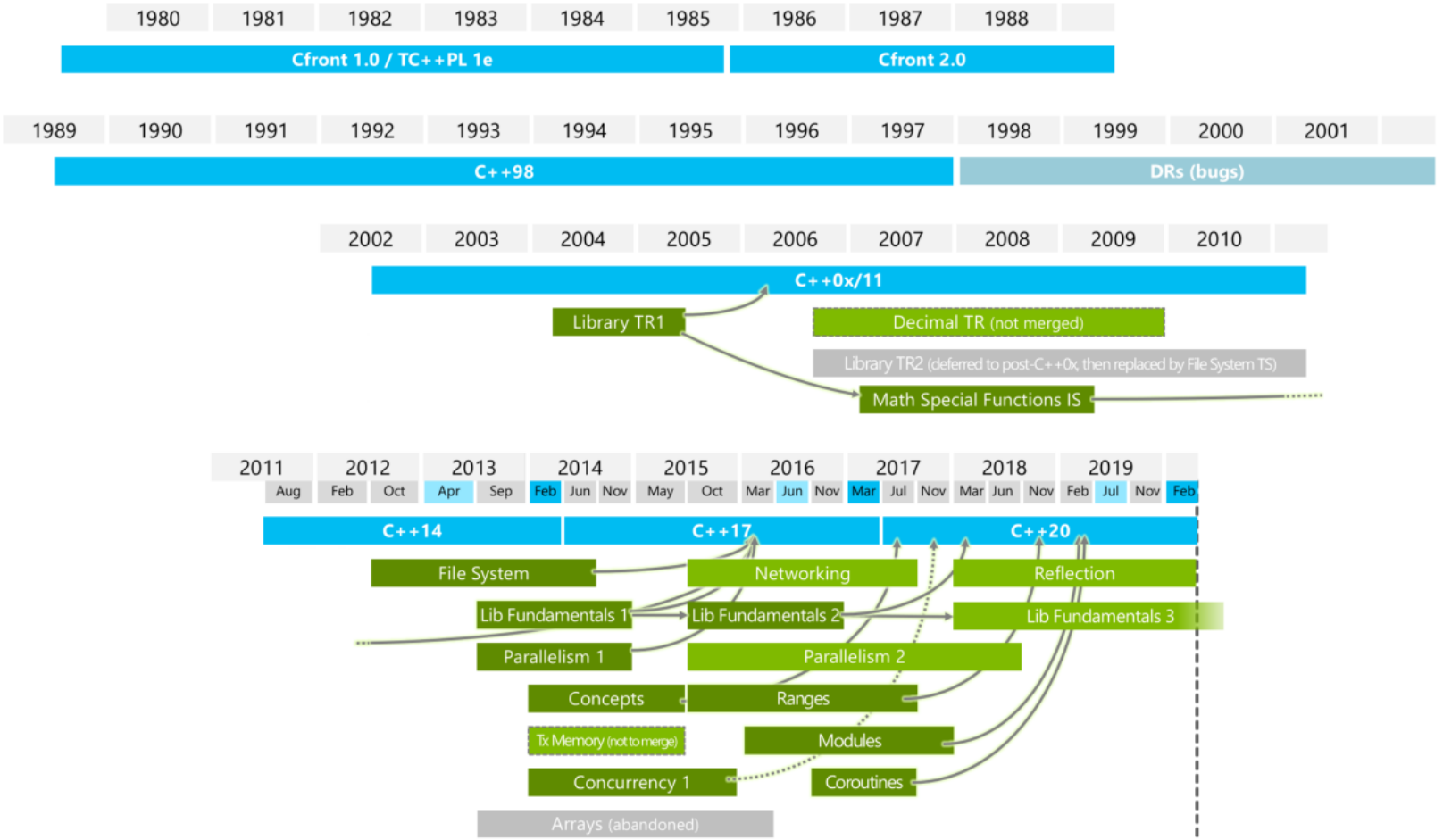
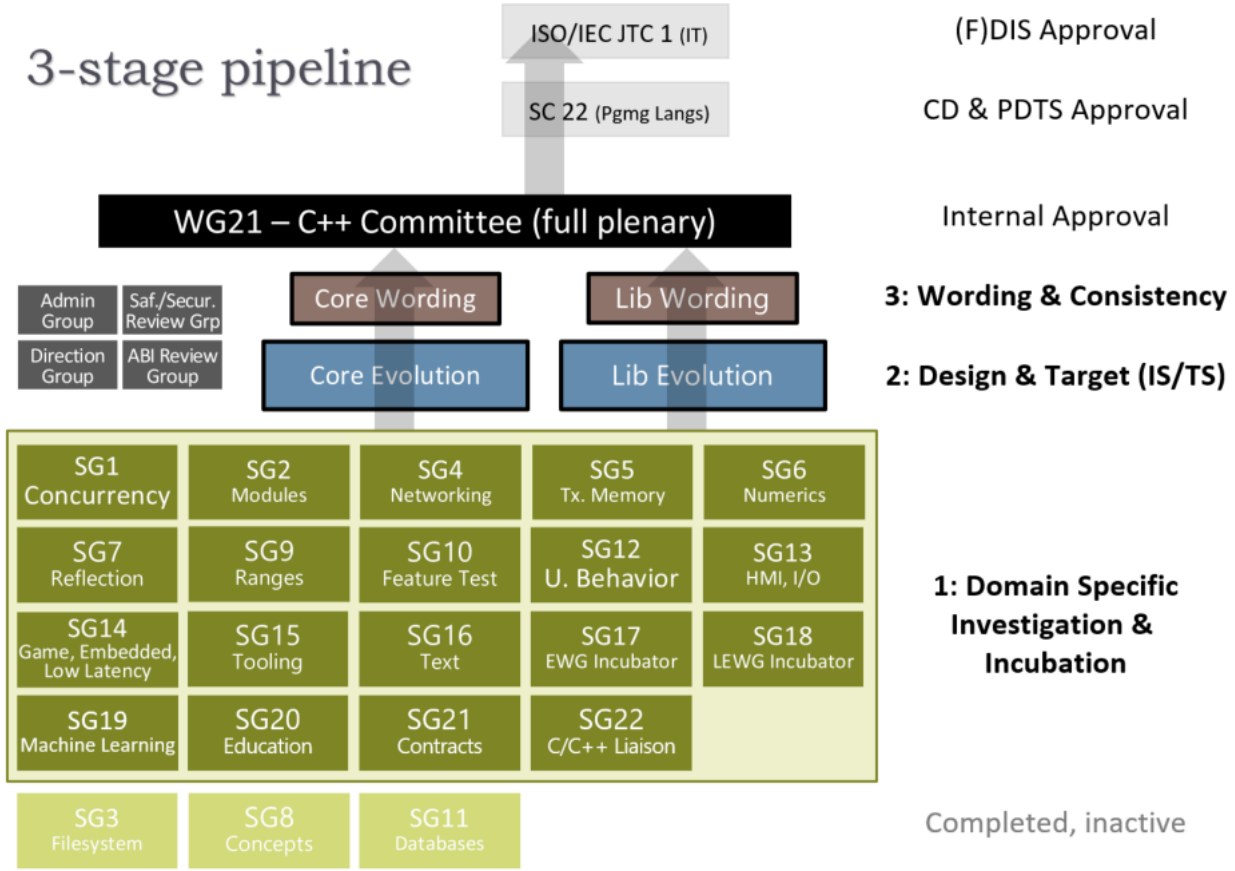

2022
Sneak Peek: C++23
Mateusz Pusz

C++ Timeline

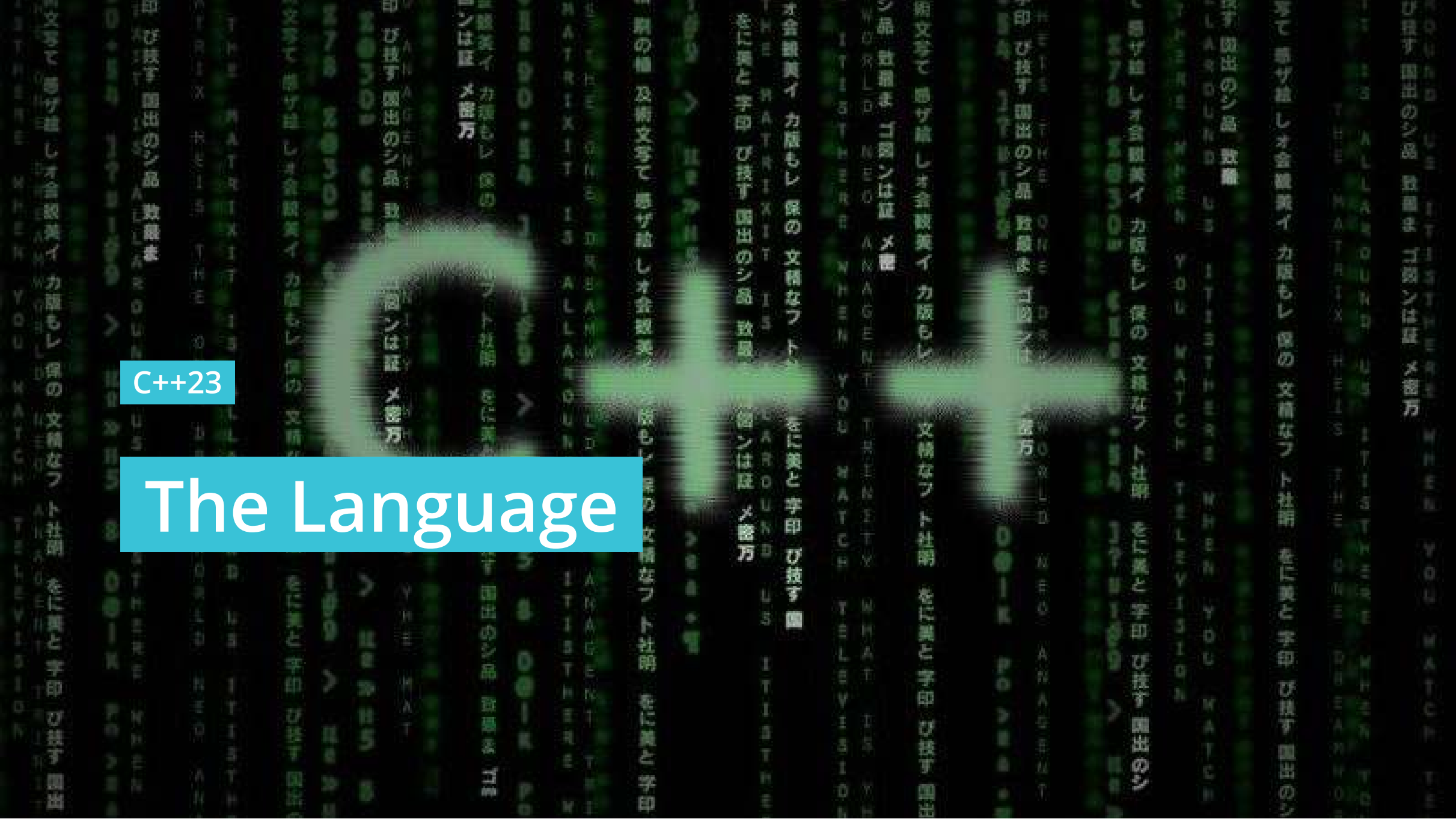

1980 1981 1982 1983 1984 1985 1986 1987 1988
Cfront 1.0 / TC++PL 1e
Cfront 2.0
1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001
C++98
DRs (bugs)
2002 2003 2004 2005 2006 2007 2008 2009 2010
C++0x/11
Library TR1
Decimal TR (not merged)
Library TR2 (deferred to post-C++0x, then replaced by File System TS)
Math Special Functions IS
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 Jul Nov Feb
C++14
C++17
C++20
File System
Networking
Reflection
Lib Fundamentals 1
Lib Fundamentals 2
Lib Fundamentals 3
Parallelism 1
Parallelism 2
Concepts
Ranges
Tx Memory (not to merge)
Modules
Concurrency 1
Coroutines
Arrays (abandoned)

ISO C++ Committee structure



Finding a paper - https://wg21.link

- Usage info
  - wg21.link
- Get paper
  - wg21.link/pXXXX - latest version (e.g. wg21.link/p0463)
  - wg21.link/pXXXXrX
- Get paper status
  - wg21.link/pXXXX/status
- Get working draft
  - wg21.link/std
  - wg21.link/std20
- Get everything
  - wg21.link/index.html


C++23
The Language

P0847 Deducing this

MOTIVATION

```
template<typename T>
class optional {
  // ...
  constexpr const T& value() const & {
    if(has_value()) return this->m_value;
    throw std::bad_optional_access();
  }

  constexpr T& value() & {
    if(has_value()) return this->m_value;
    throw std::bad_optional_access();
  }

  constexpr const T&& value() const && {
    if(has_value()) return std::move(this->m_value);
    throw std::bad_optional_access();
  }
  constexpr T&& value() && {
    if(has_value()) return std::move(this->m_value);
    throw std::bad_optional_access();
  }
};
```

P0847 Deducing this

MOTIVATION

```
template<typename T>
class optional {
  // ...
  constexpr const T& value() const &
  {
    if(has_value()) return this->m_value;
    throw std::bad_optional_access();
  }

  constexpr T& value() &
  {
    return const_cast<T&>(static_cast<const optional&>(*this).value());
  }

  constexpr T&& value() &&
  {
    return const_cast<T&&>(static_cast<const optional&>(*this).value());
  }

  constexpr const T&& value() const &&
  {
    return static_cast<const T&&>(value());
  }
};
```

P0847 Deducing this

MOTIVATION

```
template<typename T>
class optional {
  // ...
  constexpr const T& value() const &   { return value_impl(*this); }
  constexpr T& value() &                { return value_impl(*this); }
  constexpr const T&& value() const && { return value_impl(std::move(*this)); }
  constexpr T&& value() &&              { return value_impl(std::move(*this)); }

private:
  template<typename Opt>
  static decltype(auto) value_impl(Opt&& opt)
  {
    if(opt.has_value()) return std::forward<Opt>(opt).m_value;
    throw std::bad_optional_access();
  }
};
```

P0847 Deducing this

MOTIVATION

```cpp
template<typename T>
class optional {
  // ...
  template<typename Opt>
  friend constexpr decltype(auto) value(Opt&& o)
  {
    if(o.has_value()) return std::forward<Opt>(o).m_value;
    throw std::bad_optional_access();
  }
};
```

P0847 Deducing this

SOLUTION

```
template<typename T>
class optional {
  // ...
  template<typename Self>
  constexpr decltype(auto) value(this Self&& self)
  {
    if(self.has_value()) return std::forward<Self>(self).m_value;
    throw std::bad_optional_access();
  }
};
```

P0847 Deducing this

NAME LOOKUP

C++20

```
struct X {
  // implicit object has type X&
  void foo() &;

  // implicit object has type X const&
  void foo() const &;

  // implicit object has type X&&
  void bar() &&;
};
```

C++23

```
struct X {
  // explicit object has type X&
  void foo(this X&);

  // explicit object has type const X&
  void foo(this const X&);

  // explicit object has type X&&
  void bar(this X&&);
};
```

P0847 Deducing this

NAME LOOKUP

- It is *ill-formed to declare two functions with the same parameters and the same qualifiers* for the object parameter

```
struct X {
  void bar() &&;
  void bar(this X&&); // error: same 'this' parameter type

  static void f();
  void f(this const X&); // error: two functions taking no parameters
};
```

P0847 Deducing this

NAME LOOKUP

- OK as long as any of the *qualifiers are different*

```
struct Y {
  void bar() &;
  void bar() const &;
  void bar(this Y&&);
};
```

P0847 Deducing this

TYPE DEDUCTION

```
struct X {
  template<typename Self>
  void foo(this Self&&, int);
};

struct D : X {};
```

```
x.foo(1);              // Self = X&
std::move(x).foo(2); // Self = X
d.foo(3);              // Self = D&
```

P0847 Deducing this

TYPE DEDUCTION

```
struct X {
  template<typename Self>
  void foo(this Self&&, int);
};

struct D : X {};
```

```
x.foo(1);              // Self = X&
std::move(x).foo(2); // Self = X
d.foo(3);              // Self = D&
```

Deduction is able to deduce a derived type!

P0847 Deducing this

NAME LOOKUP: WITHIN MEMBER FUNCTIONS

```
struct B {
  int i = 0;

  template<typename Self>
  auto&& f1(this Self&&) { return i;  }                          // Error
  template<typename Self>
  auto&& f2(this Self&&) { return this->i; }                     // Error
  template<typename Self>
  auto&& f3(this Self&&) { return std::forward<Self>(*this).i; } // Error
  template<typename Self>
  auto&& f4(this Self&& self) { return std::forward<Self>(self).i; } // OK
};
```

P0847 Deducing this

NAME LOOKUP: WITHIN MEMBER FUNCTIONS

```
struct B {
  int i = 0;

  template<typename Self>
  auto&& f1(this Self&&) { return i;  }                          // Error
  template<typename Self>
  auto&& f2(this Self&&) { return this->i; }                     // Error
  template<typename Self>
  auto&& f3(this Self&&) { return std::forward<Self>(*this).i; } // Error
  template<typename Self>
  auto&& f4(this Self&& self) { return std::forward<Self>(self).i; } // OK
};
```

There is no implicit **`this`** in above functions. All member access must be done directly through the object parameter.

P0847 Deducing this

NAME LOOKUP: WITHIN MEMBER FUNCTIONS

```
struct B {
  int i = 0;

  template<typename Self>
  auto&& f(this Self&& self) { return std::forward<Self>(self).i; }
};

struct D : B {
  // shadows B::i
  double i = 3.14;
};
```

```
B{}.f();  // returns B::i
D{}.f();  // returns D::i
```

P0847 Deducing this

NAME LOOKUP: WITHIN MEMBER FUNCTIONS

```
struct B {
  int i = 0;

  template<typename Self>
  auto&& f(this Self&& self) { return std::forward<Self>(self).B::i; }
};

struct D : B {
  // shadows B::i
  double i = 3.14;
};
```

```
B{}.f();  // returns B::i
D{}.f();  // returns B::i
```

P0847 Deducing this

REPLACING CRTP

```
template<class Derived>
class base {
public:
  void process()
  {
    static_cast<Derived*>(this)->setup();
    static_cast<Derived*>(this)->run();
    static_cast<Derived*>(this)->cleanup();
  }
};

class derived : public base<derived> {
public:
  void setup()   { /* ... */ }
  void run()     { /* ... */ }
  void cleanup() { /* ... */ }
};
```

P0847 Deducing this

REPLACING CRTP

```
template<class Derived>
class base {
public:
  void process()
  {
    static_cast<Derived*>(this)->setup();
    static_cast<Derived*>(this)->run();
    static_cast<Derived*>(this)->cleanup();
  }
};

class derived : public base<derived> {
public:
  void setup()   { /* ... */ }
  void run()     { /* ... */ }
  void cleanup() { /* ... */ }
};
```

```
class base {
public:
  template<typename Self>
  void process(this Self& self)
  {
    self.setup();
    self.run();
    self.cleanup();
  }
};

class derived : public base {
public:
  void setup()   { /* ... */ }
  void run()     { /* ... */ }
  void cleanup() { /* ... */ }
};
```

P0847 Deducing this

RECURSIVE LAMBDAS

```
auto fib = [](this auto self, int n) {
  if(n < 2) return n;
  return self(n-1) + self(n-2);
};
```

P0847 Deducing this

RECURSIVE LAMBDAS

```
struct Leaf {};
struct Node;
using Tree = std::variant<Leaf, Node*>;
struct Node {
  Tree left;
  Tree right;
};

int num_leaves(const Tree& tree)
{
  return std::visit(overloaded( // creates a functor by inheriting from lambdas
      [](const Leaf&) { return 1; },
      [](this const auto& self, Node* n) -> int { // deduced the most derived object
        return std::visit(self, n->left) + std::visit(self, n->right);
      }
  ), tree);
}
```

P0847 Deducing this

BY VALUE this

```
struct my_vector : std::vector<int> {
  my_vector sorted(this my_vector self)
  {
    std::sort(self.begin(), self.end());
    return self;
  }
};
```

- The code above returns a sorted copy of the vector
- Making a *copy in a local variable could be less efficient* in case a function is *called on an rvalue*
  - separate *overloads with different reference specifiers* would be needed to provide similar performance

P0847 Deducing this

BY VALUE this

```
struct less_than {
  template<typename T, typename U>
  bool operator()(this less_than,
                  const T& lhs, const U& rhs)
  {
    return lhs < rhs;
  }
};
```

```
less_than{}(4, 5);
```

P0847 Deducing this

BY VALUE this

```
struct less_than {
  template<typename T, typename U>
  bool operator()(this less_than,
                  const T& lhs, const U& rhs)
  {
    return lhs < rhs;
  }
};
```

```
less_than{}(4, 5);
```

The *implicit object parameter is always a reference*, so any such member functions that do not get inlined incur a *double indirection*. Any member function of small types that does not perform any modifications **can take the object parameter by value to improve performance**.

P0847 Deducing this

BY VALUE this

```
struct C {
  int val;

  generator<int> always() const
  {
    for(;;) co_yield val;
  }
};
```

```
// 'this' ends up being a dangling pointer
for(int i : C{42}.always()) {
  // ...
}
```

P0847 Deducing this

BY VALUE this

```
struct C {
  int val;

  generator<int> always() const
  {
    for(;;) co_yield val;
  }
};
```

```
// 'this' ends up being a dangling pointer
for(int i : C{42}.always()) {
  // ...
}
```

```
struct C {
  int val;

  generator<int> always(this C c)
  {
    for(;;) co_yield c.val;
  }
};
```

```
// OK: C is copied
for(int i : C{42}.always()) {
  // ...
}
```

P0849 auto(x): decay-copy in the language

MOTIVATION

```
void foo(Container auto& x)
{
  std::erase(x, x.front());   // Invalid
}
```

P0849 auto(x): decay-copy in the language

MOTIVATION

```
void foo(Container auto& x)
{
  std::erase(x, x.front());   // Invalid
}
```

```
void foo(Container auto& x)
{
  {
    auto v = x.front();  // new variable in a scope
    std::erase(x, v);    // OK
  }
}
```

P0849 auto(x): decay-copy in the language

MOTIVATION

```
void foo(Container auto& x)
{
  std::erase(x, x.front());   // Invalid
}
```

```
void foo(Container auto& x)
{
  {
    auto v = x.front();  // new variable in a scope
    std::erase(x, v);    // OK
  }
}
```

```
void foo(auto& x)
{
  std::erase(x, typename std::remove_reference_t<decltype(x)>::value_type{x.front()});  // OK
}
```

P0849 auto(x): decay-copy in the language

SOLUTION

```
void foo(Container auto& x)
{
  std::erase(x, auto{x.front()});
}
```

```
void foo(Container auto& x)
{
  std::erase(x, auto(x.front()));
}
```

- Always efficient thanks to mandatory copy elision
  - no copy being made if a prvalue already

P2128 Multidimensional subscript operator

MOTIVATION

```
template<class ElementType, class Extents>
class mdspan {
public:
  template<class... IndexType>
  constexpr reference operator()(IndexType...);
  // ...
};
```

```
int buffer[2 * 3 * 4] = {};
auto span = mdspan<int, extents<2, 3, 4>>(buffer);
span(1, 1, 1) = 42;
```

P2128 Multidimensional subscript operator

SOLUTION

```
template<class ElementType, class Extents>
class mdspan {
public:
  template<class... IndexType>
  constexpr reference operator[](IndexType...);
  // ...
};
```

```
int buffer[2 * 3 * 4] = {};
auto span = mdspan<int, extents<2, 3, 4>>(buffer);
span[1, 1, 1] = 42;
```

P1938 if consteval

MOTIVATION

```
constexpr void clear_bytes(unsigned char *p, int n)
{
  if constexpr(std::is_constant_evaluated()) {
    // when "manifestly constant-evaluated
    for(int k = 0; k < n; ++k) p[k] = 0;
  }
  else {
    std::memset(p, 0, n);  // not a core constant expression
  }
}
```

P1938 `if consteval`

MOTIVATION

```
constexpr void clear_bytes(unsigned char *p, int n)
{
  if constexpr(std::is_constant_evaluated()) {
    // when "manifestly constant-evaluated
    for(int k = 0; k < n; ++k) p[k] = 0;
  }
  else {
    std::memset(p, 0, n);  // not a core constant expression
  }
}
```

`if constexpr()` branch is always a **manifestly constant-evaluated** expressions which means that **`std::is_constant_evaluated()`** will always evaluate to **`true`** in such cases and remove the runtime optimization.

P1938 if consteval

MOTIVATION

```
void clear_bytes(unsigned char *p, int n)
{
  if(std::is_constant_evaluated()) {
    // when "manifestly constant-evaluated
    for(int k = 0; k < n; ++k) p[k] = 0;
  }
  else {
    std::memset(p, 0, n);  // not a core constant expression
  }
}
```

P1938 `if consteval`

MOTIVATION

```
void clear_bytes(unsigned char *p, int n)
{
  if(std::is_constant_evaluated()) {
    // when "manifestly constant-evaluated
    for(int k = 0; k < n; ++k) p[k] = 0;
  }
  else {
    std::memset(p, 0, n);  // not a core constant expression
  }
}
```

Lack of **`constexpr`** in the function signature will always force the **`false`** branch to be taken.

P1938 if consteval

MOTIVATION

```
constexpr void clear_bytes(unsigned char *p, int n)
{
  if(std::is_constant_evaluated()) {
    // when "manifestly constant-evaluated
    for(int k = 0; k < n; ++k) p[k] = 0;
  }
  else {
    std::memset(p, 0, n);  // not a core constant expression
  }
}
```

P1938 if consteval

MOTIVATION

```
consteval int consteval_pow2(int v) { return v * v; }
```

```
template<std::size_t N>
constexpr my_array<N> pow2(const my_array<N>& a)
{
  my_array<N> ret;
  for(int i = 0; const auto& v : a) {
    ret[i++] = std::is_constant_evaluated() ? consteval_pow2(v) : std::pow(v, 2);
  }
  return ret;
}
```

P1938 if consteval

MOTIVATION

```
consteval int consteval_pow2(int v) { return v * v; }
```

```
template<std::size_t N>
constexpr my_array<N> pow2(const my_array<N>& a)
{
  my_array<N> ret;
  for(int i = 0; const auto& v : a) {
    ret[i++] = std::is_constant_evaluated() ? consteval_pow2(v) : std::pow(v, 2);
  }
  return ret;
}
```

```
<source>:28:61: error: the value of 'v' is not usable in a constant expression
   28 |     ret[i++] = std::is_constant_evaluated() ? consteval_pow2(v) : std::pow(v, 2);
      |                                                ~~~~~~~~~~~~~~~^~~
```

P1938 if consteval

```
consteval int consteval_pow2(int v) { return v * v; }
```

```
template<std::size_t N>
constexpr my_array<N> pow2(const my_array<N>& a)
{
  my_array<N> ret;
  for(int i = 0; const auto& v : a) {
    if consteval {
      ret[i++] = consteval_pow2(v);
    }
    else {
      ret[i++] = std::pow(v, 2);
    }
  }
  return ret;
}
```

P1938 `if consteval`

```
if consteval {
  /*...*/
}
else {
  /*...*/
}
```

The braces (in both the **`if`** and the optional **`else`**) are mandatory and there is no condition.

P1938 `if consteval`

- Behaves like

```
if(std::is_constant_evaluated()) { /*...*/ }
```

- With the following differences
  - *no header include* is necessary
  - the syntax is different, which *completely sidesteps the confusion* over the proper way to check if we're in constant evaluation
  - we can use **`if consteval`** to *allow invoking immediate functions*

P1938 `if consteval`

- Negated forms are available

```
if !consteval { /*...*/ }
```

```
if not consteval { /*...*/ }
```

P1938 `if consteval`

- Allows an easy implementation of the original **`std::is_constant_evaluated()`**

```
constexpr bool is_constant_evaluated()
{
  if consteval {
    return true;
  } else {
    return false;
  }
}
```

P2242 Non-literal variables (and labels and gotos) in constexpr functions

MOTIVATION

- In C++20 the constexpr function body **must not contain**
  - a **`goto`** statement
  - labels other than **`case`** and **`default`**
  - variable of *non-literal type*
  - variable of *static* or *thread storage duration*
- The compile-time execution **must not evaluate**
  - Undefined Behavior
  - an **`asm`** declaration
  - **`reinterpret_cast`**
  - **`throw`** expression
  - **`co_await`** and **`co_yield`**

P2242 Non-literal variables (and labels and gotos) in constexpr functions

MOTIVATION

```
template<typename T>
constexpr bool f()
{
  if (std::is_constant_evaluated()) {
    // ...
    return true;
  } else {
    T t;
    // ...
    return true;
  }
}
```

```
struct nonliteral { nonliteral(); };
static_assert(f<nonliteral>());
```

- Rejecting reasonable code
- Inconsistent with the general direction of allowing various constructs in non-constant regions of constexpr functions

P2242 Non-literal variables (and labels and gotos) in constexpr functions

SOLUTION

- The compile-time execution must not evaluate
  - Undefined Behavior
  - variable of *non-literal type*
  - variable of *static* or *thread storage duration*
  - a `goto` statement
  - an **`asm`** declaration
  - **`reinterpret_cast`**
  - **`throw`** expression
  - **`co_await`** and **`co_yield`**

P0330 Literal Suffix for (signed) size_t

MOTIVATION

```cpp
std::vector<int> v{0, 1, 2, 3};
for(auto i = 0u, s = v.size(); i < s; ++i) {
  /* use both i and v[i] */
}
```

P0330 Literal Suffix for (signed) size_t

MOTIVATION

```
std::vector<int> v{0, 1, 2, 3};
for(auto i = 0u, s = v.size(); i < s; ++i) {
  /* use both i and v[i] */
}
```

- Compiles on 32-bit, truncates (maybe with warnings) on 64-bit

```
std::vector<int> v{0, 1, 2, 3};
for(auto i = 0, s = v.size(); i < s; ++i) {
  /* use both i and v[i] */
}
```

- Compilation error

P0330 Literal Suffix for (signed) size_t

SOLUTION

```
std::vector<int> v{0, 1, 2, 3};
for(auto i = 0uz, s = v.size(); i < s; ++i) {
  /* use both i and v[i] */
}
```

P0330 Literal Suffix for (signed) size_t

MOTIVATION

```
auto it = std::find(boost::counting_iterator(0), boost::counting_iterator(v.size()), 3);
```

- Compilation error

P0330 Literal Suffix for (signed) size_t

MOTIVATION

```
auto it = std::find(boost::counting_iterator(0), boost::counting_iterator(v.size()), 3);
```

- Compilation error

SOLUTION

```
auto it = std::find(boost::counting_iterator(0uz), boost::counting_iterator(v.size()), 3uz);
```

P1102 Down with ()!

MOTIVATION

- C++ lambdas with no parameters do not require a parameter declaration clause
- C++20 requires the empty parenthesis when using the **`mutable`** keyword

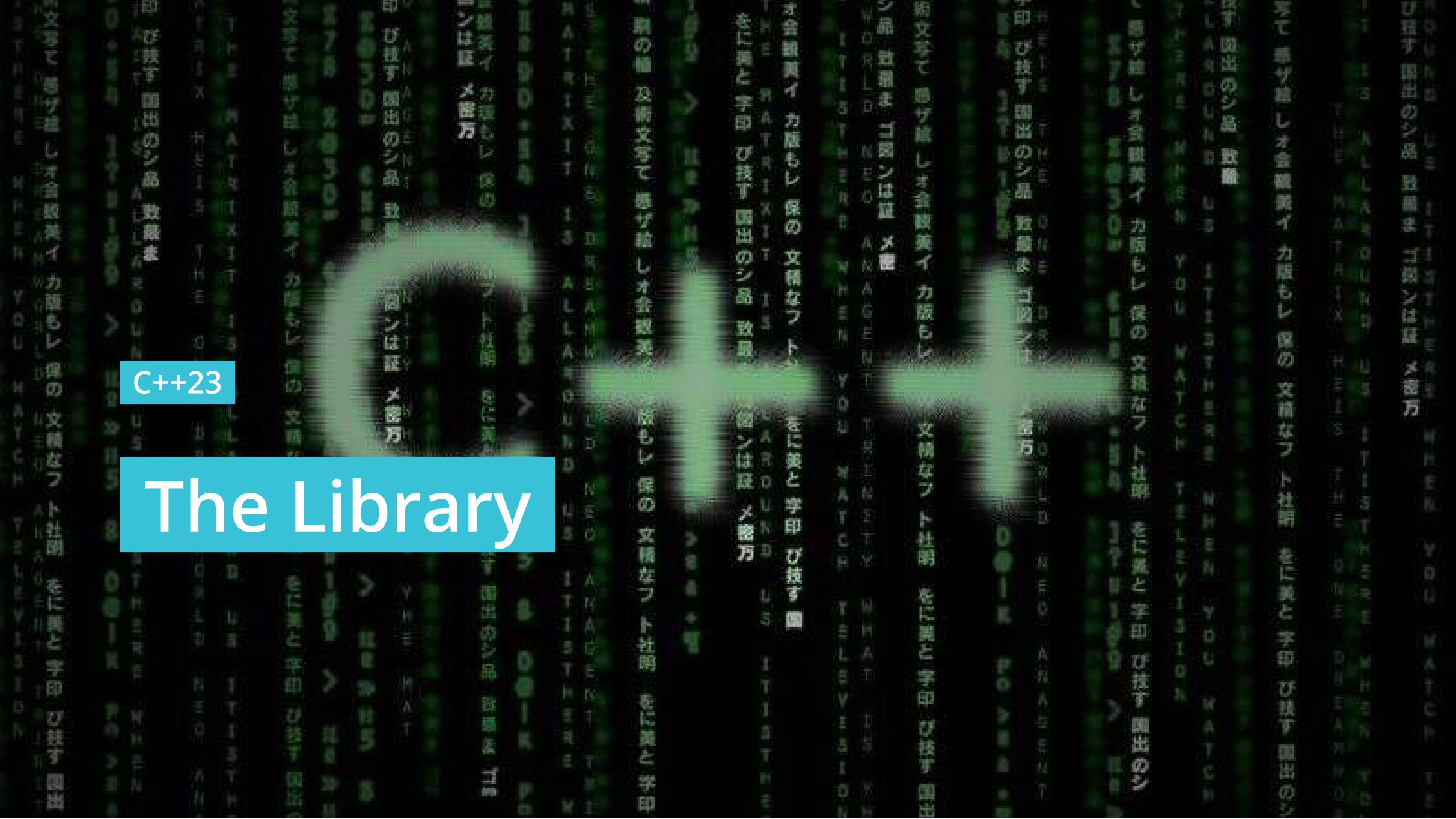
```
std::string s1 = "abc";
auto withParen = [s1 = std::move(s1)] () mutable {
  s1 += "d";
  std::cout << s1 << '\n';
};

std::string s2 = "abc";
auto noParen = [s2 = std::move(s2)] mutable { // Error in C++20
  s2 += "d";
  std::cout << s2 << '\n';
};
```

P1102 Down with ()!

SOLUTION

- Make parentheses unnecessary for
  - lambda template parameters
  - **`constexpr`**
  - **`mutable`**
  - **`consteval`**
  - Exception specifications and **`noexcept`**
  - attributes
  - trailing return types
  - **`requires`**


C++23
The Library

P0798 Monadic operations for `std::optional`

MOTIVATION

```
std::optional<A> to_A(int);
std::optional<B> to_B(A);
std::optional<C> to_C(B);
std::optional<D> to_D(C);
```

P0798 Monadic operations for `std::optional`

MOTIVATION

```
std::optional<A> to_A(int);
std::optional<B> to_B(A);
std::optional<C> to_C(B);
std::optional<D> to_D(C);
```

```
std::optional<D> foo(int v)
{
  auto optA = to_A(v);
  if(optA) {
    auto optB = to_B(*optA);
    if(optB) {
      auto optC = to_C(*optB);
      if(optC) {
        return to_D(*optC);
      }
    }
  }
  return std::nullopt;
}
```

P0798 Monadic operations for `std::optional`

MOTIVATION

```
std::optional<A> to_A(int);
std::optional<B> to_B(A);
std::optional<C> to_C(B);
std::optional<D> to_D(C);
```

```
std::optional<D> foo(int v)
{
  auto optA = to_A(v);
  if(optA) {
    auto optB = to_B(*optA);
    if(optB) {
      auto optC = to_C(*optB);
      if(optC) {
        return to_D(*optC);
      }
    }
  }
  return std::nullopt;
}
```

```
std::optional<D> foo(int v)
{
  auto optA = to_A(v);
  if(!optA)
    return std::nullopt;
  auto optB = to_B(*optA);
  if(!optB)
    return std::nullopt;
  auto optC = to_C(*optB);
  if(!optC)
    return std::nullopt;
  return to_D(*optC);
}
```

P0798 Monadic operations for `std::optional`

SOLUTION

```
std::optional<A> to_A(int);
std::optional<B> to_B(A);
std::optional<C> to_C(B);
std::optional<D> to_D(C);
```

```
std::optional<D> foo(int v)
{
  return to_A(v).and_then(to_B).and_then(to_C).and_then(to_D);
}
```

P0798 Monadic operations for `std::optional`

`.and_then()`

```
std::optional<std::string> opt_s = get();
std::optional<int> opt_i = opt_s.and_then(std::stoi);
```

P0798 Monadic operations for `std::optional`

`.and_then()`

```
std::optional<std::string> opt_s = get();
std::optional<int> opt_i = opt_s.and_then(std::stoi);
```

`.or_else()`

```
get().or_else([]{ std::cout << "get() failed"; });
get().or_else([]{ throw std::runtime_error("get() failed") });
```

P0798 Monadic operations for `std::optional`

`.and_then()`

```
std::optional<std::string> opt_s = get();
std::optional<int> opt_i = opt_s.and_then(std::stoi);
```

`.or_else()`

```
get().or_else([]{ std::cout << "get() failed"; });
get().or_else([]{ throw std::runtime_error("get() failed") });
```

`.transform()`

```
std::optional<std::string> opt_s = get();
std::optional<std::string::size_type> = opt_s.transform([](auto&& s) { return s.size(); });
```

P0323 `std::expected`

```
template<class T, class E>
class expected;
```

- **Represents either a valid value or an error** that you can pass by value
- Like a **`std::variant`** that stores either **`T`** or **`E`**
- Unlike **`std::variant`** it provides a *never-empty guarantee*
- Has an API of **`std::optional`** with **`T`** being "expected" and **`E`** being "unexpected" thing
- Partial specialization for **`std::expected<void, E>`**
- **Never allocates memory**

P0323 std::expected: Interface (Simplification)

```
template<class T, class E>
class expected {
  bool has_val;
  union { value_type val; unexpected_type unex; };
public:
  using value_type = T;
  using error_type = E;
  using unexpected_type = unexpected<E>;

  constexpr expected();
  template<class U>
  explicit constexpr expected(U&& v);
  template<class... Args>
  constexpr explicit expected(in_place_t, Args&&...);

  template<class G>
  constexpr expected(const unexpected<G>&);
  template<class G>
  constexpr expected(unexpected<G>&&);
  template<class... Args>
  constexpr explicit expected(unexpect_t, Args&&...);

  // ...
```

```
  // ...

  constexpr T* operator->();
  constexpr T& operator*();

  constexpr explicit operator bool() const noexcept;
  constexpr bool has_value() const noexcept;

  template<typename Self>
  constexpr auto&& value(this Self&& self);
  template<typename Self>
  constexpr auto&& error(this Self&& self);

  template<class U>
  T value_or(U&&);

  bool operator==(const expected&);
  bool operator==(const T&);
  bool operator==(const unexpected<E>&);

  // ...
};
```

P0323 std::expected: Interface (Simplification)

```
template<class E>
class unexpected {
  E val;
public:
  template<class... Args>
  constexpr explicit unexpected(in_place_t, Args&&...);
  template<class Err = E>
  constexpr explicit unexpected(Err&&);

  template<typename Self> constexpr auto&& value(this Self&& self);

  bool operator==(const unexpected&);

  // ...
};
```

P0323 std::expected

EXAMPLE

```
std::expected<int, std::string> to_int(std::string_view txt)
{
  int result;
  const auto end = txt.data() + txt.size();
  auto [ptr, ec] = std::from_chars(txt.data(), end, result);
  if(ec == std::errc() && ptr == end)
    return result;
  else
    return std::unexpected(std::format("'{}' isn't a number", txt));
}
```

P0323 std::expected

EXAMPLE

```
std::expected<int, std::string> to_int(std::string_view txt)
{
  int result;
  const auto end = txt.data() + txt.size();
  auto [ptr, ec] = std::from_chars(txt.data(), end, result);
  if(ec == std::errc() && ptr == end)
    return result;
  else
    return std::unexpected(std::format("'{}' isn't a number", txt));
}
```

```
auto value = to_int(text);
if(value)
  std::cout << "'" << text << "' is " << *value << '\n';
else
  std::cerr << "Error: " << value.error() << '\n';
```

P0323 std::expected

EXAMPLE

```
struct handle;        // abstract base class for some handle implementation
class handle_ref;     // some sort of smart pointer managing a handle*

std::expected<handle_ref, std::error_code> open_file(const char* path) noexcept
{
  int fd = open(path, O_RDONLY);
  if(fd == -1)
    return std::unexpected(std::error_code(errno, std::system_category()));

  std::error_code ec;
  auto* p = new(std::nothrow) derived_handle{fd, ec};
  if(p == nullptr) {
    close(fd);
    return std::unexpected(std::error_code(std::errc::not_enough_memory));
  }

  return handle_ref(p);
}
```

P0881 A Proposal to add stacktrace library

MOTIVATION

```
boost/array.hpp:123: T& boost::array<T, N>::operator[](boost::array<T, N>::size_type): Assertion '(i < N)&&("out of range")' faile
Aborted (core dumped)
```

P0881 A Proposal to add stacktrace library

MOTIVATION

```
boost/array.hpp:123: T& boost::array<T, N>::operator[](boost::array<T, N>::size_type): Assertion '(i < N)&&("out of range")' faile
Aborted (core dumped)
```

- In the C++20 there is *no way to get, store and decode the current call sequence*
- Such call sequences are *useful for debugging* and post mortem debugging
- They are popular in other programming languages
- **Assertions often can't describe the whole picture of a bug and do not provide enough information to locate the problem**

P0881 A Proposal to add stacktrace library

SOLUTION

```
Expression 'i < N' is false in function 'T& boost::array<T, N>::operator[](boost::array<T, N>::size_type)': out of range.
Backtrace:
 0# boost::assertion_failed_msg(char const*, char const*, char const*, char const*, long) at ../example/assert_handler.cpp:39
 1# boost::array<int, 5ul>::operator[](unsigned long) at ../../../boost/array.hpp:124
 2# bar(int) at ../example/assert_handler.cpp:17
 3# foo(int) at ../example/assert_handler.cpp:25
 4# bar(int) at ../example/assert_handler.cpp:17
 5# foo(int) at ../example/assert_handler.cpp:25
 6# main at ../example/assert_handler.cpp:54
 7# 0x00007F991FD69F45 in /lib/x86_64-linux-gnu/libc.so.6
 8# 0x0000000000401139
```

P0881 A Proposal to add stacktrace library

- Based on the Boost.Stacktrace library
- Does not attempt to provide a signal-safe solution for capturing and decoding stacktraces
  - functionality currently not implementable on some of the popular platforms
- All the **`stack_frame`** *functions and constructors are lazy* and won't decode the pointer information if there was no explicit request from class user
  - fast way to store stacktrace, without decoding the function names (improves performance)
- *Custom allocators support provided*

P0881 A Proposal to add stacktrace library

```
template<class Allocator>
class basic_stacktrace;

using stacktrace = std::basic_stacktrace<std::allocator<std::stacktrace_entry>>;
```

- P2301 Add a **`pmr`** alias for **`std::stacktrace`**

```
namespace pmr {

using stacktrace =
  std::basic_stacktrace<std::pmr::polymorphic_allocator<std::stacktrace_entry>>;

}
```

P0881 A Proposal to add stacktrace library

```
class stacktrace_entry {
public:
  using native_handle_type = implementation-defined;

  constexpr stacktrace_entry() noexcept;
  constexpr stacktrace_entry(const stacktrace_entry& other) noexcept;
  constexpr stacktrace_entry& operator=(const stacktrace_entry& other) noexcept;
  ~stacktrace_entry();

  constexpr native_handle_type native_handle() const noexcept;
  constexpr explicit operator bool() const noexcept;

  std::string description() const;
  std::string source_file() const;
  std::uint_least32_t source_line() const;

  friend constexpr bool operator==(const stacktrace_entry& x,
                                   const stacktrace_entry& y) noexcept;
  friend constexpr std::strong_ordering operator<=>(const stacktrace_entry& x,
                                                    const stacktrace_entry& y) noexcept;
};
```

P0881 A Proposal to add stacktrace library

```
template<class Allocator>
class basic_stacktrace {
public:
  static basic_stacktrace current(const allocator_type& alloc = allocator_type()) noexcept;
  static basic_stacktrace current(size_type skip,
                                  const allocator_type& alloc = allocator_type()) noexcept;
  static basic_stacktrace current(size_type skip, size_type max_depth,
                                  const allocator_type& alloc = allocator_type()) noexcept;

  // random-access-container-like interface
  // ...
};
```

P0881 A Proposal to add stacktrace library

```
std::string to_string(const std::stacktrace_entry& f);

template<class Allocator>
std::string to_string(const std::basic_stacktrace<Allocator>& st);
```

P0881 A Proposal to add stacktrace library

```
std::string to_string(const std::stacktrace_entry& f);

template<class Allocator>
std::string to_string(const std::basic_stacktrace<Allocator>& st);
```

```
template<class charT, class traits>
std::basic_ostream<charT, traits>& operator<<(std::basic_ostream<charT, traits>& os,
                                              const std::stacktrace_entry& f);

template<class charT, class traits, class Allocator>
std::basic_ostream<charT, traits>& operator<<(std::basic_ostream<charT, traits>& os,
                                              const std::basic_stacktrace<Allocator>& st);
```

P1132 out_ptr - a scalable output pointer abstraction

MOTIVATION

```
#include <memory>
#include <avformat.h>

struct AVFormatContextDeleter {
  void operator()(AVFormatContext* c) const noexcept
  {
    avformat_close_input(&c);
    avformat_free_context(c);
  }
};
using av_format_context_ptr = std::unique_ptr<AVFormatContext, AVFormatContextDeleter>;
```

```
int avformat_open_input(AVFormatContext **ps, const char *url,
                        AVInputFormat *fmt, AVDictionary **options);
```

P1132 out_ptr - a scalable output pointer abstraction

MOTIVATION

```cpp
int main (int, char* argv[])
{
  av_format_context_ptr context(avformat_alloc_context());
  // ... tweak the context

  // user-supplied context will be freed on failure
  AVFormatContext* raw_context = context.release();
  if(avformat_open_input(&raw_context, argv[0], nullptr, nullptr) != 0) {
    std::stringstream ss;
    ss << "ffmpeg_image_loader could not open file '" << path << "'";
    throw FFmpegInputException(ss.str().c_str());
  }
  context.reset(raw_context);

  // ... use the stream
}
```

P1132 out_ptr - a scalable output pointer abstraction

SOLUTION

```
int main (int, char* argv[])
{
  av_format_context_ptr context(avformat_alloc_context());
  // ... tweak the context

  // user-supplied context will be freed on failure

  if(avformat_open_input(std::inout_ptr(context), argv[0], nullptr, nullptr) != 0) {
    std::stringstream ss;
    ss << "ffmpeg_image_loader could not open file '" << path << "'";
    throw FFmpegInputException(ss.str().c_str());
  }


  // ... use the stream
}
```

P1132 `out_ptr` - a scalable output pointer abstraction

- Abstractions to *make C APIs and smart pointers cooperate nicely*
- A smart pointer *convertible to a* ***`T**`*** *that updates* (with a reset call or semantically equivalent behavior) *the smart pointer it is created with when it goes out of scope*
- Automates the **`.reset()`** and the **`.release()`** calls for smart pointers when interfacing with **`T**`** output arguments
- *Users are allowed to override this type* for their own shared, unique, handle-alike, reference-counting, and similar smart pointers

P1132 out_ptr - a scalable output pointer abstraction

```
error_num c_api_create_handle(int seed_value, int** p_handle);
error_num c_api_re_create_handle(int seed_value, int** p_handle);
void c_api_delete_handle(int* handle);

struct resource_deleter {
  void operator()( int* handle ) {
    c_api_delete_handle(handle);
  }
};
```

```
std::unique_ptr<int, resource_deleter> res;
error_num err = c_api_create_handle(
  24, std::out_ptr(res)
);
if(err == C_API_ERROR_CONDITION) {
  // handle errors
}
```

```
std::unique_ptr<int, resource_deleter> res;
error_num err = c_api_re_create_handle(
  24, std::inout_ptr(res)
);
if(err == C_API_ERROR_CONDITION) {
  // handle errors
}
```

P1132 `out_ptr` - a scalable output pointer abstraction

```
template<class Smart, class Pointer, class... Args>
class out_ptr_t {
public:
  out_ptr_t(Smart&, Args...) noexcept(/* conditional */);
  ~out_ptr_t();
  operator Pointer*() const noexcept;
  operator void**() const noexcept; // only if Pointer != void*
};

template<class Pointer = void, class Smart, class... Args>
out_ptr_t<Smart, conditional_t<is_void_v<Pointer>, POINTER_OF(Smart), Pointer>, Args&&...>
out_ptr(Smart& s, Args&&... args) noexcept;
```

- *Destructor calls* ***`.reset()`*** *on the stored smart pointer* of type **`Smart`** with the stored pointer of type **`Pointer`** and arguments stored as **`Args...`**.

P1132 out_ptr - a scalable output pointer abstraction

```
template<class Smart, class Pointer, class... Args>
class inout_ptr_t {
public:
  inout_ptr_t(Smart&, Args...) noexcept(/* conditional */);
  ~inout_ptr_t();
  operator Pointer*() const noexcept;
  operator void**() const noexcept; // only if Pointer != void*
};

template<class Pointer = void, class Smart, class... Args>
inout_ptr_t<Smart, conditional_t<is_void_v<Pointer>, POINTER_OF(Smart), Pointer>, Args&&...>
inout_ptr(Smart& s, Args&&... args) noexcept;
```

- *Destructor calls* ***`.reset()`*** *on the stored smart pointer* of type **`Smart`** with the stored pointer of type **`Pointer`** and arguments stored as **`Args...`**.
- The *constructor calls* ***`.release()`*** so that a reset doesn't double-delete in case the raw pointer API also releases the resource

P1989 Range constructor for `std::string_view`

MOTIVATION

- A lot of codebases have *custom string types that could benefit from being convertible to* ***`std::string_view`***
- *Manipulating the contents of a vector or an array as a string* is also useful
- Makes *contiguous views operating on characters* easier to use with **`std::string_view`**

P1989 Range constructor for `std::string_view`

BEFORE

```
using namespace std::literals;
constexpr std::string_view words{"Introduction; to; C++23; !"};
for(auto word : std::views::split(words, "; "sv))
  std::cout << std::quoted(std::string_view(word.begin(), word.end())) << ' ';
  // prints: "Introduction" "to" "C++23" "!"
```

P1989 Range constructor for `std::string_view`

BEFORE

```
using namespace std::literals;
constexpr std::string_view words{"Introduction; to; C++23; !"};
for(auto word : std::views::split(words, "; "sv))
  std::cout << std::quoted(std::string_view(word.begin(), word.end())) << ' ';
  // prints: "Introduction" "to" "C++23" "!"
```

AFTER

```
using namespace std::literals;
constexpr std::string_view words{"Introduction; to; C++23; !"};
for(std::string_view word : std::views::split(words, "; "sv))
  std::cout << std::quoted(word) << ' ';
  // prints: "Introduction" "to" "C++23" "!"
```

P1659 `starts_with` and `ends_with`

MOTIVATION

- C++20 introduced
  - **`std::basic_string_views::starts_with()`**, **`std::basic_string_views::ends_with()`**
  - **`std::basic_string::starts_with()`**, **`std::basic_string::ends_with()`**
- String types have a large set of member functions that either
  - **duplicate the algorithms**
  - are directly **incompatible with them**, and thus limit the amount of composition that's possible
- *Not immediately possible to query whether or not any range* (other than a standard string type) *is prefixed or suffixed by another range*
  - to do so, one must use **`std::mismatch`** or **`std::equal`**, and at least in the case of **`ends_with`**, **`std::ranges::next`**

P1659 starts_with and ends_with

BEFORE

```
auto some_ints = std::views::iota(0, 50);
auto some_more_ints = std::views::iota(0, 30);
if(std::ranges::mismatch(some_ints, some_more_ints).in2 == std::ranges::end(some_more_ints)) {
  // ...
}
```

P1659 starts_with and ends_with

BEFORE

```
auto some_ints = std::views::iota(0, 50);
auto some_more_ints = std::views::iota(0, 30);
if(std::ranges::mismatch(some_ints, some_more_ints).in2 == std::ranges::end(some_more_ints)) {
  // ...
}
```

AFTER

```
auto some_ints = std::views::iota(0, 50);
auto some_more_ints = std::views::iota(0, 30);
if(std::ranges::starts_with(some_ints, some_more_ints)) {
  // ...
}
```

P1659 starts_with and ends_with

BEFORE

```
auto some_ints = std::views::iota(0, 50);
auto some_more_ints = std::views::iota(0, 30);
assert(std::ranges::distance(some_ints) >= std::ranges::distance(some_more_ints));
auto some_ints_tail = std::ranges::subrange{std::ranges::next(std::ranges::begin(some_ints),
    std::ranges::distance(some_ints) - std::ranges::distance(some_more_ints),
    std::ranges::end(some_ints)), std::ranges::end(some_ints)};
if(!std::ranges::equal(some_ints_tail, some_more_ints)) {
  // ...
}
```

P1659 starts_with and ends_with

BEFORE

```
auto some_ints = std::views::iota(0, 50);
auto some_more_ints = std::views::iota(0, 30);
assert(std::ranges::distance(some_ints) >= std::ranges::distance(some_more_ints));
auto some_ints_tail = std::ranges::subrange{std::ranges::next(std::ranges::begin(some_ints),
    std::ranges::distance(some_ints) - std::ranges::distance(some_more_ints),
    std::ranges::end(some_ints)), std::ranges::end(some_ints)};
if(!std::ranges::equal(some_ints_tail, some_more_ints)) {
  // ...
}
```

AFTER

```
auto some_ints = std::views::iota(0, 50);
auto some_more_ints = std::views::iota(0, 30);
if(!std::ranges::ends_with(some_ints, some_more_ints)) {
  // ...
}
```

P2440 ranges::iota, ranges::shift_left, and ranges::shift_right

```cpp
std::list<int> l(10);
std::ranges::iota(l.begin(), l.end(), -4);

std::vector<std::list<int>::iterator> v(l.size());
std::ranges::iota(v, l.begin());

std::ranges::shuffle(v, std::mt19937{std::random_device{}()});

std::cout << "Contents of the list: ";
for(auto n : l) std::cout << n << ' ';
std::cout << '\n';

std::cout << "Contents of the list, shuffled: ";
for(auto i : v) std::cout << *i << ' ';
std::cout << '\n';
```

```
Contents of the list: -4 -3 -2 -1 0 1 2 3 4 5
Contents of the list, shuffled: 0 -1 3 4 -4 1 -2 -3 2 5
```

P2440 `ranges::iota`, `ranges::shift_left`, and `ranges::shift_right`

- The **`ranges::iota`** algorithm *isn't redundant* to **`views::iota`**
- The algorithm *determines the number of values to write based on the output range*
- Using **`ranges::copy`** or similar algorithms with **`views::iota`** requires that information to be computed upfront
- When we already have a range, ***`ranges::iota`*** *can be more convenient and efficient*

P2440 ranges::iota, ranges::shift_left, and ranges::shift_right

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7};
std::cout << v << '\n';
std::ranges::shift_left(v, 3);
std::cout << v << '\n';
std::ranges::shift_right(v, 2);
std::cout << v << '\n';
std::ranges::shift_left(v, 8);
std::cout << v << '\n';
```

```
1 2 3 4 5 6 7
4 5 6 7 5 6 7
4 5 4 5 6 7 5
4 5 4 5 6 7 5
```

P2321 `zip`

- Four views
  - **`std::views::zip`**
  - **`std::views::zip_transform`**
  - **`std::views::adjacent`**
  - **`std::views::adjacent_transform`**
- Changes to **`std::tuple`** and **`std::pair`** necessary to make them usable as proxy references
- Changes to **`std::vector<bool>::reference`** to make it usable as a proxy reference for writing

P2321 zip

```
std::vector v1 = { 1, 2 };
std::vector v2 = { 'a', 'b', 'c' };
std::vector v3 = { 3, 4, 5 };

fmt::print("{}\n", std::views::zip(v1, v2));
fmt::print("{}\n", std::views::zip_transform(std::multiplies(), v1, v3));
fmt::print("{}\n", v2 | std::views::adjacent);
fmt::print("{}\n", v3 | std::views::adjacent_transform(std::plus()));
```

```
{(1, 'a'), (2, 'b')}
{3, 8}
{('a', 'b'), ('b', 'c')}
{7, 9}
```

P2441 `views::join_with`

- Given a range `r` and a pattern `p`

```
r | views::split(p) | views::join_with(p)
```

should yield a range consisting of the same elements as `r`

P2441 `views::join_with`

- Given a range `r` and a pattern `p`

```
r | views::split(p) | views::join_with(p)
```

should yield a range consisting of the same elements as `r`

EXAMPLE

```
std::vector v{"Introduction"sv, "to"sv, "C++23!"sv};
auto joined = v | std::views::join_with(' ');

for(auto c : joined) std::cout << c;
std::cout << '\n';
```

```
Introduction to C++23!
```

P2442 Windowing range adaptors: `views::chunk` and `views::slide`

```cpp
std::vector v = {1, 2, 3, 4, 5};
fmt::print("{}\n", v | std::views::chunk(2));
fmt::print("{}\n", v | std::views::slide(2));
```

```
[[1, 2], [3, 4], [5]]
[[1, 2], [2, 3], [3, 4], [4, 5]]
```

P2443 views::chunk_by

```
std::vector v = {1, 2, 2, 3, 0, 4, 5, 2};
fmt::print("{}\n", v | std::views::chunk_by(std::ranges::less_equal{}));
```

```
[[1, 2, 2, 3], [0, 4, 5], [2]]
```

P2443 views::chunk_by

```
std::vector v = {1, 2, 2, 3, 0, 4, 5, 2};
fmt::print("{}\n", v | std::views::chunk_by(std::ranges::less_equal{}));
```

```
[[1, 2, 2, 3], [0, 4, 5], [2]]
```

Similar to **split**, **chunk_by** calculates the end of the first range in its **begin** and caches it to meet the amortized **O(1)** requirement. That means that it does not support const-iteration.

P1206 Conversions from ranges to containers

MOTIVATION

```
std::list<int> lst = /*...*/;
std::vector<int> vec {std::begin(lst), std::end(lst)};
```

P1206 Conversions from ranges to containers

MOTIVATION

```
std::list<int> lst = /*...*/;
std::vector<int> vec {std::begin(lst), std::end(lst)};
```

SOLUTION

```
std::vector<int> vec = lst | std::ranges::to<std::vector>();
```

P1206 Conversions from ranges to containers

MOTIVATION

```
auto view = std::views::iota(0, 42);
std::vector<std::iter_value_t<std::ranges::iterator_t<decltype(view)>>> vec;
if constexpr(std::ranges::sized_range<decltype(view)>)
  vec.reserve(std::ranges::size(view));
std::ranges::copy(view, std::back_inserter(vec));
```

P1206 Conversions from ranges to containers

MOTIVATION

```
auto view = std::views::iota(0, 42);
std::vector<std::iter_value_t<std::ranges::iterator_t<decltype(view)>>> vec;
if constexpr(std::ranges::sized_range<decltype(view)>)
  vec.reserve(std::ranges::size(view));
std::ranges::copy(view, std::back_inserter(vec));
```

SOLUTION

```
auto vec = std::views::iota(0, 42) | std::ranges::to<std::vector>();
```

P1206 Conversions from ranges to containers

MOTIVATION

```
std::map<int, widget> map = get_widgets();
std::vector<decltype(map)::value_type> vec;
vec.reserve(map.size());
std::ranges::move(map, std::back_inserter(vec));
```

P1206 Conversions from ranges to containers

MOTIVATION

```
std::map<int, widget> map = get_widgets();
std::vector<decltype(map)::value_type> vec;
vec.reserve(map.size());
std::ranges::move(map, std::back_inserter(vec));
```

SOLUTION

```
auto vec = get_widgets() | std::ranges::to<std::vector>();
```

P1206 Conversions from ranges to containers

- **`ranges::to<C>(r, args)`** returns an instance **`c`** of **`C`** constructed *by the first valid option*
  - construct **`c`** *from* **`r`**
  - construct **`c`** from **`r`** using the *tagged ranged constructor* (**`from_range_t`**)
  - construct **`c`** from the *pair of iterators* **`std::ranges::begin(r)`**, **`std::ranges::end(r)`**
  - construct **`c`**, then *insert each element* of **`r`** at the end of **`c`**
  - if **`C`** is a range whose value type is itself a range (and is not a view), and **`r`**'s value type is also a range, the application of **`to<range_value_t<C>>`** for each element of **`r`** is inserted at the end of **`c`**
- Any *additional arguments passed to* ***`to`*** *are forwarded as the last parameters of the constructor*
  - allocators, comparators, hasher, ...

P1206 Conversions from ranges to containers

- **`ranges::to<C>(r, args)`** returns an instance **`c`** of **`C`** constructed *by the first valid option*
  - construct **`c`** *from* **`r`**
  - construct **`c`** from **`r`** using the *tagged ranged constructor* (**`from_range_t`**)
  - construct **`c`** from the *pair of iterators* **`std::ranges::begin(r)`**, **`std::ranges::end(r)`**
  - construct **`c`**, then *insert each element* of **`r`** at the end of **`c`**
  - if **`C`** is a range whose value type is itself a range (and is not a view), and **`r`**'s value type is also a range, the application of **`to<range_value_t<C>>`** for each element of **`r`** is inserted at the end of **`c`**
- Any *additional arguments passed to* ***`to`*** *are forwarded as the last parameters of the constructor*
  - allocators, comparators, hasher, ...

Tagged constructors offer an opportunity for containers to provide a more efficient implementation.

P1206 Conversions from ranges to containers

```
std::vector vec(std::ranges::from_range, range1);
vec.append_range(range2);
```

P1206 Conversions from ranges to containers

```
std::vector vec(std::ranges::from_range, range1);
vec.append_range(range2);
```

- For any constructor or member function taking a pair of **`InputIterators`** in containers a *similar member function, suffixed with **`_range`** is added taking a range instead*
  - exception: **`std::regex`** and **`std::filesystem::path`**
- Additionally, **`.prepend_range(R)`** and **`.append_range(R)`** are *added when an equivalent **`.push_front(T)`** or **`.push_back(T)`** exists*
  - both **preserve the order of elements**
- All added **constructors are tagged** with **`from_range_t`**

P2387 Pipe support for user-defined range adaptors

- Range adaptor closure objects *have to inherit* from the class template

  `std::ranges::range_adaptor_closure<T>`

  - for **`viewable_range R`**, and range adaptor closure objects **`C`** and **`D`**
    - **`C(R)`** and **`R | C`** are equivalent
    - **`R | C | D`** and **`R | (C | D)`** are equivalent
- *New function adaptor* **`std::bind_back`**
  - **`std::bind_back(f, ys...)(xs...)`** is equivalent to **`f(xs..., ys...)`**

P2387 Pipe support for user-defined range adaptors

```
template<typename F>
class closure : public std::ranges::range_adaptor_closure<closure<F>> {
  F f_;
public:
  constexpr closure(F f) : f_(std::move(f)) {}

  template<std::ranges::viewable_range R>
    requires std::invocable<const F&, R>
  constexpr operator()(R&& r) const
  {
    return f_(std::forward<R>(r));
  }
};
```

P2387 Pipe support for user-defined range adaptors

```
template<typename F>
class adaptor {
  F f_;
public:
  constexpr adaptor(F f) : f_(std::move(f)) {}

  template<typename... Args>
  constexpr operator()(Args&&... args) const
  {
    if constexpr (std::invocable<const F&, Args...>) {
      return f_(std::forward<Args>(args)...);
    } else {
      return closure(std::bind_back(f_, std::forward<Args>(args)...));
    }
  }
};
```

P2387 Pipe support for user-defined range adaptors

```
inline constexpr closure join =
    []<std::ranges::viewable_range R> requires /* ... */
    (R&& r) {
      return std::ranges::join_view(std::forward<R>(r));
    };
```

P2387 Pipe support for user-defined range adaptors

```
inline constexpr closure join =
    []<std::ranges::viewable_range R> requires /* ... */
    (R&& r) {
      return std::ranges::join_view(std::forward<R>(r));
    };
```

```
inline constexpr adaptor transform =
    []<std::ranges::viewable_range R, typename F> requires /* ... */
    (R&& r, F&& f) {
      return std::ranges::transform_view(std::forward<R>(r), std::forward<F>(f));
    };
```

P0627 Function to mark unreachable code

MOTIVATION

```
[[noreturn]] void kill_self()
{
  kill(getpid(), SIGKILL);
}
```

P0627 Function to mark unreachable code

MOTIVATION

```
[[noreturn]] void kill_self()
{
  kill(getpid(), SIGKILL);
}
```

```
<source>: In function 'void kill_self()':
<source>:4:58: warning: 'noreturn' function does return
    4 | [[noreturn]] void kill_self() { kill(getpid(), SIGKILL); }
      |                                                          ^
```

P0627 Function to mark unreachable code

SOLUTION

```
[[noreturn]] void kill_self()
{
  kill(getpid(), SIGKILL);
  std::unreachable();
}
```

P0627 Function to mark unreachable code

SOLUTION

```
[[noreturn]] void kill_self()
{
  kill(getpid(), SIGKILL);
  std::unreachable();
}
```

- *Compilers cannot know every situation in which code may execute*
- When the programmer knows that a situation is impossible, but it is not obvious to the compiler, it is helpful to be able to tell the compiler to *avoid runtime checking for a case that is impossible*

P0627 Function to mark unreachable code

MOTIVATION

```
void do_something(int zero_to_three)
{
  switch(zero_to_three) {
  case 0:
  case 2:
    handle_0_or_2();
    break;
  case 1:
    handle_1();
    break;
  case 3:
    handle_3();
    break;
  }
}
```

```
cmp eax, 4
jae skip_switch
lea rcx, [jump_table]
jmp qword [rcx + rax*8]
```

P0627 Function to mark unreachable code

MOTIVATION

```
void do_something(int zero_to_three)
{
  switch(zero_to_three) {
  case 0:
  case 2:
    handle_0_or_2();
    break;
  case 1:
    handle_1();
    break;
  case 3:
    handle_3();
    break;
  default:
    std::unreachable();
  }
}
```

```
lea rcx, [jump_table]
jmp qword [rcx + rax*8]
```

P0448 A `strstream` replacement using `span<charT>` as buffer

MOTIVATION

- **`strstream`** that can use pre-allocated buffers has been *deprecated for a long time*
  - waiting for a replacement that can *use a fixed size pre-allocated buffer* (e.g. allocated on the stack), that is used *as the stream buffers internal storage*

P0448 A `strstream` replacement using `span<charT>` as buffer

- Reading input from a fixed pre-arranged character buffer

```
char input[] = "10 20 30";
std::ispanstream is{std::span{input}};

int i;
is >> i;
ASSERT_EQUAL(10, i);

is >> i;
ASSERT_EQUAL(20, i);

is >> i;
ASSERT_EQUAL(30, i);

is >> i;
ASSERT(!is);
```

P0448 A `strstream` replacement using `span<charT>` as buffer

- Writing to a fixed pre-arranged character buffer

```
char output[30]{}; // zero-initialize array
std::ospanstream os{std::span{output}};

os << 10 << 20 << 30;
const auto sp = os.span();

ASSERT_EQUAL(6, sp.size());
ASSERT_EQUAL("102030", std::string_view(sp));
ASSERT_EQUAL(static_cast<void*>(output), sp.data()); // no copying of underlying data
ASSERT_EQUAL("102030", std::string_view(output)); // initialization guaranteed NULL termination
```

P1679 string contains function

MOTIVATION

- iterators-like

```
std::string haystack = "no place for needles";
if(haystack.find("needle") != std::string::npos) {
  // ...
}
```

P1679 string contains function

MOTIVATION

- iterators-like

```
std::string haystack = "no place for needles";
if(haystack.find("needle") != std::string::npos) {
  // ...
}
```

- C-like

```
std::string haystack = "no place for needles";
if(strstr(haystack.c_str(), "needle")) {
  // ...
}
```

P1679 string contains function

SOLUTION

```
std::string haystack = "no place for needles";
if(haystack.contains("needle")) {
  // ...
}
```

P1679 string contains function

SOLUTION

```
std::string haystack = "no place for needles";
if(haystack.contains("needle")) {
  // ...
}
```

- Adds the following overloads to class templates **`std::basic_string`** and **`std::basic_string_view`**

```
constexpr bool contains(std::basic_string_view<charT, traits> str) const noexcept;
constexpr bool contains(charT ch) const noexcept;
constexpr bool contains(const charT* str) const;
```

P1048 A proposal for a type trait to detect scoped enumerations

```
class A {};
enum E {};
enum struct Es { oz };
enum class Ec : int {};
```

```
std::cout << std::boolalpha;
std::cout << std::is_scoped_enum_v<int> << '\n';  // false
std::cout << std::is_scoped_enum_v<A> << '\n';    // false
std::cout << std::is_scoped_enum_v<E> << '\n';    // false
std::cout << std::is_scoped_enum_v<Es> << '\n';   // true
std::cout << std::is_scoped_enum_v<Ec> << '\n';   // true
```

P1682 `std::to_underlying` for enumerations

MOTIVATION

- Many codebases write a version of a small utility function converting an enumeration to its underlying type
- Typical *casts can mask potential bugs from size/signed-ness changes* and hide programmer intent

```
enum class my_enum {
  A = 0x1012,
  B = 0x405324,
  C = A & B
};

void do_work(my_enum value)
{
  internal_untyped_api(static_cast<int>(value)); // OK
}
```

P1682 `std::to_underlying` for enumerations

MOTIVATION

- Many codebases write a version of a small utility function converting an enumeration to its underlying type
- Typical *casts can mask potential bugs from size/signed-ness changes* and hide programmer intent

```
enum class my_enum : std::uint32_t {
  A = 0x1012,
  B = 0x405324,
  C = A & B,
  D = 0xFFFFFFFF
};

void do_work(my_enum value)
{
  internal_untyped_api(static_cast<int>(value)); // !!
}
```

P1682 `std::to_underlying` for enumerations

MOTIVATION

- Many codebases write a version of a small utility function converting an enumeration to its underlying type
- Typical *casts can mask potential bugs from size/signed-ness changes* and hide programmer intent

```
enum class my_enum : std::uint32_t {
  A = 0x1012,
  B = 0x405324,
  C = A & B,
  D = 0xFFFFFFFF
};

void do_work(my_enum value)
{
  internal_untyped_api(static_cast<std::underlying_type_t<my_enum>>(value)); // OK
}
```

P1682 `std::to_underlying` for enumerations

MOTIVATION

- Many codebases write a version of a small utility function converting an enumeration to its underlying type
- Typical *casts can mask potential bugs from size/signed-ness changes* and hide programmer intent

```
enum class my_enum : std::uint32_t {
  A = 0x1012,
  B = 0x405324,
  C = A & B,
  D = 0xFFFFFFFF
};

void do_work(other_enum value)
{
  internal_untyped_api(static_cast<std::underlying_type_t<my_enum>>(value)); // !!
}
```

P1682 `std::to_underlying` for enumerations

SOLUTION

```cpp
#include <utility>

void do_work(some_enum value)
{
  internal_untyped_api(std::to_underlying(value)); // Always OK
}
```

P0533 constexpr for <cmath> and <cstdlib>

- std::abs()
- std::abs()
- std::ceil()
- std::copysign()
- std::div()
- std::fabs()
- std::fdim()
- std::floor()
- std::fma()
- std::fmax()
- std::fmin()
- std::fmod()
- std::fpclassify()
- std::frexp()
- std::ilogb()
- std::isfinite()
- std::isgreater()
- std::isgreaterequal()
- std::isinf()
- std::isless()
- std::islessequal()
- std::islessgreater()

P0533 constexpr for <cmath> and <cstdlib>

- std::isnan()
- std::isnormal()
- std::isunordered()
- std::ldexp()
- std::logb()
- std::modf()
- std::nextafter()
- std::nexttoward()
- std::remainder()
- std::remquo()
- std::round()
- std::scalbln()
- std::scalbn()
- std::signbit()
- std::trunc()


CAUTION
Programming
is addictive
(and too much fun)