# C++17 FEATURES

## AN INTRODUCTION TO C++17 VIA INSPIRING EXAMPLES

**Bryce Adelstein Lelbach <bryce@cppnow.org>**

*Material from:*

*Tony Van Eerd, JF Bastien, Thomas Köppe, Anthony Calandra, Bartlomiej Filipek,*

*Steve Lorimer, Arthur O'Dwyer, Jonas Devlieghere, Louis Dionne, Francisco Lopes,*

*Marius Bancila, cppreference.com, Jason Turner, Simon Brand*

brycelelbach.github.io/cpp17_features

Presented at C++Now 2017. Copyright (C) 2017.

## Language Changes

- Structured bindings
- Selection statements with initializers
- Compile-time conditional statments
- Fold expressions
- Class template deduction
- `auto` non-type template parameters
- `inline` variables
- `constexpr` lambdas
- Unary `static_assert`
- Guaranteed copy elision
- Nested namespace definitions
- Preprocessor predicate for header testing
- ...

## Library Changes

- `string_view`
- `optional`
- `variant`
- `any`
- Parallel algorithms
- Filesystem support
- Polymorphic allocators and memory resources
- Aligned `new`
- Improved insertion and splicing for associative containers
- Math special functions
- Variable templates for metafunctions
- Boolean logic metafunctions
- ...

# Before

```
struct point_3d { double x, y, z; };

point_3d p = // ...

double  x = p.x;
double  y = p.y;
double  z = p.z;
```

# Before

```
struct point_3d { double x, y, z; };

point_3d p = // ...

double& x = p.x;
double& y = p.y;
double& z = p.z;
```

# Before

```
std::tuple<double, double, double> t = // ...
double        x,   y,   z;
std::tie(x, y, z) = t;
```

# Before

```
std::tuple<double, double, double> t = // ...

double        x,  y,  z;

std::tie(y, y, z) = t;
       // ^ UH-OH: No warning for repeated names.
```

# Before

```
std::tuple<double, double, double> t = // ...

double       &x, &y, &z;   // COMPILE ERROR
                           // Uninitialized refs.
std::tie(x, y, z) = t;
```

# Before

```
std::tuple<double, double, double> t = // ...

double const x,  y,  z;

std::tie(x, y, z) = t; // COMPILE ERROR
                       // Assignment to const.
```

# Before

```
point_3d p = // ...
double x, y, z;
std::tie(x, y, z) = p; // COMPILE ERROR.
                       // p isn't a std::tuple.

std::array<double, 3> c = // ...
double x, y, z;
std::tie(x, y, z) = c; // COMPILE ERROR.
                       // c isn't a std::tuple.
```

# C++17

```
point_3d p = // ...
auto [x, y, z] = p;

std::tuple<double, double, double> t = // ...
auto [x, y, z] = t;

std::array<double, 3> c = // ...
auto [x, y, z] = c;
```

```
auto [a, b, ...] = obj;
```

The type of `obj` must be `Destructurable`:

- Either all non-static data members:
  - Must be public.
  - Must be direct members of the type or members of the same public base class of the type.
  - Cannot be anonymous unions.
- Or the type has:
  - An `obj.get<>` method or an ADL-able `get<>` overload.
  - Specializations of `std::tuple_size<>` and `std::tuple_element<>`.

```
auto [a, b, ...] = obj;
```

- `Destructurable` types in the standard library:
  - `std::array`
  - `std::tuple`
  - `std::pair`
- Uses regular `auto` deduction rules (`auto const`, `auto&`, `auto&&`, etc).

```
template <typename Key, typename Value, typename F>
void update(std::map<Key, Value>& m, F f)
{
  for (auto&& [key, value] : m)
    value = f(key);
}
```

## Before

```
template <typename Key, typename Value,
          typename F>
void find_and_update(std::map<Key, Value>& m,
                     F f)
{
  auto it = map.find(key);
  if (it != m.end())
    it->second = f(it->first);
}
```

## Before

```cpp
template <typename Key, typename Value,
          typename F>
void find_and_update(std::map<Key, Value>& m,
                     F f)
{
  auto it = map.find(key);
  if (it != m.end())
    it->second = f(it->first);
}
```

## C++17

```cpp
template <typename Key, typename Value,
          typename F>
void find_and_update(std::map<Key, Value>& m,
                     F f)
{
  if (auto it = map.find(key); it != m.end())
    it->second = f(it->first);

}
```

**Syntax**

```
if (init; cond)
  statement1;
else
  statement2;
```

**Equivalent To**

```
{
  init;
  if (cond)
    statement1;
  else
    statement2;
}
```

```
if        (T0 x = /* ... */; condition(x)) {
  // x is in scope here.
} else if (T1 y = /* ... */; condition(y)) {
  // x and y are in scope here.
} else {
  // x and y are in scope here.
}
```

```cpp
struct string_pool
{
  std::string pop(std::size_t new_cap);
  // ...
 private:
  std::mutex mtx_;
  std::string pool_;
};
```

# Before

```
std::string string_pool::pop(std::size_t new_cap)
{
  std::string s;

  std::lock_guard<std::mutex> l(mtx_);
  if (!pool_.empty())
  {
      std::swap(s, pool_.back());
      pool_.pop_back();
  }

  if (s.capacity() < new_cap)
    s.reserve(new_cap); // Unnecessary contention.
                        // mtx_ still locked.

  return s;
}
```

# Before

```
std::string string_pool::pop(std::size_t new_cap)
{
  std::string s;

  {
    std::lock_guard<std::mutex> l(mtx_);
    if (!pool_.empty())
    {
        std::swap(s, pool_.back());
        pool_.pop_back();
    }
  }

  if (s.capacity() < new_cap)
    s.reserve(new_cap);


  return s;
}
```

# C++17

```
std::string string_pool::pop(std::size_t new_cap)
{
  std::string s;


  if (std::lock_guard<std::mutex> l(mtx_); !pool_.empty())
  {
      std::swap(s, pool_.back());
      pool_.pop_back();
  }


  if (s.capacity() < new_cap)
    s.reserve(new_cap);


  return s;
}
```

```cpp
template <typename Key, typename F>
void emplace_or_throw(std::set<Key>& m, Key&& k, F f)
{
  if (auto [it, s] = m.emplace(std::forward<Key>(k)); !s)
    throw /* ... */
  else
    f(*it);
}
```

**Syntax**

```
switch (init; cond)
{
  case a: statement1;
  case b: statement2;
  // ...
}
```

**Equivalent To**

```
{
  init;
  switch (cond)
  {
    case a: statement1;
    case b: statement2;
    // ...
  }
}
```

```
template <typename... Ts>
void print(Ts&&... ts);
```

## Before

```cpp
template <typename T0>
void print(T0&& t0)
{
  std::cout << std::forward<T0>(t0) << "\n";
}

template <typename T0, typename... Ts>
void print(T0&& t0, Ts&&... ts)
{
  print(std::forward<T0>(t0));
  print(std::forward<Ts>(ts)...);
}
```

## Before

```cpp
template <typename T0>
void print(T0&& t0)
{
  std::cout << std::forward<T0>(t0) << "\n";
}

template <typename T0, typename... Ts>
void print(T0&& t0, Ts&&... ts)
{
  print(std::forward<T0>(t0));
  print(std::forward<Ts>(ts)...);
}
```

## C++17

```cpp
template <typename T0, typename... Ts>
void print(T0&& t0, Ts&&... ts)
{
  std::cout << std::forward<T0>(t0) << "\n";

  if constexpr (sizeof...(ts))
    print(std::forward<T0>(ts)...);
}
```

```
if constexpr (cond1)
  statement1;
else if constexpr (cond2)
  statement2;
// ...
else
  statementN;
```

- Compile-time conditional statements.
- The condition must be a `constexpr` expression.
- Statements are discarded if their branch is not taken.
  - Discarded statements can use variables that are declared but not defined.
  - Discarded statements in templates are not instantiated.

## Before

```
template <typename T0>
void print(T0&& t0)
{
  std::cout << std::forward<T0>(t0) << "\n";
}

template <typename T0, typename... Ts>
void print(T0&& t0, Ts&&... ts)
{
  print(std::forward<T0>(t0));
  print(std::forward<Ts>(ts)...);
}
```

## C++17

```
template <typename T0, typename... Ts>
void print(T0&& t0, Ts&&... ts)
{
  std::cout << std::forward<T0>(t0) << "\n";

  if constexpr (sizeof...(ts))
    print(std::forward<T0>(ts)...);
}
```

# Before

```
template <typename T, typename... Args>
std::enable_if_t<
    std::is_constructible_v<T, Args...>, std::unique_ptr<T>
>
make_unique(Args&&... a)
{
  return std::unique_ptr(new T(std::forward<Args>(a)...));
}

template <typename T, typename... Args>
std::enable_if_t<
  !std::is_constructible_v<T, Args...>, std::unique_ptr<T>
>
make_unique(Args&&... a)
{
  return std::unique_ptr(new T{std::forward<Args>(a)...});
}
```

# C++17

```
template <typename T, typename... Args>
auto make_unique(Args&&... a)
{
  if constexpr (std::is_constructible_v<T, Args...>)
    return std::unique_ptr(new T(std::forward<Args>(a)...));
  else
    return std::unique_ptr(new T{std::forward<Args>(a)...});
}
```

# Before

```
template <typename Iterator, typename Dist>
void advance(Iterator& i, Dist n)
{
  typename std::iterator_traits<Iterator>::iterator_category c;
  advance_impl(i, n, c);
}

template <typename Iterator, typename Distance>
void advance_impl(Iterator& i, Dist n, std::random_access_iterator_tag)
{
  i += n;
}

template <typename Iterator, typename Dist>
void advance_impl(Iterator& i, Dist n, std::bidirectional_iterator_tag)
{
  if (n >= 0) while (n--) ++i;
  else        while (n++) --i;
}

template <typename Iterator, typename Dist>
void advance_impl(Iterator& i, Dist n, std::input_iterator_tag)
{
  while (n--) ++i;
}
```

# C++17

```
template <typename Iterator, typename Dist>
void advance(Iterator& i, Dist n)
{
  typename std::iterator_traits<Iterator>::iterator_category c;

  if constexpr (std::is_same_v<c, std::random_access_iterator_tag>)
    i += n;

  else if constexpr (std::is_same_v<c, std::bidirectional_iterator_tag>)
  {
    if (n >= 0) while (n--) ++i;
    else        while (n++) --i;
  }

  else // std::input_iterator_tag
    while (n--) ++i;
}
```

```
struct person
{
  std::uint64_t& get_id();
  std::string&   get_name();
  std::uint16_t& get_age();
 private:
  std::uint64_t id_;
  std::string   name_;
  std::uint16_t age_;
};
```

# Before

```
template <std::size_t I>
auto& get(person& p);

template <>
auto& get<0>(person& p)
{
  return p.get_id();
}

template <>
auto& get<1>(person& p)
{
  return p.get_name();
}

template <>
auto& get<2>(person& p)
{
  return p.get_age();
}
```

## Before

```
template <std::size_t I>
auto& get(person& p);

template <>
auto& get<0>(person& p)
{
  return p.get_id();
}

template <>
auto& get<1>(person& p)
{
  return p.get_name();
}

template <>
auto& get<2>(person& p)
{
  return p.get_age();
}
```

## C++17

```
template <std::size_t I>
auto& get(person& p)
{
  if constexpr      (I == 0)
    return p.get_id();
  else if constexpr (I == 1)
    return p.get_name();
  else if constexpr (I == 2)
    return p.get_age();
}
```

```
template <typename... Ns>
auto sum(Ns... ns);
```

## Before

```cpp
auto sum()
{
  return 0;
}

template <typename N>
auto sum(N n)
{
  return n;
}

template <typename N0, typename... Ns>
auto sum(N0 n0, Ns... ns)
{
  return n0 + sum(ns...);
}
```

## Before

```
auto sum()
{
  return 0;
}

template <typename N>
auto sum(N n)
{
  return n;
}

template <typename N0, typename... Ns>
auto sum(N0 n0, Ns... ns)
{
  return n0 + sum(ns...);
}
```

## C++17

```
template <typename... Ns>
auto sum(Ns... ns)
{
  return (ns + ... + 0);
}
```

| | | |
|---|---|---|
| Unary Right Fold | (E op ...) | $E_1$ op (... op ($E_{N-1}$ op $E_N$)) |
| Unary Left Fold | (... op E) | (($E_1$ op $E_2$) op ...) op $E_N$ |
| Binary Right Fold | (E op ... op I) | $E_1$ op (... op ($E_{N-1}$ op ($E_N$ op I))) |
| Binary Left Fold | (I op ... op E) | (((I op $E_1$) op $E_2$) op ...) op $E_N$ |

- Fold expressions apply binary operators to parameter packs.
- Parentheses around the fold expression are required.
- All binary operators are foldable:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| == | != | < | > | <= | >= | && | \|\| | , | . | -> | = |
| + | - | | / | % | ^ | & | \| | << | >> | | |
| += | -= | = | /= | %= | ^= | &= | \|= | <<= | >>= | | |

```
template <typename... Ns>
auto sum(Ns... ns) { return (ns + ... + 0); }

auto a = sum(3.14, 1e7, -42, 17);
      // 3.14 + (1e7 + (-42 + (17 + 0)))
```

```cpp
template <typename... Bs>
bool all(Bs... bs) { return (... && bs); }
```

```
template <typename... Bs>
bool all(Bs... bs) { return (... && bs); }

bool a = all(true, true, true, false);
      // ((true && true) && true) && false
```

```
template <typename... Bs>
bool all(Bs... bs) { return (... && bs); }

bool a = all();
      // ???
```

For unary folds, if the parameter pack is empty then the value of the fold is:

| | |
|---|---|
| `&&` | `true` |
| `\|\|` | `false` |
| `,` | `void()` |

For any operator not listed above, an unary fold expression with an empty parameter pack is ill-formed.

```
template <typename... Ts>
void print(Ts&&... ts)
{
  (std::cout << ... << std::forward<Ts>(ts)) << "\n";
}
```

```
template <typename F, typename... Args>
void for_each_arg(F f, Args&&... args)
{
  (f(std::forward<Args>(args)), ...);
}
```

## Before

```
std::tuple<int, double> t(42, 3.14);
auto t = std::make_tuple(42, 3.14);


return std::tuple<int, double>(42, 3.14);
return std::make tuple(42, 3.14);
```

## Before

```cpp
std::tuple<int, double> t(42, 3.14);
auto t = std::make_tuple(42, 3.14);


return std::tuple<int, double>(42, 3.14);
return std::make_tuple(42, 3.14);
```

## C++17

```cpp
std::tuple t(42, 3.14);



return std::tuple(42, 3.14);
```

**Before**

```
auto p = new std::tuple<int, int>{1, 1};
```

**C++17**

```
auto p = new std::tuple{0, 0};
```

## Before

```
auto = std::lock guard<std::mutex>(mtx);
```

## C++17

```
auto = std::lock guard(mtx);
```

- Class template parameters can now be deduced in:
  - Declarations.
  - Function-style cast expressions.
  - `new` expressions.
- Only performed if no template arguments are provided.
  - `std::tuple<int> t(0, 1)` is not allowed.

```
template_name (param0, ...) -> template_name<...>;
```

- User-defined deduction guides can be used to control how class template deduction operates.
- They do not have to be templates.
- They must be within the same scope (e.g. namespace, enclosing class) as the class template.

```cpp
namespace std
{

template <typename It>
vector(It b, It e) -> vector<typename std::iterator_traits<It>::value_type>;

}


std::vector f8({0, 1, 1, 2, 3, 5, 8, 13}); // Uses automatic deduction.

auto it = f8.begin();

std::vector f4(it, it + 4);                // Uses deduction guide.
```

```cpp
template <typename T>
struct name
{
  constexpr name(T first_, T last_);

  T first;
  T last;
};

name(char const*) -> name<std::string_view>;


name n{"John", "Smith"};     // name<std::string_view>


std::string first = // ...;
std::string last  = // ...;

name n{first, last};         // name<std::string>
```

## Before

```cpp
template <typename T, T v>
struct constant
{
  static constexpr T value = v;
};



using i = constant<int, 2048>;
using c = constant<char, 'a'>;
using b = constant<bool, true>;
using f = constant<decltype(F), F>;
```

## Before

```
template <typename T, T v>
struct constant
{
  static constexpr T value = v;
};


using i = constant<int, 2048>;
using c = constant<char, 'a'>;
using b = constant<bool, true>;
using f = constant<decltype(F), F>;
```

## C++17

```
template <auto v>
struct constant
{
  static constexpr auto value = v;
};


using i = constant<2048>;
using c = constant<'a'>;
using b = constant<true>;
using f = constant<F>;
```

```
template <auto parameter, ...>;
```

- Also known as `template <auto>`
- Uses regular `auto` deduction rules (`auto const`, `auto&`, `auto&&`, etc).

## Before

```
template <typename T, T... Elements>
struct sequence {};

using idxs = sequence<int, 0, 1, 2>;
using str  = sequence<char, 'h', 'i'>;
```

## C++17

```
template <auto... Elements>
struct sequence {};

using idxs = sequence<0, 1, 2>;
using str  = sequence<'h', 'i'>;
using tup  = sequence<0, 'h', true>;
```

## Before

```
template <std::size_t... Dims>
struct dimensions;


constexpr std::size_t dyn = -1;

dimensions<64, dyn, 32> d;
```

## C++17

```
template <auto... Dims>
struct dimensions;

struct dynamic_extent {};
constexpr dynamic_extent dyn = {};

dimensions<64, dyn, 32> d;
```

```cpp
template <auto... Dims>
struct dimensions;

struct dynamic_extent {};
inline constexpr dynamic_extent dyn = {};

dimensions<64, dyn, 32> d;
```

- Variables can now be `inline` just like functions.
- They may be defined in more than one translation unit as long as the definitions are identical.
- The definition must be present in a translation unit that accesses an `inline` variable.
- An `inline` variable with external linkage (e.g. not `static`):
  - Must be declared `inline` in every translation unit.
  - Has the same address in every translation unit.
- A `static constexpr` member variable is implicitly `inline`.

## Before

```
// In header (.hpp):

extern std::atomic<bool> ready;

// In source file (.cpp):
std::atomic<bool> ready = false;
```

## Before

```cpp
// In header (.hpp):

extern std::atomic<bool> ready;

// In source file (.cpp):
std::atomic<bool> ready = false;
```

## C++17

```cpp
inline std::atomic<bool> ready = false;
```

## Before

```cpp
// In header (.hpp):
struct system
{
  static std::atomic<bool> ready;
};

// In source file (.cpp):
std::atomic<bool> system::ready = false;
```

## C++17

```cpp
struct system
{
  inline static std::atomic<bool> ready = false;
};
```

## Before

```
auto add = [] (int n, int m)
{
  return n + m;
};

constexpr int i = add(5, 6);
// COMPILE ERROR: The lambda's
// call operator is not constexpr.
```

## Before

```
auto add = [] (int n, int m)
{
  return n + m;
};

constexpr int i = add(5, 6);
// COMPILE ERROR: The lambda's
// call operator is not constexpr.
```

## C++17

```
auto add = [] (int n, int m) constexpr
{
  return n + m;
};

constexpr int i = add(5, 6);
```

## Before

```
auto add = [] (int n, int m)
{
  return n + m;
};

constexpr int i = add(5, 6);
// COMPILE ERROR: The lambda's
// call operator is not constexpr.
```

## C++17

```
auto add = [] (int n, int m)
{
  return n + m;
};

constexpr int i = add(5, 6);
```

## Before

```
auto add = [] (int n, int m)
{
  return n + m;
};

constexpr int i = add(5, 6);
// COMPILE ERROR: The lambda's
// call operator is not constexpr.
```

## C++17

```
constexpr auto add = [] (int n, int m)
{
  return n + m;
};

constexpr int i = add(5, 6);
```

```cpp
template <typename... Xs>
constexpr auto make_storage(Xs... xs)
{
  auto storage = [=](auto f) { return f(xs...); };
  return storage;
}

template <typename... Xs>
struct tuple
{
  explicit constexpr tuple(Xs... xs)
    : storage{make_storage(xs...)} {}
  decltype(make_storage(declval<Xs>()...)) storage;
}

template <size_t N, typename... T>
constexpr decltype(auto) get(tuple<T...>& t)
{
  return t.storage([] (auto&&... xs) { /* ... */ }
}
```

## Before

```
template <typename T>
auto add(T x, T y)
{
  static_assert(is_addable_v<T>, "");
  return x + y;
}
```

## Before

```
template <typename T>
auto add(T x, T y)
{
  static_assert(is_addable_v<T>, "");
  return x + y;
}
```

## C++17

```
template <typename T>
auto add(T x, T y)
{
  static_assert(is_addable_v<T>);
  return x + y;
}
```

## Before

```cpp
auto
grab_lock(std::mutex& m)
{
  return std::lock_guard<std::mutex>(m);
  // COMPILE ERROR: Copy or move ctor
  // required; lock_guard has neither.
}


std::mutex mtx;

auto guard = grab lock(mtx);
```

## Before

```
auto
grab_lock(std::mutex& m)
{
  return std::lock_guard<std::mutex>(m);
  // COMPILE ERROR: Copy or move ctor
  // required; lock_guard has neither.
}


std::mutex mtx;

auto guard = grab lock(mtx);
```

## C++17

```
auto
grab_lock(std::mutex& m)
{
  return std::lock_guard(m);
}


std::mutex mtx;

auto guard = grab lock(mtx);
```

- Return Value Optimization (RVO) is now mandatory.
  - You can RVO `NonMoveable` types.
- Named Return Value Optimization (NRVO) and other forms of copy elision are not mandatory.

## Syntax

```
namespace A::B::C
{
  // ...
}
```

## Equivalent To

```
namespace A
{
  namespace B
  {
    namespace C
    {
      // ...
    }
  }
}
```

```
#if __has_include(<string_view>)
  #include <string_view>
  #define HAVE_STRING_VIEW 1
#elif __has_include(<experimental/string_view>)
  #include <experimental/string_view>
  #define HAVE_STRING_VIEW 1
  #define HAVE_EXP_STRING_VIEW 1
#else
  #define HAVE_STRING_VIEW 0
#endif
```

# Before

```
std::string first_3(std::string const& s)
{
  if (s.size() < 3) return s;
  return s.substr(0, 2); // Expensive copy.
                         // May allocate.
}


if (first_3("ABCDEFG") == "ABC")
  // ...
```

## Before

```
std::string first_3(std::string const& s)
{
  if (s.size() < 3) return s;
  return s.substr(0, 2); // Expensive copy.
                         // May allocate.
}


if (first_3("ABCDEFG") == "ABC")
  // ...
```

## C++17

```
std::string_view first_3(std::string_view s)
{
  if (s.size() < 3) return s;
  return s.substr(0, 3); // Cheap copy.
                         // Won't allocate.
}


if (first_3("ABCDEFG") == "ABC")
  // ...
```

# `std::string_view`

- `#include <string_view>`
- A non-owning view of a string.
- Interface is mostly the same as `std::string`; it is often a drop-in replacement.

| | `std::string` | `std::string_view` |
|---|---|---|
| **Heap Allocation** | Yes. | No. |
| **Ownership Semantics** | Owns its contents. | Non-owning (pointer + length). |
| **Copying** | Expensive. | Cheap. |
| **Passing Style** | By reference. | By value. |
| **Element Mutability** | Allowed. | Not allowed. |

## Before

```cpp
std::vector<std::string>
split(std::string const& s,
      std::regex const& r)
{
  using iterator = std::regex_token_iterator<
      std::string::const_iterator
  >;

  std::vector<std::string> v;

  std::transform(
    iterator(s.begin(), s.end(), r, -1),
    iterator(),
    std::back_inserter(v),
    [] (auto m)
    {
      return std::string(m.first, m.second);
    }
  );

  return v;
}
```

## C++17

```cpp
std::vector<std::string_view>
split(std::string_view s,
      std::regex const& r)
{
  using iterator = std::regex_token_iterator<
      std::string_view::const_iterator
  >;

  std::vector<std::string_view> v;

  std::transform(
    iterator(s.begin(), s.end(), r, -1),
    iterator(),
    std::back_inserter(v),
    [] (auto m)
    {
      return std::string_view(m.first,
                              m.length());
    }
  );

  return v;
}
```

## Before

```
int to_int(std::string const& s);

int to_int(char const* s);

int to_int(my_string const& s);
```

## C++17

```
int to_int(std::string_view s);


struct my_string
{
  // ...

  operator std::string_view() const
  {
    return std::string_view(data(), size());
  }
};
```

## Before

```
// Return default int on parse error.
int to_int(std::string_view s);

// Throw on parse error.
int to_int(std::string_view s);

// Return false on parse error.
int to_int(std::string_view s);

// Return null on parse error.
std::unique_ptr<int> to_int(std::string_view s);
```

## Before

```cpp
// Return default int on parse error.
int to_int(std::string_view s);

// Throw on parse error.
int to_int(std::string_view s);

// Return false on parse error.
int to_int(std::string_view s);

// Return null on parse error.
std::unique_ptr<int> to_int(std::string_view s);
```

## C++17

```cpp
std::optional<int> to_int(std::string_view s);
```

# `std::optional<T>`

- `#include <optional>`
- A nullable object wrapper. It adds a null state to the value it wraps.
- Interface is similar to smart pointers.

| | `std::optional<T>` |
|---|---|
| **Heap Allocation** | No. |
| **Ownership Semantics** | Owns its contents. |
| **Copying** | Same as `T` (cheap when empty). |
| **Passing Style** | Same as `T`. |

```cpp
std::optional<int> to_int(std::string_view s)
{
  std::optional<int> oi;
  int i;

  if (std::stringstream stm(s); (stm >> i))
    if (stm.get() == std::char_traits<char>::eof())
      oi = i;

  return oi;
}
```

# Before

```
struct person
{
  std::string first_name_;
  std::string middle_name_;
  std::string last_name_;

  bool is_middle_name_known() const
  {
    return !middle_name_.empty();
  }

  // Throws if the middle name is unknown.
  std::string_view get_middle_name() const
  {
    if (!is_middle_name_known())
      throw /* ... */
    return middle_name_;
  }
};
```

## Before

```
struct person
{
  std::string first_name_;
  std::string middle_name_;
  std::string last_name_;
  bool middle_name_known_;

  bool is_middle_name_known() const
  {
    return middle_name_known_;
  }

  // Throws if the middle name is unknown.
  std::string_view get_middle_name() const
  {
    if (!is_middle_name_known())
      throw /* ... */
    return middle_name_;
  }
};
```

## Before

```
struct person
{
  std::string first_name_;
  std::string middle_name_;
  std::string last_name_;
  bool middle_name_known_;

  bool is_middle_name_known() const
  {
    return middle_name_known_;
  }

  // Throws if the middle name is unknown.
  std::string_view get_middle_name() const
  {
    if (!is_middle_name_known())
      throw /* ... */
    return middle_name_;
  }
};
```

## C++17

```
struct person
{
  std::string first_name_;
  std::optional<std::string> middle_name_;
  std::string last_name_;


  bool is_middle_name_known() const
  {
    return middle_name_;
  }

  // Throws if the middle name is unknown.
  std::string_view get_middle_name() const
  {
    if (!is_middle_name_known())
      throw /* ... */
    return *middle_name_;
  }
};
```

```
auto slice(std::string_view str,
           std::optional<int> start,
           std::optional<int> end)
{
  auto s = start.value_or(0);
  auto e = end.value_or(str.size());
  return std.substr(s, e - s);
}
```

# Before

```
struct convert_result
{
  union data_type
  {

    int i;
    double d;
    std::string s;
  };

  enum kind_type
  {
    INT, DOUBLE, STRING
  };

  data_type data;
  kind_type kind;
};

convert_result
convert(std::string_view s);
```

# Before

```
struct convert_result
{
  union data_type
  {
    bool b;
    int i;
    double d;
    std::string s; // Do constructors and
  };               // destructors get run?

  enum kind_type
  {
    INT, DOUBLE, STRING
  }; // Forgot to update this!

  data_type data;
  kind_type kind;
};

convert_result
convert(std::string_view s);
```

## Before

```cpp
struct convert_result
{
  union data_type
  {
    bool b;
    int i;
    double d;
    std::string s; // Do constructors and
  };               // destructors get run?

  enum kind_type
  {
    BOOL, INT, DOUBLE, STRING
  };

  data_type data;
  kind_type kind;
};

convert_result
convert(std::string view s);
```

## C++17

```cpp
std::variant<bool, int, double, std::string>
convert(std::string view s);
```

# `std::variant<T0, T1, ...>`

- `#include <variant>`
- A discriminated union.
- Interface is similar to Boost.Variant.
- Access uses the visitor pattern.

| | `std::variant<T0, T1, ...>` |
|---|---|
| **Heap Allocation** | No. |
| **Ownership Semantics** | Owns its contents. |
| **Copying** | Depends on `T0`, `T1`, ... |
| **Passing Style** | Depends on `T0`, `T1`, ... |

```cpp
template <unsigned N>
std::string repeat(std::string_view s)
{
    std::string tmp;
    for (unsigned i = 0; i != N; ++i)
      tmp += s;
    return tmp;
}

template <unsigned N>
struct multiplier_visitor
{
  void operator()(std::string& t) const        { t = repeat(t); }
  void operator()(int& t) const                { t = t * N; }
  void operator()(std::array<int, 2>& t) const { t = {{t[0] * N, t[1] * N}}; }
};



std::variant<std::string, int, std::array<int, 2>> v;
// Default state is the first type, e.g. std::string.

v = 21;
std::visit(multiplier_visitor<2>{}, v); // v == 42.

v = "Ha";
std::visit(multiplier visitor<3>{}, v); // v == "HaHaHa".
```

```
template <unsigned N>
std::string repeat(std::string_view s);


std::variant<std::string, int, std::array<int, 2>> v = // ...

std::visit(
  [](auto& t)
  {
    constexpr unsigned N = 10;

    using T = std::decay_t<decltype(t)>;

    if constexpr      (std::is_same_v<T, std::string>)
      t = repeat(t);
    else if constexpr (std::is_same_v<T, int>)
      t = t * N;
    else if constexpr (std::is_same_v<T, std::array<int, 2>>)
      t = {{t[0] * N, t[1] * N}};
    else
      static_assert(false);
  }, v);
```

```
template <unsigned N>
std::string repeat(std::string_view s);

template <typename... Ts>
struct overloaded : Ts...
{
    using Ts::operator()...;
};

template <typename... Ts>
overloaded(Ts...) -> overloaded<Ts...>;



std::variant<std::string, int, std::array<int, 2>> v = // ...

constexpr unsigned N = 10;

std::visit(
  overloaded{
    [=](std::string& t)        { t = repeat(t); },
    [=](int& t)                { t = t * N; },
    [=](std::array<int, 2>& t) { t = {{t[0] * N, t[1] * N}}; }
  }, v);
```

```
template <typename Leaf>
struct binary_tree;

template <typename Leaf>
struct binary_tree_branch
{
  std::shared_ptr<binary_tree<leaf> > left;
  std::shared_ptr<binary_tree<leaf> > right;
};

template <typename Leaf>
struct binary_tree
{
  std::variant<Leaf, binary_tree_branch<leaf> > value;
};
```

```
std::vector<std::any> v;

v.push_back("hello world");
v.push_back(std::tuple(3.14, true));
v.push_back(42);
v.push_back(std::vector<std::any>{});
```

# `std::any`

- `#include <any>`
- Type-erasure for copyable objects.
- Four main operations:
  - Copy it.
  - Assign a value of some type T (`operator=`).
  - Ask whether it contains a value of some type T (`.type`).
  - Retrieve a value of some type T (`any_cast<>`).

| | `std::any` |
|---|---|
| **Heap Allocation** | Yes. |
| **Ownership Semantics** | Owns its contents. |
| **Cost of Copying** | Same as the contained type. |
| **Passing Style** | By reference. |

## Before

```
std::vector<T> x = // ...

#pragma omp parallel for simd
for (std::size_t i = 0; i < x.size(); ++i)
  process(x[i]);
```

## Before

```
std::vector<T> x = // ...

#pragma omp parallel for simd
for (std::size_t i = 0; i < x.size(); ++i)
  process(x[i]);
```

## C++17

```
std::vector<T> x = // ...

std::for_each(std::par_unseq,
              x.begin(), x.end(), process);
```

## Serial

```
std::vector<T> x = // ...

std::sort(x.begin(), x.end());
```

**Serial**

```
std::vector<T> x = // ...

std::sort(x.begin(), x.end());
```

**Parallel**

```
std::vector<T> x = // ...

std::sort(std::par, x.begin(), x.end());
```

```
template <typename ExecutionPolicy, ...>
auto algorithm(ExecutionPolicy&& exec, ...);
```

- `#include <algorithm>`, `<numeric>` and `<execution>`.
- New parallel (e.g. `ExecutionPolicy`) overloads for most of the existing algorithms.
  - `InputIterator` requirements strengthened to `ForwardIterator`.
  - Complexity guarantees relaxed for some algorithms.
- New algorithms designed for parallel programming.
  - `reduce`, `inclusive_scan` and `exclusive_scan`.
  - `transform_reduce`, `transform_inclusive_scan` and `transform_exclusive_scan`.

```
template <typename ExecutionPolicy, ...>
auto algorithm(ExecutionPolicy&& exec, ...);
```

- `ExecutionPolicy`: Describes the "how" of execution.
  - Is parallelism allowed?
  - What restrictions must be respected by the algorithm?
- `Executor` (planned C++20): Describes the "where" of execution.
  - On which abstract resource (thread pool, current thread, GPU, etc) should work be executed?
  - How should that work be queued?
- Execution policies:
  - `std::seq`: Serial.
  - `std::par`: Parallel (SIMT, ex: NVIDIA GPUs).
  - `std::par_unseq`: Parallel and unordered (SIMT and SIMD, ex: NVIDIA GPUs and Intel CPUs).

## Serial

```
std::vector<double> x = // ...
std::vector<double> y = // ...

double dot_product =
  (x[0] * y[0]) + (x[1] * y[1]) + // ...
```

## Serial

```
std::vector<double> x = // ...
std::vector<double> y = // ...

double dot_product =
  (x[0] * y[0]) + (x[1] * y[1]) + // ...
```

## Parallel

```
std::vector<double> x = // ...
std::vector<double> y = // ...

double dot_product = std::transform_reduce(
  std::par_unseq, x.begin(), x.end(), y.begin()
);
```

## Serial

```
std::vector<double> x = // ...
std::vector<double> y = // ...

double dot_product = std::transform_reduce(
  x.begin(), x.end(), y.begin()
);
```

## Parallel

```
std::vector<double> x = // ...
std::vector<double> y = // ...

double dot_product = std::transform_reduce(
  std::par_unseq, x.begin(), x.end(), y.begin()
);
```

# Serial

```
std::vector<double> x = // ...

double norm =
  std::sqrt(
    (x[0] * x[0]) + (x[1] * x[1]) + /* ... */
  );
```

## Serial

```
std::vector<double> x = // ...

double norm =
  std::sqrt(
    (x[0] * x[0]) + (x[1] * x[1]) + /* ... */
  );
```

## Parallel

```
std::vector<double> x = // ...

double norm =
  std::sqrt(
    std::transform_reduce(
      std::par_unseq,

      // Input sequence.
      x.begin(), x.end(),

      // Initial reduction value.
      double(0.0),

      // Binary reduction op: Addition.
      [] (double xl, double xr) { return xl + xr; }

      // Unary transform op: Squaring.
      [] (double x) { return x * x; }
    )
  );
```

# Before (Windows)

```
std::wstring dir = L"\\sandbox";
std::wstring p = dir + L"\\foobar.txt";
std::wstring copy = p;
copy += ".bak";
CopyFile(p, copy, false);
```

```
std::string dir_copy = dir + ".bak";
SHFILEOPSTRUCT s = { 0 };
s.hwnd = someHwndFromSomewhere;
s.wFunc = FO_COPY;
s.fFlags = FOF_SILENT;
s.pFrom = dir.c_str();
s.pTo = dir_copy.c_str();
SHFileOperation(&s);
```

## Before (Windows)

```
std::wstring dir = L"\\sandbox";
std::wstring p = dir + L"\\foobar.txt";
std::wstring copy = p;
copy += ".bak";
CopyFile(p, copy, false);
```

```
std::string dir_copy = dir + ".bak";
SHFILEOPSTRUCT s = { 0 };
s.hwnd = someHwndFromSomewhere;
s.wFunc = FO_COPY;
s.fFlags = FOF_SILENT;
s.pFrom = dir.c_str();
s.pTo = dir_copy.c_str();
SHFileOperation(&s);
```

## C++17

```
fs::path dir = "/";
dir /= "sandbox";
fs::path p = dir / "foobar.txt";
fs::path copy = p;
copy += ".bak";
fs::copy(p, copy);
```

```
fs::path dir_copy = dir;
dir_copy += ".bak";
```

```
fs::copy(dir, dir_copy,
         fs::copy_options::recursive);
```

- `#include <filesystem>`
- Usual convention: `namespace fs = std::filesystem;`
- Based on Boost.Filesystem.
- Interface is primarily non-member functions that operator on path objects.
  - `fs::path`
  - `fs::directory_entry` and `fs::directory_iterator`
  - `fs::file_status` - file type and permissions

Four major capabilities:

- Path creation/manipulation.
- Directory iteration and recursion.
- File/directory metadata query.
- File/directory creation, removal and modification.

## Before (POSIX)

```
void display_contents(std::string const & p)
{
  std::cout << p << "\n";

  struct dirent *dp;
  DIR *dfd;

  if ((dfd = opendir(p.c_str()) == nullptr)
    return;

  while ((dp = readdir(dfd)) != nullptr)
  {
    struct stat st;
    string filename = p + "/" + dp->d_Name;
    if (stat(filename.c_str(), &st) == -1)
      continue;

    if ((st.st_mode & S_IFMT) == S_IFDIR)
      std::cout << "  " << filename << "\n";
    else
      std::cout << "  " << filename
                << " [" << st.st_size
                << " bytes]\n";
  }
}
```

## C++17

```
void display_contents(fs::path const & p)
{
  std::cout << p.filename() << "\n";

  if (!fs::is_directory(p))
    return;

  for (auto const & e: fs::directory_iterator{p})
  {
    if (fs::is_regular_file(e.status()))
      std::cout << "  " << e.path().filename()
                << " [" << fs::file_size(e) << " b
    else if (fs::is_directory(e.status()))
      std::cout << "  " << e.path().filename() <<
  }
}
```

## Before

```
std::vector<int, std::allocator<int>> a;

std::vector<int, my_memory_pool<int>> b;

std::vector<int, my_slab_allocator<int, 1024>> c;

std::vector<int, my slab allocator<int, 4096>> d;
```

## Before

```cpp
std::vector<int, std::allocator<int>> a;

std::vector<int, my_memory_pool<int>> b;

std::vector<int, my_slab_allocator<int, 1024>> c;

std::vector<int, my_slab_allocator<int, 4096>> d;
```

## C++17

```cpp
std::vector<int, std::pmr::allocator<int>> a;

std::pmr::unsynchronized_pool_resource p;
std::vector<int, std::pmr::allocator<int>> b(&p);

my_slab_allocator<1024> s1k;
std::vector<int, std::pmr::allocator<int>> c(&s1k);

my_slab_allocator<4096> s4k;
std::vector<int, std::pmr::allocator<int>> d(&s4k);
```

## Before

```cpp
std::vector<int, std::allocator<int>> a;

std::vector<int, my_memory_pool<int>> b;

std::vector<int, my_slab_allocator<int, 1024>> c;

std::vector<int, my slab allocator<int, 4096>> d;
```

## C++17

```cpp
std::pmr::vector<int> a;

std::pmr::unsynchronized_pool_resource p;
std::pmr::vector<int> b(&p);

my_slab_allocator<1024> s1k;
std::pmr::vector<int> c(&s1k);

my_slab_allocator<4096> s4k;
std::pmr::vector<int> d(&s4k);
```

- `#include <memory_resource>`
- `std::pmr::polymorphic_allocator` is a type-erasing allocator wrapping an object that implements the `std::pmr::memory_resource` interface.
- `std::pmr` allocator-aware STL aliases.
- Standard `std::pmr::memory_resources`:
    - `std::pmr::new_delete_resource` - global `new`/`delete`.
    - `std::pmr::unsynchronized_pool_resource` - thread-unsafe pool.
    - `std::pmr::synchronized_pool_resource` - thread-safe pool.
    - `std::pmr::monotonic_buffer_resource` - memory is only released when the resource goes out of scope.

# Before

```
template <typename T, typename... Args>
T* new_aligned_array(std::size_t size,
                     std::size_t alignment)
{
  void* vp = nullptr;
  int r = posix_memalign(&vp, alignment,
                         size * sizeof(T));
  return new (vp) T[] ();
  // T must be DefaultConstructible.
}


struct alignas(128) person
{
  // ...
};


person* p = new_aligned_array(
  1024, alignof(person)
);
```

## Before

```cpp
template <typename T, typename... Args>
T* new_aligned_array(std::size_t size,
                     std::size_t alignment)
{
  void* vp = nullptr;
  int r = posix_memalign(&vp, alignment,
                         size * sizeof(T));
  return new (vp) T[] ();
  // T must be DefaultConstructible.
}


struct alignas(128) person
{
  // ...
};


person* p = new_aligned_array(
  1024, alignof(person)
);
```

## C++17

```cpp
struct alignas(128) person
{
  // ...
};


person* p = new person[1024];
// Calls operator new[](
//   sizeof(person),
//   std::align_val_t(alignof(person))
// )
```

```cpp
void* operator new(std::size_t, std::align_val_t);
void* operator new[](std::size_t, std::align_val_t);
void operator delete(void*, std::align_val_t);
void operator delete[](void*, std::align_val_t);
void operator delete(void*, std::align_val_t, std::size_t);
void operator delete[](void*, std::align_val_t, std::size_t);
```

- `#include <aligned_new>`
- Alignment-aware global `new`.
- For allocations requiring an alignment that is not guaranteed by alignment-unaware global `new`, the lookup order is:
  - Class-specific and alignment-aware `new`.
  - Class-specific and alignment-unaware `new`.
  - Global and alignment-aware `new`.

## Before

```cpp
std::map<int, person> m = // ...

person p = // ...
auto res = m.emplace(42, std::move(p));
// If the insertion failed, was p moved?

person p = // ...
m[42] = std:::move(p);
// Value type must be DefaultConstructible.
// Did we insert or assign?
// What iterator is the key at?
```

## C++17

```cpp
std::map<int, person> m = // ...

person p = // ...
auto res = m.try_emplace(42, std::move(p));
// If the insertion failed, p wasn't moved.

person p = // ...
auto res = m.insert_or_assign(std:::move(p));
// Returns info as a pair<iterator, bool>.
```

# Before

```
std::map<int, person> m0 = // ...
std::map<int, person> m1 = // ...

m0.merge(m1);
```

```
template <typename Key, typename Value>
void move_and_rekey(std::map<Key, Value>& src,
                    Key const& srckey,
                    std::map<Key, Value>& dest
                    Key const& destkey)
{
  auto node = src.extract(srckey);
  node.key() = destkey;
  dest.insert(std::move(node));
}
```

| | | |
|---|---|---|
| `std::beta` | `std::assoc_legendre` | `std::sph_neumann` |
| `std::expint` | `std::sph_legendre` | `std::ellint_1` |
| `std::riemann_zeta` | `std::cyl_bessel_i` | `std::comp_ellint_1` |
| `std::hermite` | `std::cyl_bessel_j` | `std::ellint_2` |
| `std::laguerre` | `std::cyl_bessel_k` | `std::comp_ellint_2` |
| `std::assoc_laguerre` | `std::sph_bessel` | `std::ellint_3` |
| `std::legendre` | `std::cyl_neumman` | `std::comp_ellint_3` |

- `#include <cmath>`
- A collection of common mathematical functions know as special functions.
- Based on Boost.Math and the Fortran netlib library.
- Previously part of ISO/IEC 29124:2010.
- `*f` and `*l` forms to control the return type.

## Before

```cpp
template <typename T>
std::enable_if_t<
  std::is_integral<T>::value, T
>
sqrt(T t);

template <typename T>
std::enable_if_t<
  std::is_floating_point<T>::value, T
>
sqrt(T t);
```

## C++17

```cpp
template <typename T>
std::enable_if_t<
  std::is_integral_v<T>, T
>
sqrt(T t);

template <typename T>
std::enable_if_t<
  std::is_floating_point_v<T>, T
>
sqrt(T t);
```

```
template <...>
bool constexpr trait_v = trait<...>::value;
```

- `#include <type_traits>`
- Variable templates for metafunctions.
- Defined for all the type traits: `is_*`, etc.

# Before

```
template <typename... Ts>
struct all_integral;

template <>
struct all_integral<> : std::true_type {};

template <typename T0, typename... Ts>
struct all_integral<T0, Ts...>
  : std::integral_constant<
      bool,
      std::is_integral<T0>::value &&
      all_integral<Ts...>::value
    > {};
```

## Before

```
template <typename... Ts>
struct all_integral;

template <>
struct all_integral<> : std::true_type {};

template <typename T0, typename... Ts>
struct all_integral<T0, Ts...>
  : std::integral_constant<
      bool,
      std::is_integral<T0>::value &&
      all_integral<Ts...>::value
    > {};
```

## C++17

```
template <typename... Ts>
struct all_integral
  : std::bool_constant<
      (... && std::is_integral_v<Ts>)
    > {};
```

## Before

```
template <typename... Ts>
struct all_integral;

template <>
struct all_integral<> : std::true_type {};

template <typename T0, typename... Ts>
struct all_integral<T0, Ts...>
  : std::integral_constant<
      bool,
      std::is_integral<T0>::value &&
      all_integral<Ts...>::value
    > {};
```

## C++17

```
template <typename... Ts>
struct all_integral
  : std::conjunction<
      std::is_integral<Ts>...
    > {};
```

| | |
|---|---|
| `std::bool_constant<B>` | `std::integral_constant<bool, B>` |
| `std::conjunction<Ts...>` | `std::bool_constant<(... && Ts::value)>` |
| `std::disjunction<Ts...>` | `std::bool_constant<(... \|\| Ts::value)>` |
| `std::negation<T>` | `std::bool_constant<!T::value>` |

- `#include <type_traits>`
- Boolean logic metafunctions.
- Lazily evaluated.
    - E.g. `std::conjunction` and `std::disjunction` are short-circuited.
- `*_t` and `*_v` aliases are also defined.

## Language Changes

- Structured bindings
- Selection statements with initializers
- Compile-time conditional statments
- Fold expressions
- Class template deduction
- `auto` non-type template parameters
- `inline` variables
- `constexpr` lambdas
- Unary `static_assert`
- Guaranteed copy elision
- Nested namespace definitions
- Preprocessor predicate for header testing
- ...

## Library Changes

- `string_view`
- `optional`
- `variant`
- `any`
- Parallel algorithms
- Filesystem support
- Polymorphic allocators and memory resources
- Aligned `new`
- Improved insertion and splicing for associative containers
- Math special functions
- Variable templates for metafunctions
- Boolean logic metafunctions
- ...