result code 1
```

P0898 STANDARD LIBRARY CONCEPTS (WIP)

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 (WIP)

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 (WIP)

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;
```

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 (WIP)

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;
```

WHAT DAY OF THE WEEK IS JULY 4, 2001?

C

```c
#include <stdio.h>
#include <time.h>

static const char* const wday[] =
{
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday", "-unknown-"
};

int main()
{
  struct tm time_str;
  time_str.tm_year   = 2001 - 1900;
  time_str.tm_mon    = 7 - 1;
  time_str.tm_mday   = 4;
  time_str.tm_hour   = 0;
  time_str.tm_min    = 0;
  time_str.tm_sec    = 0;
  time_str.tm_isdst  = -1;
  if (mktime(&time_str) == (time_t)(-1))
    time_str.tm_wday = 7;
  printf("%s\n", wday[time_str.tm_wday]);
}
```

WHAT DAY OF THE WEEK IS JULY 4, 2001?

C

```c
#include <stdio.h>
#include <time.h>

static const char* const wday[] =
{
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday", "-unknown-"
};

int main()
{
  struct tm time_str;
  time_str.tm_year   = 2001 - 1900;
  time_str.tm_mon    = 7 - 1;
  time_str.tm_mday   = 4;
  time_str.tm_hour   = 0;
  time_str.tm_min    = 0;
  time_str.tm_sec    = 0;
  time_str.tm_isdst  = -1;
  if (mktime(&time_str) == (time_t)(-1))
    time_str.tm_wday = 7;
  printf("%s\n", wday[time_str.tm_wday]);
}
```

C++20

```c++
#include <chrono>
#include <iostream>

int main()
{
  using namespace std::chrono;
  std::cout << weekday{jul/4/2001} << '\n';
}
```

P0355 EXTENDING CHRONO TO CALENDARS AND TIME ZONES

GOALS

- Seamless integration with the existing library
- Type safety
- Detection of errors at compile time
- Performance
- Ease of use
- Readable code
- No artificial restrictions on precision

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};
std::cout << zt << '\n';                                     // 2016-05-29 16:30:06.153 JST
```

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

DOCUMENTATION

- Calendar: http://howardhinnant.github.io/date/date.html
- TimeZone: http://howardhinnant.github.io/date/tz.html

VIDEO INTRODUCTION

- Calendar: https://www.youtube.com/watch?v=tzyGjOm8AKo
- Time Zone: https://www.youtube.com/watch?v=Vwd3pduVGKY

FULL IMPLEMENTATION

- https://github.com/HowardHinnant/date

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.

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.

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*

P0122

```
constexpr ptrdiff_t dynamic_extent = -1;

template <class ElementType, ptrdiff_t Extent = dynamic_extent>
class span;
```

P0122

```
constexpr ptrdiff_t dynamic_extent = -1;

template <class ElementType, ptrdiff_t Extent = dynamic_extent>
class span;
```

DYNAMIC-SIZE (PROVIDED AT RUNTIME)

- **dynamic_extent** is a unique value outside the normal range of lengths reserved to indicate that the length of the sequence is only known at runtime and must be stored within the span
- A dynamic-size **span** is, conceptually, just a pointer and size field

```
int* somePointer = new int[someLength];

span<int> s{somePointer, someLength};
```

P0122

```
constexpr ptrdiff_t dynamic_extent = -1;

template <class ElementType, ptrdiff_t Extent = dynamic_extent>
class span;
```

STATIC-SIZE (FIXED AT COMPILE-TIME)

- Provides a value for **`Extent`** that is between **`0`** and **`PTRDIFF_MAX`** (inclusive)
- Requires no storage size overhead beyond a single pointer

```
int arr[10];

span<int, 10> s1{arr};    // fixed-size span of 10 ints
// span<int, 20> s2{arr}; // ERROR: will fail to compile
span<int> s3{arr};        // dynamic-size span of 10 ints
```

P0122

```
constexpr ptrdiff_t dynamic_extent = -1;

template <class ElementType, ptrdiff_t Extent = dynamic_extent>
class span;
```

FIXED AND STATIC SIZE CONVERSIONS

- A **fixed-size** span may be constructed or assigned from *another fixed-size span of equal length*
- A **dynamic-size** span may always be constructed or assigned from a *fixed-size span*
- A **fixed-size** span may always be constructed or assigned from a *dynamic-size span*
  - undefined behavior will result if the construction or assignment is not bounds-safe

P0122

CONSTRUCTION

```
constexpr span();
constexpr span(pointer ptr, index_type count);
constexpr span(pointer firstElem, pointer lastElem);
template <size_t N>
constexpr span(element_type (&arr)[N]);
template <size_t N>
constexpr span(array<remove_const_t<element_type>, N>& arr);
template <size_t N>
constexpr span(const array<remove_const_t<element_type>, N>& arr);
template <class Container>
constexpr span(Container& cont);
template <class Container>
constexpr span(const Container& cont);
constexpr span(const span& other) noexcept = default;
template <class OtherElementType, ptrdiff_t OtherExtent>
constexpr span(const span<OtherElementType, OtherExtent>& other);
```

P0122

ELEMENT ACCESS AND ITERATION

```
constexpr reference operator[](index_type idx) const;
constexpr reference operator()(index_type idx) const;
constexpr pointer data() const noexcept;
```

P0122

ELEMENT ACCESS AND ITERATION

```
constexpr reference operator[](index_type idx) const;
constexpr reference operator()(index_type idx) const;
constexpr pointer data() const noexcept;
```

```
constexpr iterator begin() const noexcept;
constexpr iterator end() const noexcept;
constexpr const_iterator cbegin() const noexcept;
constexpr const_iterator cend() const noexcept;
constexpr reverse_iterator rbegin() const noexcept;
constexpr reverse_iterator rend() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;
```

P0122

BYTE REPRESENTATIONS AND CONVERSIONS

```
template <class ElementType, ptrdiff_t Extent>
span<const byte, ((Extent == dynamic_extent) ? dynamic_extent : (sizeof(ElementType)*Extent))>
  as_bytes(span<ElementType, Extent> s) noexcept;

template <class ElementType, ptrdiff_t Extent>
span<byte, ((Extent == dynamic_extent) ? dynamic_extent : (sizeof(ElementType)*Extent))>
  as_writeable_bytes(span<ElementType, Extent>) noexcept;
```

P0122

COMPARISONS

```
template <class ElementType, ptrdiff_t Extent>
constexpr bool operator==(span<ElementType, Extent> l, span<ElementType, Extent> r);
template <class ElementType, ptrdiff_t Extent>
constexpr bool operator!=(span<ElementType, Extent> l, span<ElementType, Extent> r);
template <class ElementType, ptrdiff_t Extent>
constexpr bool operator<(span<ElementType, Extent> l, span<ElementType, Extent> r);
template <class ElementType, ptrdiff_t Extent>
constexpr bool operator<=(span<ElementType, Extent> l, span<ElementType, Extent> r);
template <class ElementType, ptrdiff_t Extent>
constexpr bool operator>(span<ElementType, Extent> l, span<ElementType, Extent> r);
template <class ElementType, ptrdiff_t Extent>
constexpr bool operator>=(span<ElementType, Extent> l, span<ElementType, Extent> r);
```

P0122

CREATING SUB-SPANS

```
constexpr span<element_type, dynamic_extent> first(index_type count) const;
constexpr span<element_type, dynamic_extent> last(index_type count) const;
constexpr span<element_type, dynamic_extent> subspan(index_type offset,
                                                     index_type count = dynamic_extent) const;
```

P0122

CREATING SUB-SPANS

```
constexpr span<element_type, dynamic_extent> first(index_type count) const;
constexpr span<element_type, dynamic_extent> last(index_type count) const;
constexpr span<element_type, dynamic_extent> subspan(index_type offset,
                                                     index_type count = dynamic_extent) const;
```

```
template <ptrdiff_t Count>
constexpr span<element_type, Count> first() const;
template <ptrdiff_t Count>
constexpr span<element_type, Count> last() const;
template <ptrdiff_t Offset, ptrdiff_t Count = dynamic_extent>
constexpr span<element_type, Count> subspan() const;
```

P0122

- *Cheap* to construct, copy, move, and use
- Users are encouraged to use it as a *pass-by-value parameter* type
- Construction or assignment between span objects *with different element types* is allowed whenever it can be determined statically that the element types are exactly storage-size equivalent
- It is always possible to convert from a **span<T>** to a **span<const T>**, it is not allowed to convert in the opposite direction, from **span<const T>** to **span<T>**
- Span has a *trivial destructor*, so common ABI conventions allow it to *be passed in registers*

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

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

<=> operator nearly ended in a header named "=" ;-)

P0515 P0768 P0905 CONSISTENT COMPARISON

```
class totally_ordered : public base {
  std::string tax_id_;
  std::string first_name_;
  std::string last_name_;
public:
  std::strong_ordering operator<=>(const totally_ordered& other) const
  {
    if(auto cmp = (base&)(*this) <=> (base&)other; cmp != 0) return cmp;
    if(auto cmp = last_name_ <=> other.last_name_; cmp != 0) return cmp;
    if(auto cmp = first_name_ <=> other.first_name_; cmp != 0) return cmp;
    return tax_id_ <=> other.tax_id_;
 }
 // ... non-comparison functions ...
};
```

P0515 P0768 P0905 CONSISTENT COMPARISON

```
class totally_ordered : public base {
  std::string tax_id_;
  std::string first_name_;
  std::string last_name_;
public:
  std::strong_ordering operator<=>(const totally_ordered& other) const
  {
    if(auto cmp = (base&)(*this) <=> (base&)other; cmp != 0) return cmp;
    if(auto cmp = last_name_ <=> other.last_name_; cmp != 0) return cmp;
    if(auto cmp = first_name_ <=> other.first_name_; cmp != 0) return cmp;
    return tax_id_ <=> other.tax_id_;
 }
 // ... non-comparison functions ...
};
```

Compile-time error if a member does not have a **`strong_ordering`**

P0515 P0768 P0905 CONSISTENT COMPARISON

TYPE	CATEGORY
`bool`, integral, pointer types	**`std::strong_ordering`**
floating point types	**`std::partial_ordering`**
enumerations	the same as underlying type
`std::nullptr_t`	**`std::strong_ordering`**
copyable arrays **`T[N]`**	the same as **`T`**
other arrays	no **`<=>`**
`void`	no **`<=>`**

All built-in **`<=>`** comparisons are **`constexpr`** beside pointers into the different object/allocation

P0515 P0768 P0905 CONSISTENT COMPARISON

```
class std::weak_equality;
class std::strong_equality;
class std::partial_ordering;
class std::weak_ordering;
class std::strong_ordering;
```

P0515 P0768 P0905 CONSISTENT COMPARISON

```
class std::weak_equality;
class std::strong_equality;
class std::partial_ordering;
class std::weak_ordering;
class std::strong_ordering;
```

```
constexpr bool std::is_eq  (std::weak_equality cmp)    noexcept { return cmp == 0; }
constexpr bool std::is_neq (std::weak_equality cmp)    noexcept { return cmp != 0; }
constexpr bool std::is_lt  (std::partial_ordering cmp) noexcept { return cmp <  0; }
constexpr bool std::is_lteq(std::partial_ordering cmp) noexcept { return cmp <= 0; }
constexpr bool std::is_gt  (std::partial_ordering cmp) noexcept { return cmp >  0; }
constexpr bool std::is_gteq(std::partial_ordering cmp) noexcept { return cmp >= 0; }
```

P0515 P0768 P0905 CONSISTENT COMPARISON

```
class std::weak_equality;
class std::strong_equality;
class std::partial_ordering;
class std::weak_ordering;
class std::strong_ordering;
```

```
constexpr bool std::is_eq  (std::weak_equality cmp)    noexcept { return cmp == 0; }
constexpr bool std::is_neq (std::weak_equality cmp)    noexcept { return cmp != 0; }
constexpr bool std::is_lt  (std::partial_ordering cmp) noexcept { return cmp <  0; }
constexpr bool std::is_lteq(std::partial_ordering cmp) noexcept { return cmp <= 0; }
constexpr bool std::is_gt  (std::partial_ordering cmp) noexcept { return cmp >  0; }
constexpr bool std::is_gteq(std::partial_ordering cmp) noexcept { return cmp >= 0; }
```

```
template<class T> constexpr std::strong_ordering  std::strong_order (const T& a, const T& b);
template<class T> constexpr std::weak_ordering    std::weak_order   (const T& a, const T& b);
template<class T> constexpr std::partial_ordering std::partial_order(const T& a, const T& b);
template<class T> constexpr std::strong_equality  std::strong_equal (const T& a, const T& b);
template<class T> constexpr std::weak_equality    std::weak_equal   (const T& a, const T& b);
```

P0515 P0768 P0905 CONSISTENT COMPARISON

```
template<class T, class U>
constexpr auto std::compare_3way(const T& a, const U& b);

template<class InputIterator1, class InputIterator2>
constexpr auto std::lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                                 InputIterator2 b2, InputIterator2 e2);

template<class InputIterator1, class InputIterator2, class Cmp>
constexpr auto std::lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                                 InputIterator2 b2, InputIterator2 e2,
                                                 Cmp comp)
    -> std::common_comparison_category_t<decltype(comp(*b1,*b2)), std::strong_ordering>;
```

P0515 P0768 P0905 CONSISTENT COMPARISON

```
template<class T, class U>
constexpr auto std::compare_3way(const T& a, const U& b);

template<class InputIterator1, class InputIterator2>
constexpr auto std::lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                                 InputIterator2 b2, InputIterator2 e2);

template<class InputIterator1, class InputIterator2, class Cmp>
constexpr auto std::lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                                 InputIterator2 b2, InputIterator2 e2,
                                                 Cmp comp)
    -> std::common_comparison_category_t<decltype(comp(*b1,*b2)), std::strong_ordering>;
```

std::rel_ops are now deprecated

P0053 SYNCHRONIZED BUFFERED OSTREAMS

- *Atomically transfers* the contents of an internal stream buffer to a **basic_ostream**'s stream buffer *on destruction of the* ***basic_osyncstream***

```
{
  std::osyncstream out{std::cout};
  out << "Hello, " << "World!" << '\n';
}
```

```
std::osyncstream{std::cout} << "The answer is " << 6*7 << std::endl;
```

P0053 SYNCHRONIZED BUFFERED OSTREAMS

```
template<class charT, class traits, class Allocator>
class basic_syncbuf : public basic_streambuf<charT, traits> {
public:
  bool emit();
  streambuf_type* get_wrapped() const noexcept;
  void set_emit_on_sync(bool) noexcept;
protected:
  int sync() override;
  // ...
};

using syncbuf = basic_syncbuf<char>;
using wsyncbuf = basic_syncbuf<wchar_t>;
```

- **`emit()`** *atomically transfers the contents* of the internal buffer to the wrapped stream buffer, so that they appear in the output stream as a contiguous sequence of characters
- **`sync()`** *records that* the wrapped stream buffer is *to be flushed*, then, if **`emit_on_sync == true`**, calls **`emit()`**

P0053 SYNCHRONIZED BUFFERED OSTREAMS

```
template<class charT, class traits, class Allocator>
class basic_osyncstream : public basic_ostream<charT, traits> {
  basic_syncbuf<charT, traits, Allocator> sb_;
public:
  void emit() { sb_.emit(); }
  streambuf_type* get_wrapped() const noexcept { return sb_.get_wrapped(); }
  syncbuf_type* rdbuf() const noexcept { return &sb_; }
  // ...
};

using osyncstream = basic_osyncstream<char>;
using wosyncstream = basic_osyncstream<wchar_t>;
```

P0053 SYNCHRONIZED BUFFERED OSTREAMS

EXAMPLE: A FLUSH ON A BASIC_OSYNCSTREAM DOES NOT FLUSH IMMEDIATELY

```
{
  std::osyncstream out{std::cout};
  out << "Hello," << '\n';       // no flush
  out.emit();                    // characters transferred; cout not flushed
  out << "World!" << std::endl; // flush noted; cout not flushed
  out.emit();                    // characters transferred; cout flushed
  out << "Greetings." << '\n';  // no flush
}                                // characters transferred; cout not flushed
```

P0053 SYNCHRONIZED BUFFERED OSTREAMS

EXAMPLE: OBTAINING THE WRAPPED STREAM BUFFER WITH GET_WRAPPED() ALLOWS WRAPPING IT AGAIN WITH AN OSYNCSTREAM

```
{
  std::osyncstream out1{std::cout};
  out1 << "Hello, ";
  {
    std::osyncstream{out1.get_wrapped()} << "Goodbye, " << "Planet!" << '\n';
  }
  out1 << "World!" << '\n';
}
```

```
Goodbye, Planet!
Hello, World!
```

P0753 MANIPULATORS FOR C++ SYNCHRONIZED BUFFERED OSTREAM

- In case **basic_osyncstream** is known only via **ostream&**

```
template <class charT, class traits>
std::basic_ostream<charT, traits>& std::emit_on_flush(std::basic_ostream<charT, traits>& os);

template <class charT, class traits>
std::basic_ostream<charT, traits>& std::noemit_on_flush(std::basic_ostream<charT, traits>& os);

template <class charT, class traits>
std::basic_ostream<charT, traits>& std::flush_emit(std::basic_ostream<charT, traits>& os);
```

P0718 std::atomic<std::shared_ptr<T>>

MOTIVATION

- The C++ standard provides an API to access and manipulate specific **shared_ptr** objects atomically

```
auto ptr = std::make_shared<int>(0);
runThreads(5, [&](int i)
{
  std::atomic_store(&ptr, std::make_shared<int>(i));
  return *ptr;
});
```

P0718 std::atomic<std::shared_ptr<T>>

MOTIVATION

- The C++ standard provides an API to access and manipulate specific **shared_ptr** objects atomically

```
auto ptr = std::make_shared<int>(0);
runThreads(5, [&](int i)
{
  std::atomic_store(&ptr, std::make_shared<int>(i));
  return *ptr;
});
```

- Fragile and error-prone
  - **shared_ptr** objects manipulated through this API are *indistinguishable* from other **shared_ptr** objects
  - They may be manipulated/accessed *only through this API* (i.e. you cannot dereference such a **shared_ptr** without first loading it into another **shared_ptr** object, and then dereferencing through the second object)

P0718 std::atomic<std::shared_ptr<T>>

- Merge **atomic_shared_ptr** from Concurrency TS into IS
- Refactor to be **std::atomic** specializations for smart pointers

```
template<class T>
struct std::atomic<std::shared_ptr<T>>;

template<class T>
struct std::atomic<std::weak_ptr<T>>;
```

P0718 std::atomic<std::shared_ptr<T>>

- Merge **atomic_shared_ptr** from Concurrency TS into IS
- Refactor to be **std::atomic** specializations for smart pointers

```
template<class T>
struct std::atomic<std::shared_ptr<T>>;

template<class T>
struct std::atomic<std::weak_ptr<T>>;
```

- The C++11 Atomic Interface for **shared_ptr** is *deprecated*

P0020 FLOATING POINT ATOMIC

- Adds support for atomic addition on an object conforming to the **`std::atomic<T>`** where **`T`** is a **floating-point type**
- Capability critical for parallel high performance computing (HPC) applications
- Explicit specialization for **`float`**, **`double`**, **`long double`** to provide additional atomic operations appropriate to floating-point types

P0840 LANGUAGE SUPPORT FOR EMPTY OBJECTS

MOTIVATION

EBO idiom introduces a number of problems

P0840 LANGUAGE SUPPORT FOR EMPTY OBJECTS

MOTIVATION

EBO idiom introduces a number of problems

- *Limited applicability*
  - EBO is not available for final classes, nor for classes with virtual bases that have non-public destructors

P0840 LANGUAGE SUPPORT FOR EMPTY OBJECTS

MOTIVATION

EBO idiom introduces a number of problems

- *Limited applicability*
  - EBO is not available for final classes, nor for classes with virtual bases that have non-public destructors
- *Name leakage*
  - member names of base classes are visible to users of the derived class (unless shadowed), even if the base class is inaccessible
  - unqualified lookups in code deriving from the class employing EBO is affected by names in the EBO base class

P0840 LANGUAGE SUPPORT FOR EMPTY OBJECTS

MOTIVATION

EBO idiom introduces a number of problems

- *Limited applicability*
  - EBO is not available for final classes, nor for classes with virtual bases that have non-public destructors
- *Name leakage*
  - member names of base classes are visible to users of the derived class (unless shadowed), even if the base class is inaccessible
  - unqualified lookups in code deriving from the class employing EBO is affected by names in the EBO base class
- *Implementation awkwardness*
  - EBO requires state that would naturally be represented as a data member to be moved into a base class

P0840 LANGUAGE SUPPORT FOR EMPTY OBJECTS

SOLUTION

```
template<typename Key, typename Value, typename Hash, typename Pred, typename Allocator>
class hash_map {
  [[no_unique_address]] Hash hasher;
  [[no_unique_address]] Pred pred;
  [[no_unique_address]] Allocator alloc;
  Bucket *buckets;
  // ...
public:
  // ...
};
```

- Unique address is not required for an empty non-static data member of a class
- An empty non-static data member with this attribute will share its address with another object, if it could when used as a base class
- *It is meant to replace EBO Idiom*

P0479 PROPOSED WORDING FOR `likely` AND `unlikely` ATTRIBUTES

MOTIVATION

- *Compiler's optimizers often have no information relating to branch probability* which can lead to suboptimal code generation
- In many cases the *excellent dynamic branch predictors* on modern processors can make up for this lack of information
- However *in some cases code may execute more slowly than necessary* even though the programmer knew the probability of particular branches being executed
- Currently code developers do not have an easy way to communicate this to the compiler

P0479 PROPOSED WORDING FOR `likely` AND `unlikely` ATTRIBUTES

SOLUTION

- The attribute-tokens **`likely`** and **`unlikely`** may be applied to statements
- They shall appear *at most once* in each attribute-list and no attribute-argument-clause shall be present
- The **`likely`** attribute is *not allowed to appear* in the same attribute-list as the **`unlikely`** attribute

P0479 PROPOSED WORDING FOR `likely` AND `unlikely` ATTRIBUTES

- When a **`[[likely]]`** attribute appears in *an **`if`** statement*, implementations are encouraged to optimize for the case where that statement is executed

```
if (foo()) [[likely]] {
  baz();
}
```

P0479 PROPOSED WORDING FOR `likely` AND `unlikely` ATTRIBUTES

- When a **`[[likely]]`** attribute appears in *an **`if`** statement*, implementations are encouraged to optimize for the case where that statement is executed

```
if (foo()) [[likely]] {
  baz();
}
```

- When a **`[[likely]]`** attributes appears in *a nested **`if`** statement*, implementations are encouraged to optimize for the case where that statement is executed

```
if (foo()) {
  if (bar()) [[likely]] {
    baz();
  }
}
```

P0479 PROPOSED WORDING FOR `likely` AND `unlikely` ATTRIBUTES

- When a **`[[likely]]`** attribute appears inside of *a **`switch case`** statement*, implementations are encouraged to optimize for that **`case`** being executed

```
switch (a) {
case 1:
  [[likely]] foo();
  break;
case 2:
  bar();
  break;
default:
  baz();
  break;
}
```

P0479 PROPOSED WORDING FOR `likely` AND `unlikely` ATTRIBUTES

- When an **`[[unlikely]]`** attribute appears inside of *a loop*, implementations are encouraged to optimize for the case where that statement is not executed

```
while (foo()) {
  [[unlikely]] baz();
}
```

P0479 PROPOSED WORDING FOR `likely` AND `unlikely` ATTRIBUTES

- When an **`[[unlikely]]`** attribute appears inside of *a loop*, implementations are encouraged to optimize for the case where that statement is not executed

```
while (foo()) {
  [[unlikely]] baz();
}
```

Excessive usage of either of these attributes is liable to result in performance degradation

P0463 ENDIAN, JUST ENDIAN

TYPE_TRAITS

```
enum class endian
{
  little = __ORDER_LITTLE_ENDIAN__,
  big    = __ORDER_BIG_ENDIAN__,
  native = __BYTE_ORDER__
};
```

P0463 ENDIAN, JUST ENDIAN

TYPE_TRAITS

```
enum class endian
{
  little = __ORDER_LITTLE_ENDIAN__,
  big    = __ORDER_BIG_ENDIAN__,
  native = __BYTE_ORDER__
};
```

```
if(endian::native == endian::big)
  // handle big endian
else if(endian::native == endian::little)
  // handle little endian
else
  // handle mixed endian
```

P0329 DESIGNATED INITIALIZATION

```
struct A {
  int x;
  int y;
  int z;
};
A a{.x = 1, .z = 2};   // OK: a.y initialized to 0
A b{.y = 2, .x = 1};   // Error: designator order does not match declaration order
```

P0329 DESIGNATED INITIALIZATION

```
struct A {
  int x;
  int y;
  int z;
};
A a{.x = 1, .z = 2};   // OK: a.y initialized to 0
A b{.y = 2, .x = 1};   // Error: designator order does not match declaration order
```

```
struct A {
  std::string a;
  int b = 42;
  int c = -1;
};
A a{.c = 21};          // a.a initialized to std::string{}, a.b to 42, a.c to 21
```

P0329 DESIGNATED INITIALIZATION

```
union u {
  int a;
  const char* b;
};
u a = { 1 };
u d = { 0, "asdf" };            // Error
u e = { "asdf" };               // Error
u f = { .b = "asdf" };
u g = { .a = 1, .b = "asdf" };  // Error
```

P0683 DEFAULT MEMBER INITIALIZERS FOR BIT-FIELDS

```
struct S {
  int x : 8 = 42;
};
```

P0614 RANGE-BASED FOR STATEMENTS WITH INITIALIZER

C++17

```
{
  T thing = f();
  for(auto& x : thing.items()) {
    mutate(&x);
    log(x);
  }
}
```

```
{
  for(auto& x : f().items()) {  // WRONG
    mutate(&x);
    log(x);
  }
}
```

P0614 RANGE-BASED FOR STATEMENTS WITH INITIALIZER

C++17

```
{
  T thing = f();
  for(auto& x : thing.items()) {
    mutate(&x);
    log(x);
  }
}
```

```
{
  for(auto& x : f().items()) {  // WRONG
    mutate(&x);
    log(x);
  }
}
```

C++20

```
for(T thing = f(); auto& x : thing.items()) {
  mutate(&x);
  log(x);
}
```

P0614 RANGE-BASED FOR STATEMENTS WITH INITIALIZER

C++17

```
{
  std::size_t i = 0;
  for(const auto& x : foo()) {
    bar(x, i);
    ++i;
  }
}
```

P0614 RANGE-BASED FOR STATEMENTS WITH INITIALIZER

C++17

```
{
  std::size_t i = 0;
  for(const auto& x : foo()) {
    bar(x, i);
    ++i;
  }
}
```

C++20

```
for(std::size_t i = 0; const auto& x : foo()) {
  bar(x, i);
  ++i;
}
```

P0614 RANGE-BASED FOR STATEMENTS WITH INITIALIZER

C++17

```
{
  std::size_t i = 0;
  for(const auto& x : foo()) {
    bar(x, i);
    ++i;
  }
}
```

C++20

```
for(std::size_t i = 0; const auto& x : foo()) {
  bar(x, i);
  ++i;
}
```

- Enables and encourages locally scoped variables without the programmer having to introduce a scope manually

P0457 STRING PREFIX AND SUFFIX CHECKING

- Adds member functions `starts_with()` and `ends_with()` to class templates **`std::basic_string`** and **`std::basic_string_view`**
- Check, whether or not a string starts with a given prefix or ends with a given suffix

P0457 STRING PREFIX AND SUFFIX CHECKING

- Adds member functions `starts_with()` and `ends_with()` to class templates **`std::basic_string`** and **`std::basic_string_view`**
- Check, whether or not a string starts with a given prefix or ends with a given suffix

```
constexpr bool starts_with(basic_string_view x) const noexcept;
constexpr bool starts_with(charT x) const noexcept;
constexpr bool starts_with(const charT* x) const;

constexpr bool ends_with(basic_string_view x) const noexcept;
constexpr bool ends_with(charT x) const noexcept;
constexpr bool ends_with(const charT* x) const;
```

P0550 `std::remove_cvref<T>`

- New *TransformationTrait* for the **`<type_traits>`** header
- Like **`std::decay`**, it *removes any cv and reference qualifiers*
- Unlike **`std::decay`**, it *does not mimic any array-to-pointer or function-to-pointer conversion*

P0550 std::remove_cvref<T>

- New *TransformationTrait* for the **<type_traits>** header
- Like **std::decay**, it *removes any cv and reference qualifiers*
- Unlike **std::decay**, it *does not mimic any array-to-pointer or function-to-pointer conversion*

```
template<typename T>
class optional {
public:
  template<typename U = T,
           enable_if_t<conjunction_v<negation<is_same<optional<T>, remove_cvref_t<U>>>,
                                     negation<is_same<in_place_t, remove_cvref_t<U>>>,
                                     is_constructible<T, U&&>,
                                     is_convertible<U&&, T>
                                    >, bool> = true>
  constexpr optional(U&& t);
  // ...
};
```

P0550 std::remove_cvref<T>

- New *TransformationTrait* for the **<type_traits>** header
- Like **std::decay**, it *removes any cv and reference qualifiers*
- Unlike **std::decay**, it *does not mimic any array-to-pointer or function-to-pointer conversion*

```
template<typename T>
class optional {
public:
  template<typename U = T,
           enable_if_t<conjunction_v<negation<is_same<optional<T>, remove_cvref_t<U>>>,
                                     negation<is_same<in_place_t, remove_cvref_t<U>>>,
                                     is_constructible<T, U&&>,
                                     is_convertible<U&&, T>
                                    >, bool> = true>
  constexpr optional(U&& t);
  // ...
};
```

- Above and more wrong **std::decay** usages fixed with P0777

P0600 [[nodiscard]] ATTRIBUTE IN THE STANDARD LIBRARY

`[[nodiscard]]` attribute applied to

- `async()`
- `allocate()`
- `operator new`
- `launder()`
- `empty()`

P0653 UTILITY TO CONVERT A POINTER TO A RAW POINTER

- **`std::addressof(*p)`** is not well-defined when **`p`** does not reference storage that has an object constructed in it

C++17

```
auto p = a.allocate(1);
std::allocator_traits<A>::construct(a, std::addressof(*p), v);  // WRONG
```

C++20

```
auto p = a.allocate(1);
std::allocator_traits<A>::construct(a, std::to_address(p), v);
```

P0653 UTILITY TO CONVERT A POINTER TO A RAW POINTER

EXAMPLE IMPLEMENTATION

```
template<class T>
T* to_address(T* p) noexcept
{
  return p;
}

template<class Ptr>
auto to_address(const Ptr& p) noexcept
{
  return to_address(p.operator->());
}
```

P0858 CONSTEXPR ITERATOR REQUIREMENTS

MOTIVATION

Intend to make the iterators of some classes usable in constant expressions

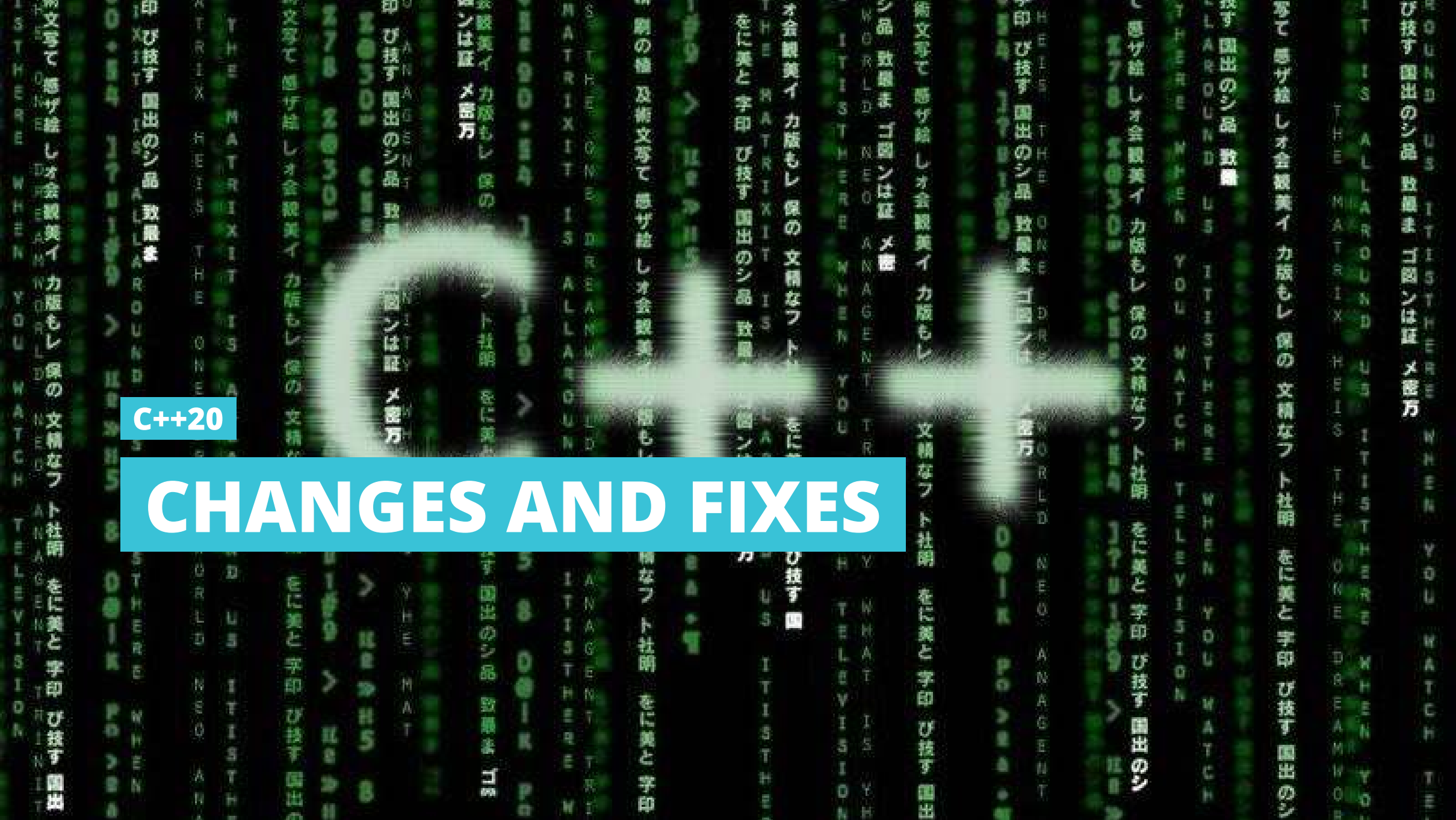
SOLUTION

Introducing the **constexpr iterator requirement** that will easily allow to make constexpr usable iterators by only adding a few words to the iterator requirements of a container

P0306 COMMA OMISSION AND COMMA DELETION

```
#define F(...)       f(0 __VA_OPT__(,) __VA_ARGS__)

F(a, b, c)      // replaced by f(0, a, b, c)
F()             // replaced by f(0)
```


C++20
CHANGES AND FIXES

P0919 HETEROGENEOUS LOOKUP FOR UNORDERED CONTAINERS (WIP)

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 (WIP)

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);
```

P0809 COMPARING UNORDERED CONTAINERS

MOTIVATION

- *The behavior of a program that uses* ***`operator==`*** *or* ***`operator!=`*** *on unordered containers is undefined unless the* ***`Hash`*** *and* ***`Pred`*** *function objects respectively have the same behavior for both containers and the equality comparison function for* ***`Key`*** *is a refinement of the partition into equivalent-key groups produced by* ***`Pred`****.*
- The UB definition for heterogenous containers should not apply merely because of inequity among hashers - and in practice, this may be valuable because of hash seeding and randomization

P0809 COMPARING UNORDERED CONTAINERS

MOTIVATION

- *The behavior of a program that uses* ***`operator==`*** *or* ***`operator!=`*** *on unordered containers is undefined unless the* ***`Hash`*** *and* ***`Pred`*** *function objects respectively have the same behavior for both containers and the equality comparison function for* ***`Key`*** *is a refinement of the partition into equivalent-key groups produced by* ***`Pred`****.*
- The UB definition for heterogenous containers should not apply merely because of inequity among hashers - and in practice, this may be valuable because of hash seeding and randomization

SOLUTION

- *The behavior of a program that uses* ***`operator==`*** *or* ***`operator!=`*** *on unordered containers is undefined unless the* ***`Pred` function object has*** *the same behavior for both containers and the equality comparison operator for* ***`Key`****...*

P0428 FAMILIAR TEMPLATE SYNTAX FOR GENERIC LAMBDAS

P0428 FAMILIAR TEMPLATE SYNTAX FOR GENERIC LAMBDAS

```
[]<typename T>(T x) { /* ... */ }
[]<typename T, int N>(T (&a)[N]) { /* ... */ }
```

P0428 FAMILIAR TEMPLATE SYNTAX FOR GENERIC LAMBDAS

```
[]<typename T>(T x) { /* ... */ }
[]<typename T, int N>(T (&a)[N]) { /* ... */ }
```

```
auto f = [](auto vector) {


  using T =
    typename decltype(vector)::value_type;
  // ...
};
```

P0428 FAMILIAR TEMPLATE SYNTAX FOR GENERIC LAMBDAS

```
[]<typename T>(T x) { /* ... */ }
[]<typename T, int N>(T (&a)[N]) { /* ... */ }
```

```
template<typename T>
struct is_std_vector :
          std::false_type {};
template<typename T>
struct is_std_vector<std::vector<T>> :
          std::true_type {};

auto f = [](auto vector) {
  static_assert(
    is_std_vector<decltype(vector)>::value);
  using T =
    typename decltype(vector)::value_type;
  // ...
};
```

P0428 FAMILIAR TEMPLATE SYNTAX FOR GENERIC LAMBDAS

```
[]<typename T>(T x) { /* ... */ }
[]<typename T, int N>(T (&a)[N]) { /* ... */ }
```

```
template<typename T>
struct is_std_vector :
          std::false_type {};
template<typename T>
struct is_std_vector<std::vector<T>> :
          std::true_type {};

auto f = [](auto vector) {
  static_assert(
    is_std_vector<decltype(vector)>::value);
  using T =
    typename decltype(vector)::value_type;
  // ...
};
```

```
auto f = []<typename T>(std::vector<T> vector) {
  // ...
};
```

P0409 ALLOW LAMBDA CAPTURE [=, THIS]

MOTIVATION

- When both `[=]` and `[=, *this]` are present in a code base, it may be easy to forget that the former is different from the latter

P0409 ALLOW LAMBDA CAPTURE [=, THIS]

MOTIVATION

- When both **[=]** and **[=, *this]** are present in a code base, it may be easy to forget that the former is different from the latter

SOLUTION

```
struct S {
  void f(int i)
  {
    [&, i]{ };          // OK
    [&, this, i]{ };    // OK: equivalent to [&, i]
    [=]{ };             // OK
    [=, *this]{ };      // OK
    [=, this]{ };       // OK: equivalent to [=]
    [this, *this]{ };   // Error: this appears twice
  }
};
```

P0624 DEFAULT CONSTRUCTIBLE AND ASSIGNABLE STATELESS LAMBDAS

LIBRARY.H

```
auto greater = [](auto x, auto y) { return x > y; };
```

USER.CPP

```
// No need to care whether 'greater' is a lambda or a function object
std::map<std::string, int, decltype(greater)> map1;
```

P0624 DEFAULT CONSTRUCTIBLE AND ASSIGNABLE STATELESS LAMBDAS

LIBRARY.H

```
auto greater = [](auto x, auto y) { return x > y; };
```

USER.CPP

```
// No need to care whether 'greater' is a lambda or a function object
std::map<std::string, int, decltype(greater)> map1;
```

```
std::map<std::string, int, decltype(greater)> map2{/* ... */};
map1 = map2;   // OK to assign lambdas
```

P0780 ALLOW PACK EXPANSION IN LAMBDA INIT-CAPTURE

MOTIVATION

- In C++17 *template parameter packs* can only be captured in lambda by copy, by reference, or by...

  `std::tuple`

- *No possibility to do a simple move*

P0780 ALLOW PACK EXPANSION IN LAMBDA INIT-CAPTURE

MOTIVATION

- In C++17 *template parameter packs* can only be captured in lambda by copy, by reference, or by...

  `std::tuple`

- *No possibility to do a simple move*

BY COPY

```
template <class... Args>
auto delay_invoke_foo(Args... args)
{
  return [args...]() -> decltype(auto) {



    return foo(args...);

  };
}
```

P0780 ALLOW PACK EXPANSION IN LAMBDA INIT-CAPTURE

MOTIVATION

- In C++17 *template parameter packs* can only be captured in lambda by copy, by reference, or by...

  `std::tuple`

- *No possibility to do a simple move*

BY COPY

```
template <class... Args>
auto delay_invoke_foo(Args... args)
{
  return [args...]() -> decltype(auto) {



    return foo(args...);

  };
}
```

BY MOVE

```
template <class... Args>
auto delay_invoke_foo(Args... args)
{
  return [tup=std::make_tuple(std::move(args)...)]()
                                          -> decltype(auto) {
           return std::apply([](auto const&... args)
                                          -> decltype(auto) {
                    return foo(args...);
                  }, tup);
         };
}
```

P0780 ALLOW PACK EXPANSION IN LAMBDA INIT-CAPTURE

SOLUTION

- Remove the restriction on pack expansions in init-capture, which requires defining a new form of parameter pack in the language

P0780 ALLOW PACK EXPANSION IN LAMBDA INIT-CAPTURE

SOLUTION

- Remove the restriction on pack expansions in init-capture, which requires defining a new form of parameter pack in the language

C++17

```
template <class... Args>
auto delay_invoke_foo(Args... args)
{
  return [tup=std::make_tuple(std::move(args)...)]()
                                        -> decltype(auto) {
           return std::apply([](auto const&... args)
                                        -> decltype(auto) {
                    return foo(args...);
                  }, tup);
         };
}
```

C++20

```
template <class... Args>
auto delay_invoke_foo(Args... args)
{
  return [args=std::move(args)...]() -> decltype(auto) {


    return foo(args...);

  };
}
```

P0415 constexpr FOR std::complex

MOTIVATION

```
// OK
constexpr std::complex<double> c1{1.0, 0.0};
constexpr std::complex<double> c2{};

// Failure: arithmetic operations on complex are not constexpr
constexpr auto c3 = -c1 + c2 / 100.0;
```

P0202 ADD constexpr MODIFIERS TO FUNCTIONS IN <algorithm> AND <utility> HEADERS

MOTIVATION

```
constexpr std::array<char, 6> a { 'H', 'e', 'l', 'l', 'o' }; // OK

constexpr auto it = std::find(a.rbegin(), a.rend(), 'H'); // ERROR: std::find is not constexpr
```

P0202 ADD constexpr MODIFIERS TO FUNCTIONS IN <algorithm> AND <utility> HEADERS

MOTIVATION

```
constexpr std::array<char, 6> a { 'H', 'e', 'l', 'l', 'o' }; // OK

constexpr auto it = std::find(a.rbegin(), a.rend(), 'H'); // ERROR: std::find is not constexpr
```

SOLUTION

- Add **constexpr** to all algorithms that
  - *do not use* ***std::swap***
  - *do not allocate memory* (**std::stable_partition**, **std::inplace_merge**, and **std::stable_sort**)
  - *do not rely upon* ***std::uniform_int_distribution*** (**std::shuffle** and **std::sample**)

P0616 DE-PESSIMIZE LEGACY <numeric> ALGORITHMS WITH std::move

MOTIVATION

```
std::vector<std::string> v(10000, "hello"s);
std::string s{"start"};
// s.reserve(s.size() + v.size() * v[0].size()); // useless
std::accumulate(begin(v), end(v), s);
```

P0616 DE-PESSIMIZE LEGACY <numeric> ALGORITHMS WITH std::move

MOTIVATION

```
std::vector<std::string> v(10000, "hello"s);
std::string s{"start"};
// s.reserve(s.size() + v.size() * v[0].size()); // useless
std::accumulate(begin(v), end(v), s);
```

SOLUTIONS

- **std::accumulate()** and **std::partial_sum()**

```
acc = std::move(acc) + *i;
```

- **std::inner_product()**

```
acc = std::move(acc) + (*i1) * (*i2);
```

- **std::adjacent_difference()**

P0966 `string::reserve` SHOULD NOT SHRINK

MOTIVATION

- **`basic_string::reserve()`** *optionally shrinks to fit*
- **Performance trap** - can add unexpected and costly dynamic reallocations
- **Portability barrier** - feature optionality may cause different behavior when run against different library implementations
- **Complicates generic code** - generic code which accepts **`vector`** or **`basic_string`** as a template argument must add code to avoid calling **`reserve(n)`** when **`n`** is less than capacity
- **Duplicates functionality** - **`basic_string::shrink_to_fit`**
- **Inconsistent** with **`vector::reserve()`** which does not shrink-to-fit

P0966 `string::reserve` SHOULD NOT SHRINK

MOTIVATION

- **`basic_string::reserve()`** *optionally shrinks to fit*
- **Performance trap** - can add unexpected and costly dynamic reallocations
- **Portability barrier** - feature optionality may cause different behavior when run against different library implementations
- **Complicates generic code** - generic code which accepts **`vector`** or **`basic_string`** as a template argument must add code to avoid calling **`reserve(n)`** when **`n`** is less than capacity
- **Duplicates functionality** - **`basic_string::shrink_to_fit`**
- **Inconsistent** with **`vector::reserve()`** which does not shrink-to-fit

SOLUTION

- Rewording of **`basic_string::reserve()`** to mirror **`vector::reserve()`**

P0551 THOU SHALT NOT SPECIALIZE STD FUNCTION TEMPLATES!

MOTIVATION

- Specializing function templates has proven problematic in practice

```
template<class T> void f(T);    // function template
template<>        void f(int*); // explicit specialization
template<class T> void f(T*);   // function template
```

P0551 THOU SHALT NOT SPECIALIZE STD FUNCTION TEMPLATES!

MOTIVATION

- Specializing function templates has proven problematic in practice

```
template<class T> void f(T);    // function template
template<>        void f(int*); // explicit specialization
template<class T> void f(T*);   // function template
```

```
f(new int{1});
```

- Which function is called?

P0551 THOU SHALT NOT SPECIALIZE STD FUNCTION TEMPLATES!

SOLUTION

- *Allow specialization of class templates* in namespace **`std`** provided that the added declaration *depends on at least one user-defined type*
- **Disallow specializations of function templates** in namespace **`std`**

P0634 DOWN WITH TYPENAME!

MOTIVATION

- In a template declaration or a definition, a *dependent name* that is not a member of the current instantiation *is not considered to be a type* unless the disambiguation keyword **`typename`** is used or unless it was already established as a type name

P0634 DOWN WITH TYPENAME!

MOTIVATION

- In a template declaration or a definition, a *dependent name* that is not a member of the current instantiation *is not considered to be a type* unless the disambiguation keyword **typename** is used or unless it was already established as a type name

```
template<class T, class Allocator = std::allocator<T>>
class my_vector {
public:
  using pointer = typename std::allocator_traits<Allocator>::pointer;
  // ...
};
```

P0634 DOWN WITH TYPENAME!

MOTIVATION

- In a template declaration or a definition, a *dependent name* that is not a member of the current instantiation *is not considered to be a type* unless the disambiguation keyword **typename** is used or unless it was already established as a type name

```
template<class T, class Allocator = std::allocator<T>>
class my_vector {
public:
  using pointer = typename std::allocator_traits<Allocator>::pointer;
  // ...
};
```

but...

```
template<class T>
struct D : T::B { // no typename required here
};
```

P0634 DOWN WITH TYPENAME!

SOLUTION

- Make *typename optional* in a number of commonplace contexts that are known to only permit type names

P0634 DOWN WITH TYPENAME!

SOLUTION

- Make ***typename*** *optional* in a number of commonplace contexts that are known to only permit type names

C++17

```
template<class T>
 typename T::R f(typename T::P);

template<class T>
struct S {
  using Ptr = typename PtrTraits<T>::Ptr;
  typename T::R f(typename T::P p) {
    return static_cast<typename T::R>(p);
  }
  auto g() -> typename S<T*>::Ptr;
};
```

P0634 DOWN WITH TYPENAME!

SOLUTION

- Make ***typename*** *optional* in a number of commonplace contexts that are known to only permit type names

C++17

```
template<class T>
 typename T::R f(typename T::P);

template<class T>
struct S {
  using Ptr = typename PtrTraits<T>::Ptr;
  typename T::R f(typename T::P p) {
    return static_cast<typename T::R>(p);
  }
  auto g() -> typename S<T*>::Ptr;
};
```

C++20

```
template<class T>
T::R f(T::P);

template<class T>
struct S {
  using Ptr = PtrTraits<T>::Ptr;
  T::R f(T::P p) {
    return static_cast<T::R>(p);
  }
  auto g() -> S<T*>::Ptr;
};
```

P0674 EXTENDING MAKE_SHARED TO SUPPORT ARRAYS

```
std::shared_ptr<double[]> p = std::make_shared<double[]>(1024);
```

P0692 ACCESS CHECKING ON SPECIALIZATIONS

- Provides the ability to *specialize* templates on their *private and protected nested* class-types

```
template<class T>
struct trait;

class X {
  class impl;
};

template<>
struct trait<X::impl>;
```

P0767 POD AND std::is_pod<> IS DEPRECATED

MOTIVATION

- POD is a widely-used term
- The fundamental problem with POD is that *it means a large different things to different people*

P0767 POD AND `std::is_pod<>` IS DEPRECATED

MOTIVATION

- **POD** is a widely-used term
- The fundamental problem with POD is that *it means a large different things to different people*
  - *Can I memcpy this thing?*
    - **`std::is_pod<T>`** or **`std::is_trivially_copyable<T>`** are both wrong answers in some cases
    - the correct answer is **`is_trivially_copy_constructible_v<T> && is_trivially_copy_assignable_v<T>`**

P0767 POD AND std::is_pod<> IS DEPRECATED

MOTIVATION

- **POD** is a widely-used term
- The fundamental problem with POD is that *it means a large different things to different people*
  - *Can I memcpy this thing?*
    - **std::is_pod<T>** or **std::is_trivially_copyable<T>** are both wrong answers in some cases
    - the correct answer is **is_trivially_copy_constructible_v<T> && is_trivially_copy_assignable_v<T>**
  - *POD is a struct that can be parsed by both C and C++ compilers?*

```
class Point {
public:
  int x;
  int y;
};
static_assert(std::is_pod_v<Point>);
```

P0439 MAKE std::memory_order A SCOPED ENUMERATION

C++17

```
namespace std {
  typedef enum memory_order {
    memory_order_relaxed, memory_order_consume, memory_order_acquire,
    memory_order_release, memory_order_acq_rel, memory_order_seq_cst
  } memory_order;
}
```

C++20

```
namespace std {
  enum class memory_order : unspecified {
    relaxed, consume, acquire, release, acq_rel, seq_cst
  };
  inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
  inline constexpr memory_order memory_order_consume = memory_order::consume;
  inline constexpr memory_order memory_order_acquire = memory_order::acquire;
  inline constexpr memory_order memory_order_release = memory_order::release;
  inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
  inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;
}
```

P0754 <version>

MOTIVATION

- **<ciso646>** header despite being specified to have no effect is used to determine the library version

P0754 <version>

MOTIVATION

- **<ciso646>** header despite being specified to have no effect is used to determine the library version

SOLUTION

- Standardize a dedicated **<version>** C++ header for this purpose
- Contains only the implementation-defined boilerplate comments which specify various properties of the library such as version and copyright notice
- Provides a place to put other implementation-defined library meta-information which an environment or human reader might find useful
- Ideal place to define the feature test macros

NEXT MEETINGS

DATE	PLACE	SUBJECT
04-09 June 2018	Rapperswil, Switzerland	Introducing big language features including ones with broad library impact
05-10 Nov 2018	San Diego, CA, USA	Last meeting for new proposals to enter EWG/LEWG
18-23 Feb 2019	Kona, HI, USA	Last meeting to promote papers from EWG/LEWG to CWG/LWG **C++20 design is feature-complete**
2019.2	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, Ireland	CD ballot comment resolution
2020.1	TBD	CD ballot comment resolution, C++20 technically finalized, **start DIS ballot**
2020.2	Bulgaria	First meeting of **C++23**


CAUTION
Programming
is addictive
(and too much fun)