

All C++20 core language features with examples

Apr 2, 2021

Introduction

The story behind this article is very simple, I wanted to learn about new C++20 language features and to have a brief summary for all of them on a single page. So, I decided to read all proposals and create this “cheat sheet” that explains and demonstrates each feature. This is not a “best practices” kind of article, it serves only demonstrational purpose. Most examples were inspired or directly taken from corresponding proposals, all credit goes to their authors and to members of ISO C++ committee for their work. Enjoy!

Table of contents

- [Concepts](#)
- [Modules](#)
- [Coroutines](#)
- [Three-way comparison](#)
- [Lambda expressions](#)
 - [Allow lambda-capture](#) `[=, this]`
 - [Template parameter list for generic lambdas](#)
 - [Lambdas in unevaluated contexts](#)
 - [Default constructible and assignable stateless lambdas](#)
 - [Pack expansion in lambda init-capture](#)
- [Constant expressions](#)
 - [Immediate functions](#) `constexpr`
 - `constexpr` virtual function
 - `constexpr` try-catch blocks
 - `constexpr` `dynamic_cast` and polymorphic `typeid`
 - [Changing the active member of a union inside constexpr](#)
 - `constexpr` allocations
 - [Trivial default initialization in constexpr functions](#)
 - [Unevaluated asm-declaration in constexpr functions](#)
 - `std::is_constant_evaluated()`
- [Aggregates](#)

- Prohibit aggregates with user-declared constructors
- Class template argument deduction for aggregates
- Parenthesized initialization of aggregates
- Non-type template parameters
 - Class types in non-type template parameters
 - Generalized non-type template parameters
- Structured bindings
 - Lambda capture and storage class specifiers for structured bindings
 - Relaxing the structured bindings customization point finding rules
 - Allow structured bindings to accessible members
- Range-based `for` loop
 - init-statements for range-based `for` loop
 - Relaxing the range-based `for` loop customization point finding rules
- Attributes
 - `[[likely]]` and `[[unlikely]]`
 - `[[no_unique_address]]`
 - `[[nodiscard]]` with message
 - `[[nodiscard]]` for constructors
- Character encoding
 - `char8_t`
 - Stronger Unicode requirements
- Sugar
 - Designated initializers
 - Default member initializers for bit-fields
 - More optional `typename`
 - Nested `inline` namespaces
 - `using enum`
 - Array size deduction in new-expressions
 - Class template argument deduction for alias templates
- `constexpr`
- Signed integers are two's complement
- `__VA_OPT__` for variadic macros
- Explicitly defaulted functions with different exception specifications
- Destroying `operator delete`
- Conditionally `explicit` constructors
- Feature-test macros
- Known-to-unknown bound array conversions
- Implicit move for more local objects and rvalue references
- Conversion from `T*` to `bool` is narrowing
- Deprecate some uses of `volatile`
- Deprecate comma operator in subscripts
- Fixes
 - Initializer list constructors in class template argument deduction

- `const&` -qualified pointers to members
- Simplifying implicit lambda capture
- `const` mismatch with defaulted copy constructor
- Access checking on specializations
- ADL and function templates that are not visible
- Specify when `constexpr` function definitions are needed for constant evaluation
- Implicit creation of objects for low-level object manipulation

Concepts

The basic idea behind concepts is to specify what's needed from a template argument so the compiler can check it before instantiation. As a result, the error message, if any, is much cleaner, something like `constraint X was not satisfied`. Before C++20 it was possible to use tricky `enable_if` constructions or just fail during template instantiation with cryptic error messages. With concepts failure happens early and the error message is much cleaner.

Requires expression

Let's start with `requires-expression`. It's an expression that contains actual requirements for template arguments, it evaluates to `true` if they are satisfied and `false` otherwise.

```

1  template<typename T> /*...*/
2  requires (T x) // optional set of fictional parameter(s)
3  {
4      // simple requirement: expression must be valid
5      x++;      // expression must be valid
6
7      // type requirement: `typename T`, T type must be a valid type
8      typename T::value_type;
9      typename S<T>;
10
11     // compound requirement: {expression}[noexcept][-> Concept];
12     // {expression} -> Concept<A1, A2, ...> is equivalent to
13     // requires Concept<decltype((expression)), A1, A2, ...>
14     {*x}; // dereference must be valid
15     {*x} noexcept; // dereference must be noexcept
16     // dereference must return T::value_type
17     {*x} noexcept -> std::same_as<typename T::value_type>;
18
19     // nested requirement: requires ConceptName<...>;
20     requires Addable<T>; // constraint Addable<T> must be satisfied
21 };

```

Concept

Concept is simply a named set of such constraints or their logical combination. Both concept and requires-expression render to a compile-time bool value and can be used as a normal value, for example in `if constexpr`.

```
1  template<typename T>
2  concept Addable = requires(T a, T b)
3  {
4      a + b;
5  };
6
7  template<typename T>
8  concept Dividable = requires(T a, T b)
9  {
10     a/b;
11 };
12
13 template<typename T>
14 concept DivAddable = Addable<T> && Dividable<T>;
15
16 template<typename T>
17 void f(T x)
18 {
19     if constexpr(Addable<T>){ /*...*/ }
20     else if constexpr(requires(T a, T b) { a + b; }){ /*...*/ }
21 }
```

Requires clause

To actually constrain something we need `requires-clause`. It may appear right after `template<>` block or as the last element of a function declaration, or even at both places at once, lambdas included:

```
1  template<typename T>
2  requires Addable<T>
3  auto f1(T a, T b) requires Subtractable<T>; // Addable<T> && Subtractable<
4
5  auto l = [<typename T> requires Addable<T>
6      (T a, T b) requires Subtractable<T>{}];
7
8  template<typename T>
```

```

9   requires Addable<T>
10  class C;
11
12  // infamous `requires requires`. First `requires` is requires-clause,
13  // second one is requires-expression. Useful if you don't want to introduce
14  // concept.
15  template<typename T>
16  requires requires(T a, T b) {a + b;}
17  auto f4(T x);

```

Much cleaner way is to use concept name instead of `class/typename` keyword in template parameter list:

```

1   template<Addable T>
2   void f();

```

Template template parameters can also be constrained. In this case argument must be less or equally constrained than parameter. Unconstrained template template parameters still can accept constrained templates as arguments:

```

1   template<typename T>
2   concept Integral = std::integral<T>;
3
4   template<typename T>
5   concept Integral4 = std::integral<T> && sizeof(T) == 4;
6
7   // requires-clause also works here
8   template<template<typename T1> requires Integral<T1> typename T>
9   void f2(){}
10
11  // f() and f2() forms are equal
12  template<template<Integral T1> typename T>
13  void f(){
14      f2<T>();
15  }
16
17  // unconstrained template template parameter can accept constrained argument
18  template<template<typename T1> typename T>
19  void f3(){}
20
21  template<typename T>

```

```

22 struct S1{};
23
24 template<Integral T>
25 struct S2{};
26
27 template<Integral4 T>
28 struct S3{};
29
30 void test(){
31     f<S1>();    // OK
32     f<S2>();    // OK
33     // error, S3 is constrained by Integral4 which is more constrained than
34     // f()'s Integral
35     f<S3>();
36
37     // all are OK
38     f3<S1>();
39     f3<S2>();
40     f3<S3>();
41 }

```

Functions with unsatisfied constraints become “invisible”:

```

1  template<typename T>
2  struct X{
3      void f() requires std::integral<T>
4      {}
5  };
6
7  void f(){
8      X<double> x;
9      x.f(); // error
10     auto pf = &X<double>::f; // error
11 }

```

Constrained `auto`

`auto` parameters now allowed for normal functions to make them generic just like generic lambdas. Concepts can be used to constrain placeholder types(`auto` / `decltype(auto)`) in various contexts. For parameter packs, `MyConcept... Ts` requires `MyConcept` to be true for each element of the pack, not for the whole pack at once, e.g. `requires<T1> && requires<T2> && ... && requires<TLast>`.

```

1  template<typename T>
2  concept is_sortable = true;
3
4  auto l = [](auto x){};
5  void f1(auto x){}           // unconstrained template
6  void f2(is_sortable auto x){} // constrained template
7
8  template<is_sortable auto NonTypeParameter, is_sortable TypeParameter>
9  is_sortable auto f3(is_sortable auto x, auto y)
10 {
11     // notice that nothing is allowed between constraint name and `auto`
12     is_sortable auto z = 0;
13     return 0;
14 }
15
16 template<is_sortable auto... NonTypePack, is_sortable... TypePack>
17 void f4(TypePack... args){}
18
19 int f();
20
21 // takes two parameters
22 template<typename T1, typename T2>
23 concept C = true;
24 // binds second parameter
25 C<double> auto v = f(); // means C<int, double>
26
27 struct X{
28     operator is_sortable auto() {
29         return 0;
30     }
31 };
32
33 auto f5() -> is_sortable decltype(auto){
34     f4<1,2,3>(1,2,3);
35     return new is_sortable auto(1);
36 }

```

Partial ordering by constraints

This section was inspired by the article [Ordering by constraints](#) by Andrzej Krzemiński. Check it out for a more thorough explanation.

Aside from specifying requirements for a single declaration, constraints can be used to select the best alternative for a normal function, template function or a class template. To do so, constraints have a notion of partial ordering, that is, one constraint can be *at least* or *more* constrained than the other or they can be *unordered*(unrelated). Compiler decomposes(the Standard uses term *normalization* but for me *decomposition* sounds better) constraint into a conjunction/ disjunction of *atomic* constraints. Intuitively, `C1 && C2` is more constrained than `C1`, `C1` is more constrained than `C1 || C2` and any constraint is more constrained than the unconstrained declaration. When more than one candidate with satisfied constraints are present, the most constrained one is chosen. If constraints are unordered, the usage is ambiguous.

```
1  template<typename T>
2  concept integral_or_floating = std::integral<T> || std::floating_point<T>;
3
4  template<typename T>
5  concept integral_and_char = std::integral<T> && std::same_as<T, char>;
6
7  void f(std::integral auto){}          // #1
8  void f(integral_or_floating auto){}  // #2
9  void f(std::same_as<char> auto){}    // #3
10
11 // calls #1 because std::integral is more constrained
12 // than integral_or_floating(#2)
13 f(int{});
14 // calls #2 because it's the only one whose constraint is satisfied
15 f(double{});
16 // error, #1, #2 and #3's constraints are satisfied but unordered
17 // because std::same_as<char> appears only in #3
18 f(char{});
19
20 void f(integral_and_char auto){}      // #4
21
22 // calls #4 because integral_and_char is more
23 // constrained than std::same_as<char>(#3) and std::integral(#1)
24 f(char{});
```

It's important to understand how the compiler decomposes constraints and when it can see that they have common atomic constraint and deduce order between them. During decomposition, the concept name is replaced with its definition but `requires-expression` is *not* further decomposed. Two atomic constraints are identical only if they are represented by the same expression at the same location. For example, `concept C = C1 && C2` is decomposed to conjunction of `C1` and `C2` but `concept C = requires{...}` becomes

concept `C = Expression-Location-Pair` and its body is not further decomposed. If two concepts have common or even the same requirements in their `requires-expression`, they will always be unordered because either their `requires-expression`s are not equal or they are equal but at different source locations. The same happens with duplicated usage of a naked type traits - they always represent different atomic constraints because of different locations, thus, cannot be used for ordering.

```
1  template<typename T>
2  requires std::is_integral_v<T> // uses type traits instead of concepts
3  void f1(){} // #1
4
5  template<typename T>
6  requires std::is_integral_v<T> || std::is_floating_point_v<T>
7  void f1(){} // #2
8
9  // error, #1 and #2 have common `std::is_integral_v<T>` expression
10 // but at different locations(line 2 vs. line 6), thus, #1 and #2 constraints
11 // are unordered and the call is ambiguous
12 f1(int{});
13
14 template<typename T>
15 concept C1 = requires{ // requires-expression is not decomposed
16     requires std::integral<T>;
17 };
18
19 template<typename T>
20 concept C2 = requires{ // requires-expression is not decomposed
21     requires (std::integral<T> || std::floating_point<T>);
22 };
23
24 void f2(C1 auto){} // #3
25 void f2(C2 auto){} // #4
26
27 // error, since requires-expressions are not decomposed, #3 and #4 have
28 // completely unrelated and hence unordered constraints and the call is
29 // ambiguous
30 f2(int{});
```

Conditionally trivial special member functions

For wrapper types like `std::optional` or `std::variant` it's useful to propagate *triviality* from the types they wrap. For example, `std::optional<int>` should be trivial but

`std::optional<std::string>` shouldn't. In C++17 this can be achieved using [pretty cumbersome machinery](#). Concepts provide a natural solution for this: we can create multiple versions of the same special member function with different constraints, the compiler will choose the best one and ignore the others. In this particular case, we need a trivial set of functions when the wrapped type is a trivial and a non-trivial set of functions when it's not. For this to work, some updates have been made to the definition of trivial type. In C++17, a trivially copyable class is required to have *all* of its copy and move operations either deleted or trivial. To take concepts into account, the notion of an *eligible special member function* was introduced. It is a function that's not deleted, whose constraints(if any) are satisfied and no other special member function of the same kind, with the same first parameter type(if any), is more constrained. Simply put, it's a function(s) with the most constrained satisfied constraints(if any). All existing destructors(yes, now you can have more than one) are now called *prospective* destructors. Only one "active" destructor is allowed, it's selected using normal overload resolution.

A *trivially copyable* class is now a class that has a *trivial* non-deleted destructor, *at least one* eligible copy/move operation and whose all such eligible operations are trivial. A *trivial* class is a trivially copyable class that has one or more eligible default constructors, all of which are trivial.

Here's the skeleton of this technique:

```
1  template<typename T>
2  class optional{
3  public:
4      optional() = default;
5
6      // trivial copy-constructor
7      optional(const optional&) = default;
8
9      // non-trivial copy-constructor
10     optional(const optional& rhs)
11         requires(!std::is_trivially_copy_constructible_v<T>){
12         // ...
13     }
14
15     // trivial destructor
16     ~optional() = default;
17
18     // non-trivial destructor
19     ~optional() requires(!std::is_trivial_v<T>){
20     // ...
21     }
22     // ...
23 private:
```

```
24     T value;
25 };
26
27 static_assert(std::is_trivial_v<optional<int>>);
28 static_assert(!std::is_trivial_v<optional<std::string>>);
```

Modules

Modules is a new way to organize C++ code into logical components. Historically, C++ used C model which is based on the preprocessor and repetitive textual inclusion. It has a lot of problems such as macros leakage in and out from headers, inclusion-order-dependent headers, repetitive compilation of the same code, cyclic dependencies, poor encapsulation of implementation details and so on. Modules are about to solve them but not so fast. We won't be able to use their full power until compilers *and* build tools, such as CMake, will support it too. Full description of Modules is well beyond the scope of this article, I will only show the basic ideas and use cases. For more details you can read [a series of articles by vector-of-bool](#) or just google for other blog posts or talks.

The main idea behind modules is to restrict what's accessible(`export` ed) when a module is used(`import` ed) by its clients. This allows true hiding of implementation details.

```
1 // module.cpp
2 // dots in module name are for readability purpose, they have no special m
3 export module my.tool; // module declaration
4
5 export void f(){}      // export f()
6 void g(){}            // but not g()
7
8 // client.cpp
9 import my.tool;
10
11 f();    // OK
12 g();    // error, not exported
```

Modules are macro-unfriendly, you can't pass manually `#define` d macros to module(compiler's built-in and command-line macros are still visible) and only in one special case you can import macros from module. Modules can't have cyclic dependencies. Module is a self-contained entity, compiler can precompile each module exactly once so overall compilation time is greatly improved. Import order doesn't matter for modules.

Module units

A module can be either *interface* or *implementation* module unit. Only interface units can contribute to the module's interface, that's why they have `export` in their declaration. A module can be a single file or scattered across *partitions*. Each partition is named in the form `module_name:partition_name`. Partitions are `import` able only within the same module and client can `import` only a module as a whole. This provides much better encapsulation than header files.

```
1 // tool.cpp
2 export module tool; // primary module interface unit
3 export import :helpers; // re-export(see below) helpers partition
4
5 export void f();
6 export void g();
7
8 // tool.internals.cpp
9 module tool:internals; // implementation partition
10 void utility();
11
12 // tool.impl.cpp
13 module tool; // implementation unit, implicitly imports primary module
14 import :internals;
15
16 void utility(){}
17
18 void f(){
19     utility();
20 }
21
22 // tool.impl2.cpp
23 module tool; // another implementation unit
24 void g(){}
25
26 // tool.helpers.cpp
27 export module tool:helpers; // module interface partition
28 import :internals;
29
30 export void h(){
31     utility();
32 }
33
34 // client.cpp
35 import tool;
36
```

```
37 f();
38 g();
39 h();
```

Note that partitions are imported without specifying module name. This prohibits importing other module's partitions. Multiple implementation units(`module tool;`) are allowed, all other units and partitions of any kind must be unique. All interface partitions must be re-exported by the module via `export import` .

Export

Here are various forms of `export` , the general rule is that you can't `export` names with internal linkage:

```
1 // tool.cpp
2 module tool;
3 export import :helpers; // import and re-export helpers interface partition
4
5 export int x{}; // export single declaration
6
7 export{           // export multiple declarations
8     int y{};
9     void f(){};
10 }
11
12 export namespace A{ // export the whole namespace
13     void f();
14     void g();
15 }
16
17 namespace B{
18     export void f();// export a single declaration within a namespace
19     void g();
20 }
21
22 namespace{
23     export int x;    // error, x has internal linkage
24     export void f();// error, f() has internal linkage
25 }
26
27 export class C; // export as incomplete type
28 class C{};
```

```

29 export C get_c();
30
31 // client.cpp
32 import tool;
33
34 C c1;    // error, C is incomplete
35 auto c2 = get_c(); // OK

```

Import

Import declarations should precede any other “non-module” declarations, it allows quick dependency analysis. Otherwise, it’s pretty intuitive:

```

1 // tool.cpp
2 export module tool;
3 import :helpers; // import helpers partition
4
5 export void f(){}
6
7 // tool.helpers.cpp
8 export module tool:helpers;
9
10 export void g(){}
11
12 // client.cpp
13 import tool;
14
15 f();
16 g();

```

Header units

There’s one special `import` form that allows import of *importable* headers: `import <header.h>` or `import "header.h"`. Compiler creates a synthesized *header unit* and makes all declarations implicitly exported. What headers are actually importable is implementation-defined but all C++ library headers are so. Perhaps, there will be a way to tell the compiler which user-provided headers are importable, such headers should not contain non-inline function definitions or variables with external linkage. It’s the only `import` form that allows import of macros from headers (but you still can’t re-export them via `export import "header.h"`). Don’t use it to import random legacy header if you’re not sure about its content.

Global module fragment

If you need to use old-school headers within a module, there's a special place to put

`#include`s safely: *global module fragment*:

```
1 // header.h
2 #pragma once
3 class A{};
4 void g(){}
5
6 // tool.cpp
7 module;           // global module fragment
8 #include "header.h"
9 export module tool; // ends here
10
11 export void f(){   // uses declarations from header.h
12     g();
13     A a;
14 }
```

It must appear before the named module declaration and it can contain only preprocessor directives. All declarations from all global module fragments and non-modular translation units are attached to a single global module. Thus, all rules for normal headers apply here.

Private module fragment

The final strange beast is a *private module fragment*. Its intent is to hide implementation details in a single-file module(it's not allowed elsewhere). In theory, clients might not recompile when things in a private module fragment changes:

```
1 export module tool; // interface
2
3 export void f();    // declared here
4
5 module :private;    // implementation details
6
7 void f(){}          // defined here
```

No more implicit `inline`

There's also an interesting change regarding `inline`. Member functions defined within the class definition are *not* implicitly `inline` if that class is attached to a named module.

`inline` functions in a named module can use only names that are visible to a client.

```
1 // header.h
2 struct C{
3     void f(){} // still inline because attached to a global module
4 };
5
6 // tool.cpp
7 module;
8 #include "header.h"
9
10 export module tool;
11
12 class A{}; // not exported
13
14 export struct B{// B is attached to module "tool"
15     void f(){ // not implicitly inline anymore
16         A a; // can safely use non-exported name
17     }
18
19     inline void g(){
20         A a; // oops, uses non-exported name
21     }
22
23     inline void h(){
24         f(); // fine, f() is not inline
25     }
26 };
27
28 // client.cpp
29 import tool;
30
31 B b;
32 b.f(); // OK
33 b.g(); // error, A is undefined
34 b.h(); // OK
```

Coroutines

Finally, we have stackless(their state is stored in heap, not on stack) [coroutines](#) in C++.

C++20 provides nearly the lowest possible API and leaves rest up to the user. We've got

`co_await`, `co_yield`, `co_return` keywords and rules for interaction between the caller and

callee. Those rules are so low-level that I see no point in explaining them here. You can find more details on [Lewis Baker's blog](#). Hopefully, C++23 will fill this gap with some library utilities. Until then, we can use third-party libraries, here's an example that uses [cppcoro](#):

```
1  cppcoro::task<int> someAsyncTask()
2  {
3      int result;
4      // get the result somehow
5      co_return result;
6  }
7
8  // task<> is analog of void for normal function
9  cppcoro::task<> usageExample()
10 {
11     // creates a new task but doesn't start executing the coroutine yet
12     cppcoro::task<int> myTask = someAsyncTask();
13     // ...
14     // Coroutine is only started when we later co_await the task.
15     auto result = co_await myTask;
16 }
17
18 // will lazily generate numbers from 0 to 9
19 cppcoro::generator<std::size_t> getTenNumbers()
20 {
21     std::size_t n{0};
22     while (n != 10)
23     {
24         co_yield n++;
25     }
26 }
27
28 void printNumbers()
29 {
30     for(const auto n : getTenNumbers())
31     {
32         std::cout << n;
33     }
34 }
```

Three-way comparison

Before C++20, to provide comparison operations for a class, implementations of 6 operators are needed: `==`, `!=`, `<`, `<=`, `>`, `>=`. Usually, four of them contain boiler-plate code that works in terms of `==` and `<` which contain the real comparison logic. Common practice is to implement them as free functions taking `const T&` to allow comparison of convertible types. If you want to support non-convertible types, you need to add two sets of 6 functions, `op(const T1&, const T2&)` and `op(const T2&, const T1&)` and now you have 18 comparison operators (check out `std::optional`). C++20 gives us a better way to handle and think about comparisons. Now you need to focus on `operator<=>()` and sometimes on `operator==()`. New `operator<=>` (spaceship operator) implements three-way comparison, it tells whether `a` is less, equal or greater than `b` in a single call, just like `strcmp()`. It returns a comparison category (see below) that could be compared to zero. Having this, compiler can replace calls to `<`, `<=`, `>`, `>=` with call to `operator<=>()` and check its result (`a < b` becomes `a <=> b < 0`), and calls to `==`, `!=` to `operator==()` (`a != b` becomes `!(a == b)`). Due to new lookup rules they can handle asymmetric comparisons, e.g. when you provide a single `T1::operator==(const T2&)`, you get both `T1 == T2` and `T2 == T1`, the same applies to `operator<=>()`. Now you need to write at most 2 functions to get all 6 comparisons between convertible types, and 2 functions to get all 12 comparisons between non-convertible types.

Comparison categories

The Standard provides three comparison categories (which doesn't prevent you from having your own one). `strong_ordering` implies that exactly one of `a < b`, `a > b`, `a == b` must be true and if `a == b` then `f(a) == f(b)`. `weak_ordering` implies that exactly one of `a < b`, `a > b`, `a == b` must be true and if `a == b` then `f(a)` can be *not* equal to `f(b)`. Such elements are equivalent but not equal. `partial_ordering` means that none of `a < b`, `a > b`, `a == b` might

be true and if `a == b` then `f(a)` can be not equal to `f(b)`. That is, some elements may be incomparable. Important note here is that `f()` denotes a function that accesses only *salient* attributes. For example, `std::vector<int>` is strongly ordered despite that two vectors with the same values can have different capacity. Here, capacity is not a salient attribute.

Example of a weakly ordered type is `CaseInsensitiveString`, it can store original string as-is but compare in a case-insensitive way. Example of a partially ordered type is `float/double` because `NaN` is not comparable to any other value. These categories form hierarchy, i.e., `strong_ordering` can be converted to `weak_ordering` and `partial_ordering`, and `weak_ordering` can be converted to `partial_ordering`.

Defaulted comparisons

Comparisons could be defaulted just like special member functions. In such case they operate in a member-wise fashion by comparing all underlying non-static data members with their corresponding operators. Defaulted `operator<=>()` also declares defaulted `operator==()` (if there was none), so you can write `auto operator<=>(const T&) const = default;` and get all six comparison operations with member-wise semantics.

```

1  template<typename T1, typename T2>
2  void TestComparisons(T1 a, T2 b)
3  {
4      (a < b), (a <= b), (a > b), (a >= b), (a == b), (a != b);
5  }
6
7  struct S2
8  {
9      int a;
10     int b;
11 };
12
13 struct S1
14 {
15     int x;
16     int y;
17     // support homogeneous comparisons
18     auto operator<=>(const S1&) const = default;
19     // this is required because there's operator==(const S2&) which preven
20     // implicit declaration of defaulted operator==(
21     bool operator==(const S1&) const = default;
22
23     // support heterogeneous comparisons
24     std::strong_ordering operator<=>(const S2& other) const
25     {
26         if (auto cmp = x <=> other.a; cmp != 0)
27             return cmp;
28         return y <=> other.b;
29     }
30
31     bool operator==(const S2& other) const
32     {
33         return (*this <=> other) == 0;
34     }
35 };
36
37 TestComparisons(S1{}, S1{});
38 TestComparisons(S1{}, S2{});
39 TestComparisons(S2{}, S1{});

```

Implicitly declared `operator==(X, X)` has the same signature as `operator<=>(X, X)` except that return type is `bool`.

```
1 | template<typename T>
2 | struct X
3 | {
4 |     friend constexpr std::partial_ordering operator<=>(X, X) requires(sizeof(T) != 1) = default;
5 |     // implicitly declares:
6 |     // friend constexpr bool operator==(X, X) requires(sizeof(T) != 1) = default;
7 |
8 |     [[nodiscard]] virtual std::strong_ordering operator<=>(const X&) const = default;
9 |     // implicitly declares:
10 |    // [[nodiscard]] virtual bool operator==(const X&) const = default;
11 | };
```

Deduced comparison category is the weakest one of type's members.

```
1 | struct S3{
2 |     int x;      // int-s are strongly ordered
3 |     double d;   // but double-s are partially ordered
4 |     // thus, the resulting category is std::partial_ordering
5 |     auto operator<=>(const S3&) const = default;
6 | };
7 | static_assert(std::is_same_v<decltype(S3{} <=> S3{}), std::partial_ordering>);
```

They must be members or friends and only friends can take by-value.

```
1 | struct S4
2 | {
3 |     int x;
4 |     int y;
5 |     // member version must have op(const T&) const; form
6 |     auto operator<=>(const S3&) const = default;
7 |
8 |     // friend version can take arguments by const-reference or by-value
9 |     // friend auto operator<=>(const S3&, const S3&) = default;
10 |    // friend auto operator<=>(S3, S3) = default;
11 | };
```

Can be out-of-class defaulted, just like special member functions.

```

1 struct S5
2 {
3     int x;
4     std::strong_ordering operator<=>(const S5&) const;
5     bool operator==(const S5&) const;
6 };
7
8 std::strong_ordering S5::operator<=>(const S5&) const = default;
9 bool S5::operator==(const S5&) const = default;

```

Defaulted `operator<=>()` uses `operator<=>()` of class members or their ordering can be synthesized using existing `Member::operator==()` and `Member::operator<()`. Note that it works only for members and not for the class itself, existing `T::operator<()` is never used in defaulted `T::operator<=>()`.

```

1 // not in our immediate control
2 struct Legacy
3 {
4     bool operator==(Legacy const&) const;
5     bool operator<(Legacy const&) const;
6 };
7
8 struct S6
9 {
10     int x;
11     Legacy l;
12     // deleted because Legacy doesn't have operator<=>(), comparison category
13     // can't be deduced
14     auto operator<=>(const S6&) const = default;
15 };
16
17 struct S7
18 {
19     int x;
20     Legacy l;
21
22     std::strong_ordering operator<=>(const S7& rhs) const = default;
23     /*
24     Since comparison category is provided explicitly, ordering can be
25     synthesized using operator<() and operator==(). They must return exact
26     'bool' for this to work. It will work for weak and partial ordering as
27

```

```

28     Here's an example of synthesized operator<=>():
29     std::strong_ordering operator<=>(const S7& rhs) const
30     {
31         // use operator<=>() for int
32         if(auto cmp = x <=> rhs.x; cmp != 0) return cmp;
33
34         // synthesize ordering for Legacy using operator<() and operator==()
35         if(l == rhs.l) return std::strong_ordering::equal;
36         if(l < rhs.l) return std::strong_ordering::less;
37         return std::strong_ordering::greater;
38     }
39     */
40 };
41
42 struct NoEqual
43 {
44     bool operator<(const NoEqual&) const = default;
45 };
46
47 struct S8
48 {
49     NoEqual n;
50     // deleted, NoEqual doesn't have operator<=>()
51     // auto operator<=>(const S8&) const = default;
52
53     // deleted as well because NoEqual doesn't have operator==()
54     std::strong_ordering operator<=>(const S8&) const = default;
55 };
56
57 struct W
58 {
59     std::weak_ordering operator<=>(const W&) const = default;
60 };
61
62 struct S9
63 {
64     W w;
65     // ask for strong_ordering but W can provide only weak_ordering, this will
66     // yield an error during instantiation
67     std::strong_ordering operator<=>(const S9&) const = default;
68     void f()
69     {
70         (S9{} <=> S9{}); // error

```

```
71 |     }  
72 | };
```

`union` and reference members are not supported.

```
1 | struct S4  
2 | {  
3 |     int& r;  
4 |     // deleted because of reference member  
5 |     auto operator<=>(const S4&) const = default;  
6 | };
```

Lambda expressions

Allow lambda-capture `[=, this]`

When captured implicitly, `this` is always captured by-reference, even with `[=]`. To remove this confusion, C++20 deprecates such behavior and allows more explicit `[=, this]`:

```
1 | struct S{  
2 |     void f(){  
3 |         [=]{};           // captures this by reference, deprecated since C++20  
4 |         [=, *this]{};    // OK since C++17, captures this by value  
5 |         [=, this]{};     // OK since C++20, captures this by reference  
6 |     }  
7 | };
```

Template parameter list for generic lambdas

Sometimes generic lambdas are too generic. C++20 allows to use familiar template function syntax to introduce type names directly.

```
1 | // lambda that expect std::vector<T>  
2 | // until C++20:  
3 | [](auto vector){  
4 |     using T =typename decltype(vector)::value_type;  
5 |     // use T  
6 | };  
7 | // since C++20:
```

```

8  [<typename T>(std::vector<T> vector){
9      // use T
10 };
11
12 // access argument type
13 // until C++20
14 [(const auto& x){
15     using T = std::decay_t<decltype(x)>;
16     // using T = decltype(x); // without decay_t<> it would be const T&, so
17     T copy = x;                // copy would be a reference type
18     T::static_function();      // and these wouldn't work at all
19     using Iterator = typename T::iterator;
20 };
21 // since C++20
22 [<typename T>(const T& x){
23     T copy = x;
24     T::static_function();
25     using Iterator = typename T::iterator;
26 };
27
28 // perfect forwarding
29 // until C++20:
30 [(auto&&... args){
31     return f(std::forward<decltype(args)>(args)...);
32 };
33 // since C++20:
34 [<typename... Ts>(Ts&&... args){
35     return f(std::forward<Ts>(args)...);
36 };
37
38 // and of course you can mix them with auto-parameters
39 [<typename T>(const T& a, auto b){};

```

Lambdas in unevaluated contexts

Lambda expressions can be used in unevaluated contexts, such as `sizeof()`, `typeid()`, `decltype()`, etc. Here are some key points for this feature, for a more real-world example see [Default constructible and assignable stateless lambdas](#).

The main principle is that lambdas have a unique unknown type, two lambdas and their types are never equal.


```

1  using L = decltype([]{}); // lambdas have no linkage
2  L PublicApi();           // L can't be used for external linkage
3
4  // in template , two different declarations
5  template<class T> void f(decltype([]{})) (*s)[sizeof(T)];
6  template<class T> void f(decltype([]{})) (*s)[sizeof(T)];
7
8  // again, lambda types are never equivalent
9  static decltype([]{}) f();
10 static decltype([]{}) f(); // error, return type mismatch
11
12 static decltype([]{}) g();
13 static decltype(g()) g(); // okay, redeclaration
14
15 // each specialization has its own lambda with unique type
16 template<typename T>
17 using R = decltype([]{});
18
19 static_assert(!std::is_same_v<R<int>, R<char>>);
20
21 // Lambda-based SFINAE and constraints are not supported, it just fails
22 template <class T>
23 auto f(T) -> decltype([]() { T::invalid; } ());
24 void f(...);
25
26 template<typename T>
27 void g(T) requires requires{
28     [](){typename T::invalid x;}; }
29 {}
30 void g(...){}
31
32 f(0); // error
33 g(0); // error

```

In the following example, `f()` increments the same counter in both translation units because `inline` function behaves as if there's only one definition of it. However, `g_s` violates ODR because despite that there's only one definition of it, there are still multiple declarations which are different because there are two different lambdas in `a.cpp` and `b.cpp`, thus, `S` has different non-type template argument:

```

1  // a.h
2  template<typename T>

```

```

3  int counter(){
4      static int value{};
5      return value++;
6  }
7
8  inline int f(){
9      return counter<decltype([]{})>();
10 }
11
12 template<auto> struct S{ void call(){} };
13 // cast lambda to pointer
14 inline S<+[]{}> g_s;
15
16 // a.cpp
17 #include "a.h"
18 auto v = f();
19 g_s.call();
20
21 // b.cpp
22 #include "a.h"
23 auto v = f();
24 g_s.call();

```

Default constructible and assignable stateless lambdas

In C++20 stateless lambdas are default constructible and assignable which allows to use a type of a lambda to construct/assign it later. With [Lambdas in unevaluated contexts](#) we can get a type of a lambda with `decltype()` and create a variable of that type later:

```

1  auto greater = [](auto x, auto y)
2  {
3      return x > y;
4  };
5  // requires default constructible type
6  std::map<std::string, int, decltype(greater)> map;
7  auto map2 = map;    // requires default assignable type

```

Here, `std::map` takes a comparator type to instantiate it later. While we could get a lambda type in C++17, it was not possible to instantiate it because lambdas were not default constructible.

Pack expansion in lambda init-capture

C++20 simplifies capturing parameter packs in lambdas. Until C++20 they can be captured by-value, by-reference or do some tricks with `std::tuple` if we want to move the pack. Now it's much easier, we can create *init-capture pack* and initialize it with the pack we want to capture. It's not limited to `std::move` or `std::forward`, any function can be applied to pack elements.

```
1 void g(int, int){}
2
3 // C++17
4 template<class F, class... Args>
5 auto delay_apply(F&& f, Args&&... args) {
6     return [f=std::forward<F>(f), tup=std::make_tuple(std::forward<Args>(a
7         -> decltype(auto) {
8         return std::apply(f, tup);
9     }];
10 }
11
12 // C++20
13 template<typename F, typename... Args>
14 auto delay_call(F&& f, Args&&... args) {
15     return [f = std::forward<F>(f), ...f_args=std::forward<Args>(args)]()
16         -> decltype(auto) {
17         return f(f_args...);
18     };
19 }
20
21 void f(){
22     delay_call(g, 1, 2)();
23 }
```

Constant expressions

Immediate functions(`constexpr`)

While `constexpr` implies that function *can* be evaluated at compile-time, `constexpr` specifies that function *must* be evaluated at compile-time(only). `virtual` functions are allowed to be `constexpr` but they can override and be overridden by another `constexpr` function only, i.e., mix of `constexpr` and non-`constexpr` is not allowed. Destructors and allocation/deallocation functions can't be `constexpr`.

```

1  constexpr int GetInt(int x){
2      return x;
3  }
4
5  constexpr void f(){
6      auto x1 = GetInt(1);
7      constexpr auto x2 = GetInt(x1); // error x1 is not a constant-expression
8  }

```

constexpr virtual function

Virtual functions can now be `constexpr`. `constexpr` function can override non-`constexpr` one and vice-versa.

```

1  struct Base{
2      constexpr virtual ~Base() = default;
3      virtual int Get() const = 0;    // non-constexpr
4  };
5
6  struct Derived1 : Base{
7      constexpr int Get() const override {
8          return 1;
9      }
10 };
11
12 struct Derived2 : Base{
13     constexpr int Get() const override {
14         return 2;
15     }
16 };
17
18 constexpr auto GetSum(){
19     const Derived1 d1;
20     const Derived2 d2;
21     const Base* pb1 = &d1;
22     const Base* pb2 = &d2;
23
24     return pb1->Get() + pb2->Get();
25 }
26
27 static_assert(GetSum() == 1 + 2);    // evaluated at compile-time

```

constexpr try-catch blocks

try-catch blocks are now allowed inside `constexpr` functions but `throw` is not, so, the `catch` block is simply ignored. This can be useful, for example, in combination with `constexpr new`, we can have single function that works at run/compile time:

```
1  constexpr void f(){
2      try{
3          auto p = new int;
4          // ...
5          delete p;
6      }
7      catch(...){    // ignored at compile-time
8          // ...
9      }
10 }
```

constexpr dynamic_cast and polymorphic typeid

Since virtual functions can now be `constexpr`, there's no reason not to allow `dynamic_cast` and polymorphic `typeid` in `constexpr`. Unfortunately, `std::type_info` has no `constexpr` members yet so there's a little use of it now (thanks to Peter Dimov for clarifying this for me).

```
1  struct Base1{
2      virtual ~Base1() = default;
3      constexpr virtual int get() const = 0;
4  };
5
6  struct Derived1 : Base1{
7      constexpr int get() const override {
8          return 1;
9      }
10 };
11
12 struct Base2{
13     virtual ~Base2() = default;
14     constexpr virtual int get() const = 0;
15 };
16
17 struct Derived2 : Base2{
18     constexpr int get() const override {
```

```

19         return 2;
20     }
21 };
22
23 template<typename Base, typename Derived>
24 constexpr auto downcasted_get(){
25     const Derived d;
26     const Base& upcasted = d;
27     const auto& downcasted = dynamic_cast<const Derived&>(upcasted);
28
29     return downcasted.get();
30 }
31
32 static_assert(downcasted_get<Base1, Derived1>() == 1);
33 static_assert(downcasted_get<Base2, Derived2>() == 2);
34
35 // compile-time error, cannot cast Derived1 to Base2
36 static_assert(downcasted_get<Base2, Derived1>() == 1);

```

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

Another relaxation for constant expressions. One can change an active member of a `union` but can't read an inactive member since it's UB and UB is not allowed in `constexpr` context.

```

1  union Foo {
2      int i;
3      float f;
4  };
5
6  constexpr int f() {
7      Foo foo{};
8      foo.i = 3;    // i is an active member
9      foo.f = 1.2f; // valid since C++20, f becomes an active member
10
11     // return foo.i; // error, reading inactive union member
12     return foo.f;
13 }

```

`constexpr` allocations

C++20 lays foundation for `constexpr` containers. First, it allows `constexpr` and even `virtual constexpr` destructors for *literal* types(types that can be used as a `constexpr`

variable). Second, it allows calls to `std::allocator<T>::allocate()` and `new-expression` which results in a call to one of the global `operator new` if allocated storage is deallocated at compile time. That is, memory can be allocated at compile-time but it must be freed at compile-time also. This creates a bit of friction if final data has to be used at run-time. There's no choice but to store it in some non-allocating container like `std::array` and get compile-time value twice: first, to get its size, and second, to actually copy it(thanks to **arthur-odwyer**, **beached** and **luke** from [cpplang slack](#) for explaining this to me):

```
1  constexpr auto get_str()
2  {
3      std::string s1{"hello "};
4      std::string s2{"world"};
5      std::string s3 = s1 + s2;
6      return s3;
7  }
8
9  constexpr auto get_array()
10 {
11     constexpr auto N = get_str().size();
12     std::array<char, N> arr{};
13     std::copy_n(get_str().data(), N, std::begin(arr));
14     return arr;
15 }
16
17 static_assert(!get_str().empty());
18
19 // error because it holds data allocated at compile-time
20 constexpr auto str = get_str();
21
22 // OK, string is stored in std::array<char>
23 constexpr auto result = get_array();
```

Trivial default initialization in `constexpr` functions

In C++17 `constexpr` constructor, among other requirements, must initialize all non-static data members. This rule has been removed in C++20. But, because UB is not allowed in `constexpr` context, you can't read from such uninitialized members, only write to them:

```
1  struct NonTrivial{
2      bool b = false;
3  };
4
```

```

5 struct Trivial{
6     bool b;
7 };
8
9 template <typename T>
10 constexpr T f1(const T& other) {
11     T t;           // default initialization
12     t = other;
13     return t;
14 }
15
16 template <typename T>
17 constexpr auto f2(const T& other) {
18     T t;
19     return t.b;
20 }
21
22 void test(){
23     constexpr auto a = f1(Trivial{}); // error in C++17, OK in C++20
24     constexpr auto b = f1(NonTrivial{}); // OK
25
26     constexpr auto c = f2(Trivial{}); // error, uninitialized Trivial::b is
27     constexpr auto d = f2(NonTrivial{}); // OK
28 }

```

Unevaluated `asm`-declaration in `constexpr` functions

`asm`-declaration now can appear inside `constexpr` function in case it's not evaluated at compile-time. This allows to have both compile and run time(with `asm` now) code inside a single function:

```

1 constexpr int add(int a, int b){
2     if (std::is_constant_evaluated()){
3         return a + b;
4     }
5     else{
6         asm("asm magic here");
7         //...
8     }
9 }

```


std::is_constant_evaluated()

With `std::is_constant_evaluated()` you can check whether current invocation occurs within a constant-evaluated context. I would like to say “during compile-time” but, as the authors said, “C++ doesn’t make a clear distinction between compile-time and run-time”. Instead, C++20 declares a [list](#) of expressions that are *manifestly constant-evaluated* and this function returns `true` during their evaluation and `false` otherwise.

Be careful not to use this function directly in such *manifestly constant-evaluated* expressions (e.g. `if constexpr`, array size, template arguments, etc.). By definition, in such cases `std::is_constant_evaluated()` returns `true` even if the enclosing function is not constant evaluated. Thanks to user **destroyerrocket** from [/r/cpp](#) for bringing up this issue.

```
1  constexpr int GetNumber(){
2      if(std::is_constant_evaluated()){ // should not be `if constexpr`
3          return 1;
4      }
5      return 2;
6  }
7
8  constexpr int GetNumber(int x){
9      if(std::is_constant_evaluated()){ // should not be `if constexpr`
10         return x;
11     }
12     return x+1;
13 }
14
15 void f(){
16     constexpr auto v1 = GetNumber();
17     const auto v2 = GetNumber();
18
19     // initialization of a non-const variable, not constant-evaluated
20     auto v3 = GetNumber();
21
22     assert(v1 == 1);
23     assert(v2 == 1);
24     assert(v3 == 2);
25
26     constexpr auto v4 = GetNumber(1);
27     int x = 1;
28
29     // x is not a constant-expression, not constant-evaluated
30     const auto v5 = GetNumber(x);
```

```

31     assert(v4 == 1);
32     assert(v5 == 2);
33 }
34
35 // pathological examples
36 // always returns `true`
37 constexpr bool IsInConstexpr(int){
38     if constexpr(std::is_constant_evaluated()){ // always `true`
39         return true;
40     }
41     return false;
42 }
43
44 // always returns `sizeof(int)`
45 constexpr std::size_t GetArraySize(int){
46     int arr[std::is_constant_evaluated()]; // always int arr[1];
47     return sizeof(arr);
48 }
49
50 // always returns `1`
51 constexpr std::size_t GetStdArraySize(int){
52     std::array<int, std::is_constant_evaluated()> arr; // std::array<int,
53     return arr.size();
54 }
55

```

Aggregates

Prohibit aggregates with user-declared constructors

Now aggregate types can't have *user-declared* constructors. Previously, aggregates were allowed to have only deleted or defaulted constructors. That resulted in a weird behavior for aggregates with defaulted/deleted constructors (they're *user-declared* but not *user-provided*).

```

1 // none of the types below are an aggregate in C++20
2 struct S{
3     int x{2};
4     S(int) = delete; // user-declared ctor
5 };
6
7 struct X{
8     int x;

```

```

9      X() = default; // user-declared ctor
10 };
11
12 struct Y{
13     int x;
14     Y();           // user-provided ctor
15 };
16
17 Y::Y() = default;
18
19 void f(){
20     S s(1);        // always an error
21     S s2{1};       // OK in C++17, error in C++20, S is not an aggregate now
22     X x{1};        // OK in C++17, error in C++20
23     Y y{2};        // always an error
24 }

```

Class template argument deduction for aggregates

In C++17 to use aggregates with CTAD we need explicit deduction guides, that's unnecessary now:

```

1  template<typename T, typename U>
2  struct S{
3      T t;
4      U u;
5  };
6  // deduction guide was needed in C++17
7  // template<typename T, typename U>
8  // S(T, U) -> S<T,U>;
9
10 S s{1, 2.0};    // S<int, double>

```

CTAD isn't involved when there are user-provided deduction guides:

```

1  template<typename T>
2  struct MyData{
3      T data;
4  };
5  MyData(const char*) -> MyData<std::string>;
6

```

```

7 | MyData s1{"abc"}; // OK, MyData<std::string> using deduction guide
8 | MyData<int> s2{1}; // OK, explicit template argument
9 | MyData s3{1}; // Error, CTAD isn't involved

```

Can deduce array types:

```

1 | template<typename T, std::size_t N>
2 | struct Array{
3 |     T data[N];
4 | };
5 |
6 | Array a{{1, 2, 3}}; // Array<int, 3>, notice additional braces
7 | Array str{"hello"}; // Array<char, 6>

```

Brace elision doesn't work for dependent non-array types or array types of dependent bound.

```

1 | template<typename T, typename U>
2 | struct Pair{
3 |     T first;
4 |     U second;
5 | };
6 |
7 | template<typename T, std::size_t N>
8 | struct A1{
9 |     T data[N];
10 |    T oneMore;
11 |    Pair<T, T> p;
12 | };
13 |
14 | template<typename T>
15 | struct A2{
16 |     T data[3];
17 |     T oneMore;
18 |     Pair<int, int> p;
19 | };
20 |
21 | // A1::data is an array of dependent bound and A1::p is a dependent type,
22 | // no brace elision for them
23 | A1 a1{{1,2,3}, 4, {5, 6}}; // A1<int, 3>
24 | // A2::data is an array of non-dependent bound and A1::p is a non-dependent
25 |

```

```

26 | // thus, brace elision works
    | A2 a2{1, 2, 3, 4, 5, 6}; // A2<int>

```

Works with pack expansions. Trailing aggregate element that is a pack expansion corresponds to all remaining elements:

```

1 | template<typename... Ts>
2 | struct Overload : Ts...{
3 |     using Ts::operator()...;
4 | };
5 | // no need for deduction guide anymore
6 |
7 | Overload p{[](int){
8 |     std::cout << "called with int";
9 | }, [](char){
10 |     std::cout << "called with char";
11 | }
12 | }; // Overload<lambda(int), lambda(char)>
13 | p(1); // called with int
14 | p('c'); // called with char

```

Non-trailing element that is a pack expansions corresponds to no elements:

```

1 | template<typename T, typename...Ts>
2 | struct Pack : Ts... {
3 |     T x;
4 | };
5 |
6 | // can deduce only the first element
7 | Pack p1{1}; // Pack<int>
8 | Pack p2[[]{}]; // Pack<lambda()>
9 | Pack p3{1, []{}]; // error

```

Number of elements in the pack is deduced only once but types should match exactly if repeated:

```

1 | struct A{};
2 | struct B{};
3 | struct C{};
4 | struct D{
5 |     operator C(){return C{}};

```

```

6   };
7
8   template<typename...Ts>
9   struct P : std::tuple<Ts..., Ts...>{
10  };
11
12  P{std::tuple<A, B, C>{}, A{}, B{}, C{}}; // P<A, B, C>
13
14  // equivalent to the above, since pack elements were deduced for
15  // std::tuple<A, B, C> there's no need to repeat their types
16  P{std::tuple<A, B, C>{}, {}, {}, {}}; // P<A, B, C>
17
18  // since we know the whole P<A, B, C> type after std::tuple initializer, we
19  // omit trailing initializers, elements will be value-initialized as usual
20  P{std::tuple<A, B, C>{}, {}, {}}; // P<A, B, C>
21
22  // error, pack deduced from first initializer is <A, B, C> but got <A, B, D>
23  // the trailing pack, implicit conversions are not considered
24  P{std::tuple<A, B, C>{}, {}, {}, D{}};

```

Parenthesized initialization of aggregates

Parenthesized initialization of aggregates now works in the same way as braced initialization except that narrowing conversions are permitted, designated initializers are not allowed, no lifetime extension for temporaries and no brace elision. Elements without initializer are value-initialized. This allows seamless usage of factory functions like `std::make_unique<>()`/`emplace()` with aggregates.

```

1   struct S{
2       int a;
3       int b = 2;
4       struct S2{
5           int d;
6       } c;
7   };
8
9   struct Ref{
10      const int& r;
11  };
12
13  int GetInt(){

```

```

14     return 21;
15 }
16
17 S{0.1}; // error, narrowing
18 S(0.1); // OK
19
20 S{.a=1}; // OK
21 S(.a=1); // error, no designated initializers
22
23 Ref r1{GetInt()}; // OK, lifetime is extended
24 Ref r2(GetInt()); // dangling, lifetime is not extended
25
26 S{1, 2, 3}; // OK, brace elision, same as S{1,2,{3}}
27 S(1, 2, 3); // error, no brace elision
28
29 // values without initializers take default values or value-initialized(T{})
30 S{1}; // {1, 2, 0}
31 S(1); // {1, 2, 0}
32
33 // make_unique works now
34 auto ps = std::make_unique<S>(1, 2, S::S2{3});
35
36 // arrays are also supported
37 int arr1[](1, 2, 3);
38 int arr2[2](1); // {1, 0}

```

Non-type template parameters

Class types in non-type template parameters

Non-type template parameters now can be of literal class types(types that can be used as a `constexpr` variable) with all bases and non-static members being `public` and non-`mutable` (literally, there should be no `mutable` specifier). Instances of such classes are stored as `const` objects and you can even call their member functions. There's a new kind of non-type template parameter: *placeholder for a deduced class type*. In the example below, `fixed_string` is a template name, not a type name, but we can use it to declare template parameter `template<fixed_string S>`. In such a case, the compiler will deduce template arguments for `fixed_string` before instantiating `f<>()` using an invented declaration in the form of `T x = template-argument;`. Here's how it can be used to create a simple compile-time string class:

```

1  template<std::size_t N>
2  struct fixed_string{
3      constexpr fixed_string(const char (&s)[N+1]) {
4          std::copy_n(s, N + 1, str);
5      }
6      constexpr const char* data() const {
7          return str;
8      }
9      constexpr std::size_t size() const {
10         return N;
11     }
12
13     char str[N+1];
14 };
15
16 template<std::size_t N>
17 fixed_string(const char (&)[N])->fixed_string<N-1>;
18
19 // user-defined literals are also supported
20 template<fixed_string S>
21 constexpr auto operator""_cts(){
22     return S;
23 }
24
25 // N for `S` will be deduced
26 template<fixed_string S>
27 void f(){
28     std::cout << S.data() << ", " << S.size() << '\n';
29 }
30
31 f<"abc">(); // abc, 3
32 constexpr auto s = "def"_cts;
33 f<s>();     // def, 3

```

Generalized non-type template parameters

Non-type template parameters are generalized to so-called *structural* types. Structural type is one of:

- scalar type(arithmetic, pointer, pointer-to-member, enumeration, `std::nullptr_t`)
- lvalue reference

- literal class type with the following properties: all base classes and non-static data members are public and non-`mutable`, and their types are structural or array types.

This allows usage of floating-point and class types as a template parameters:

```

1  template<auto T>    // placeholder for any non-type template parameter
2  struct X{};
3
4  template<typename T, std::size_t N>
5  struct Arr{
6      T data[N];
7  };
8
9  X<5> x1;
10 X<'c'> x2;
11 X<1.2> x3;
12 // with the help of CTAD for aggregates
13 X<Arr{{1,2,3}}> x4; // X<Arr<int, 3>>
14 X<Arr{"hi"}> x5;    // X<Arr<char, 3>>

```

Interesting moment here is that non-type template arguments are compared not with their `operator==()` but in a bitwise-*like* manner(the exact rules are [here](#)). That is, their bit representation is used for comparison. `union`s are exceptions because the compiler can track their active members. Two unions are equal if they both have no active member or have the same active member with equal value.

```

1  template<auto T>
2  struct S{};
3
4  union U{
5      int a;
6      int b;
7  };
8
9  enum class E{
10     A = 0,
11     B = 0
12 };
13
14 struct C{
15     int x;
16     bool operator==(const C&) const{    // never equal
17         return false;

```

```

18     }
19 };
20
21 constexpr C c1{1};
22 constexpr C c2{1};
23 assert(c1 != c2); // not equal using operator==(
24 assert(memcmp(&c1, &c2, sizeof(C)) == 0); // but equal bitwise
25 // thus, equal at compile-time, operator==(
26 static_assert(std::is_same_v<S<c1>, S<c2>>);
27
28 constexpr E e1{E::A};
29 constexpr E e2{E::B};
30 // equal bitwise, enum's identity isn't taken into account
31 assert(memcmp(&e1, &e2, sizeof(E)) == 0);
32 static_assert(std::is_same_v<S<e1>, S<e2>>); // thus, equal at compile-time
33
34 constexpr U u1{.a=1};
35 constexpr U u2{.b=1};
36 // equal bitwise but have different active members(a vs. b)
37 assert(memcmp(&u1, &u2, sizeof(U)) == 0);
38 // thus, not equal at compile-time
39 static_assert(!std::is_same_v<S<u1>, S<u2>>);

```

Structured bindings

Lambda capture and storage class specifiers for structured bindings

Structured bindings are allowed to have `[[maybe_unused]]` attribute, `static` and `thread_local` specifiers. Also, it's possible now to capture them by-value or by-reference in lambdas. Note that bound bit-fields can be captured only by-value.

```

1 struct S{
2     int a: 1;
3     int b: 1;
4     int c;
5 };
6
7 static auto [A,B,C] = S{};
8

```

```

9 void f(){
10     [[maybe_unused]] thread_local auto [a,b,c] = S{};
11     auto l = [=]() {
12         return a + b + c;
13     };
14
15     auto m = [&]() {
16         // error, can't capture bit-fields 'a' and 'b' by-reference
17         // return a + b + c;
18         return c;
19     };
20 }
21

```

Relaxing the structured bindings customization point finding rules

One of ways for a type to be decomposed for structured bindings is through a tuple-like API. It consists of three “functions”: `std::tuple_element`, `std::tuple_size` and two options for `get`: `e.get<I>()` or `get<I>(e)` where the first has priority over the second. That is, the member `get()` is preferred over non-member one. Imagine a type that has `get()` but it's not for a tuple-like API, for example `std::shared_ptr::get()`. Such a type can't be decomposed because the compiler will try to use member `get()` and it won't work. Now this rule has been fixed in a way that the member version is preferred only if it's a template and its first template parameter is a non-type template parameter.

```

1 struct X : private std::shared_ptr<int>{
2     std::string payload;
3 };
4
5 // due to new rules, this function is used instead of std::shared_ptr<int>
6 template<int N>
7 std::string& get(X& x) {
8     if constexpr(N==0) return x.payload;
9 }
10
11 namespace std {
12     template<>
13     class tuple_size<X>
14         : public std::integral_constant<int, 1>
15     {};
16

```

```

17     template<>
18     class tuple_element<0, X> {
19     public:
20         using type = std::string;
21     };
22 }
23
24 void f(){
25     X x;
26     auto& [payload] = x;
27 }

```

Allow structured bindings to accessible members

This fix allows structured bindings not only to `public` members but to *accessible* members in the context of structured binding declaration.

```

1 struct A {
2     friend void foo();
3 private:
4     int i;
5 };
6
7 void foo() {
8     A a;
9     auto x = a.i;    // OK
10    auto [y] = a;    // Ill-formed until C++20, now OK
11 }

```

Range-based for-loop

init-statements for range-based for-loop

Similar to if-statement, range-based for-loop now can have init-statement. It can be used to avoid dangling references:

```

1 class Obj{
2     std::vector<int>& GetItems();
3 };
4

```

```

5  Obj GetObj();
6
7  // dangling reference, lifetime of Obj return by GetObj() is not extended
8  for(auto x : GetObj().GetCollection()){
9      // ...
10 }
11
12 // OK
13 for(auto obj = GetObj(); auto item : obj.GetCollection()){
14     // ...
15 }
16
17 // also can be used to maintain index
18 for(std::size_t i = 0; auto& v : collection){
19     // use v...
20     i++;
21 }

```

Relaxing the range-based for-loop customization point finding rules

This one is similar to [structured bindings customization point fix](#). To iterate over a range, range-based for-loop needs either free or member `begin / end` functions. Old rules worked in a way that if *any* member(function or variable) named `begin / end` was found then the compiler would try to use member functions. This creates a problem for types that have a member `begin` but no `end` or vice versa. Now member functions are used only if both names exist, otherwise free functions are used.

```

1  struct X : std::stringstream {
2      // ...
3  };
4
5  std::istream_iterator<char> begin(X& x){
6      return std::istream_iterator<char>(x);
7  }
8
9  std::istream_iterator<char> end(X& x){
10     return std::istream_iterator<char>();
11 }
12
13 void f(){

```

```

14     X x;
15     // X has member with name `end` inherited from std::stringstream
16     // but due to new rules free begin()/end() are used
17     for (auto&& i : x) {
18         // ...
19     }
20 }

```

Attributes

`[[likely]]` and `[[unlikely]]`

`[[likely]]` and `[[unlikely]]` attributes give a hint to the compiler about likeliness of execution path so it can better optimize the code. They can be applied to statements(e.g. `if/else` -statements, loops) or labels(`case/default`).

```

1  int f(bool b){
2      if(b) [[likely]] {
3          return 12;
4      }
5      else{
6          return 10;
7      }
8  }

```

`[[no_unique_address]]`

`[[no_unique_address]]` can be applied to a non-static non-bitfield data member to indicate that it doesn't need a unique address. In practice, it's applied to a potentially empty data member and the compiler can optimize it to occupy no space(like empty base optimization for members). Such a member can share the address of another member or base class.

```

1  struct Empty{};
2
3  template<typename T>
4  struct Cpp17Widget{
5      int i;
6      T t;
7  };
8
9  template<typename T>

```

```

10 struct Cpp20Widget{
11     int i;
12     [[no_unique_address]] T t;
13 };
14
15 static_assert(sizeof(Cpp17Widget<Empty>) > sizeof(int));
16 static_assert(sizeof(Cpp20Widget<Empty>) == sizeof(int));

```

[[nodiscard]] with message

Like `[[deprecated("reason")]]`, `nodiscard` now can have a reason too.

```

1 // test whether it's supported
2 static_assert(__has_cpp_attribute(nodiscard) == 201907L);
3
4 [[nodiscard("Don't leave me alone")]]
5 int get();
6
7 void f(){
8     get(); // warning: ignoring return value of function declared with
9           // 'nodiscard' attribute: Don't leave me alone
10 }

```

[[nodiscard]] for constructors

This fix explicitly allows applying `[[nodiscard]]` to constructors(compilers were not required to support it prior to C++20).

```

1 struct resource{
2     // empty resource, no harm if discarded
3     resource() = default;
4
5     [[nodiscard("don't discard non-empty resource")]]
6     resource(int fd);
7 };
8
9 void f(){
10     resource{}; // OK
11     resource{1}; // warning
12 }

```

Character encoding

`char8_t`

C++17 introduced the `u8` character literal for UTF-8 string but its type was plain `char`. The inability to distinguish encoding by a type resulted in a code that had to use various tricks to handle different encodings. A new `char8_t` type was introduced to represent UTF-8 characters. It has the same size, signedness, alignment, etc, as `unsigned char` but it's a distinct type, not an alias.

```
1 void HandleString(const char*){}
2 // distinct function name is required to handle UTF-8 in C++17
3 void HandleStringUTF8(const char*){}
4 // now it can be done using convenient overload
5 void HandleString(const char8_t*){}
6
7 void Cpp17(){
8     HandleString("abc"); // char[4]
9     HandleStringUTF8(u8"abc"); // C++17: char[4] but UTF-8,
10                                // C++20: error, type is char8_t[4]
11 }
12
13 void Cpp20(){
14     HandleString("abc"); // char
15     HandleString(u8"abc"); // char8_t
16 }
```

Stronger Unicode requirements

Types `char16_t` and `char32_t` are now explicitly required to represent UTF-16 and UTF-32 string literals correspondingly. Universal character names(`\Uxxxxxxxx` and `\uNNNN`) must correspond to ISO/IEC 10646 code points (0x0 - 0x10FFFF inclusive) and not to a surrogate code points (0xD800 - 0xDFFF inclusive), otherwise the program is ill-formed.

```
1 | char32_t c{'\U00110000'}; // error: invalid universal character
```

Sugar

Designated initializers

Now it's possible to initialize specific(designated) aggregate members and skip others. Unlike C, initialization order must be the same as in aggregate declaration:

```
1 struct S{
2     int x;
3     int y{2};
4     std::string s;
5 };
6 S s1{.y = 3}; // {0, 3, {}}
7 S s2 = {.x = 1, .s = "abc"}; // {1, 2, {"abc"}}
8 S s3{.y = 1, .x = 2}; // Error, x should be initialized before y
```

Default member initializers for bit-fields

Until C++20, to provide default value for a bit-field one had to create a default constructor, now that can be achieved using convenient default member initialization syntax:

```
1 // until C++20:
2 struct S{
3     int a : 1;
4     int b : 1;
5     S() : a{0}, b{1}{}
6 };
7
8 // since C++20:
9 struct S{
10     int a : 1 {0},
11     int b : 1 = 1;
12 };
```

More optional `typename`

`typename` can be omitted in contexts where nothing but a type name can appear(type in casts, return type, type aliases, member type, argument type of a member function, etc.):

```
1 template <class T>
2 T::R f(); // OK, return type of a function declaration at global scope
3
4 template <class T>
5 void f(T::R); // Ill-formed (no diagnostic required), attempt to declare
6               // void variable template
```

```

7
8  template<typename T>
9  struct PtrTraits{
10     using Ptr = void*;
11 };
12
13 template <class T>
14 struct S {
15     using Ptr = PtrTraits<T>::Ptr; // OK, in a defining-type-id
16     T::R f(T::P p) {                // OK, class scope
17         return static_cast<T::R>(p); // OK, type-id of a static_cast
18     }
19     auto g() -> S<T*>::Ptr; // OK, trailing-return-type
20
21     T::SubType t;
22 };
23
24 template <typename T>
25 void f() {
26     void (*pf)(T::X); // Variable pf of type void* initialized with T::X
27     void g(T::X);      // Error: T::X at block scope does not denote a type
28                       // (attempt to declare a void variable)
29 }

```

Nested `inline` namespaces

`inline` keyword is allowed to appear in nested namespace definitions:

```

1  // C++20
2  namespace A::B::inline C{
3      void f(){}
4  }
5  // C++17
6  namespace A::B{
7      inline namespace C{
8          void f(){}
9      }
10 }

```

`using enum`

Scoped enumerations are great, the only problem with them is their verbose usage (e.g. `my_enum::enum_value`). For example, in a switch-statement that checks every possible enum value, `my_enum::` part should be repeated for each case-label. *Using enum declaration* introduces all enumeration's names into the current scope so they are visible as unqualified names and `my_enum::` part can be omitted. It can be applied to unscoped enumerations and even to a single enumerator.

```
1 namespace my_lib {
2     enum class color { red, green, blue };
3     enum COLOR {RED, GREEN, BLUE};
4     enum class side {left, right};
5 }
6
7 void f(my_lib::color c1, my_lib::COLOR c2){
8     using enum my_lib::color;    // introduce scoped enum
9     using enum my_lib::COLOR;    // introduce unscoped enum
10    using my_lib::side::left;    // introduce single enumerator id
11
12    // C++17
13    if(c1 == my_lib::color::red){/*...*/}
14
15    // C++20
16    if(c1 == green){/*...*/}
17    if(c2 == BLUE){/*...*/}
18
19    auto r = my_lib::side::right;    // qualified id is required for `right`
20    auto l = left;                    // but not for `left`
21 }
```

Array size deduction in new-expressions

This fix allows the compiler to deduce array size in new-expressions just like it does for local variables.

```
1 // before C++20
2 int p0[]{1, 2, 3};
3 int* p1 = new int[3]{1, 2, 3}; // explicit size is required
4
5 // since C++20
6 int* p2 = new int[]{1, 2, 3};
7 int* p3 = new int[]{}; // empty
```

```

8 | char* p4 = new char[]{"hi"};
9 | // works with parenthesized initialization of aggregates
10 | int p5[]{1, 2, 3};
11 | int* p6 = new int[]{1, 2, 3};

```

Class template argument deduction for alias templates

CTAD works with type aliases now:

```

1 | template<typename T>
2 | using IntPair = std::pair<int, T>;
3 |
4 | double d{};
5 | IntPair<double> p0{1, d}; // C++17
6 | IntPair p1{1, d}; // std::pair<int, double>
7 | IntPair p2{1, p1}; // std::pair<int, std::pair<int, double>>

```

constexpr

C++ has infamous “static initialization order fiasco” when order of initialization of static storage variables from different translation units is undefined. Variables with zero/constant initialization avoid this problem because they are initialized at compile-time. `constexpr` enforces that variable is initialized at compile-time and unlike `constexpr` it allows non-trivial destructors. Second use-case for `constexpr` is with non-initializing `thread_local` declarations. In such a case, it tells the compiler that the variable is already initialized, otherwise the compiler usually adds code to check and initialize it if required on each usage.

```

1 | struct S {
2 |     constexpr S(int) {}
3 |     ~S(){}; // non-trivial
4 | };
5 |
6 | constexpr S s1{42}; // OK
7 | constexpr S s2{42}; // error because destructor is not trivial
8 |
9 | // tls_definitions.cpp
10 | thread_local constexpr int tls1{1};
11 | thread_local int tls2{2};
12 |
13 | // main.cpp
14 | extern thread_local constexpr int tls1;

```

```

15 extern thread_local int tls2;
16
17 int get_tls1() {
18     return tls1; // pure TLS access
19 }
20
21 int get_tls2() {
22     return tls2; // has implicit TLS initialization code
23 }

```

Signed integers are two's complement

That is, signed integers are now guaranteed to be [two's complement](#). This removes some undefined and implementation-defined behavior because the binary representation is fixed. Overflow for signed integers is still UB but these are well-defined now:

```

1  int i1 = -1;
2  // left-shift for signed negative integers(previously undefined behavior)
3  i1 <<= 1;    // -2
4
5  int i2 = INT_MAX;
6  // "unrepresentable" left-shift for signed integers(previously undefined behavior)
7  i2 <<= 1;    // -2
8
9  int i3 = -1;
10 // right shift for signed negative integers, performs sign-extension(previously implementation-defined)
11 // implementation-defined)
12 i3 >>= 1;    // -1
13 int i4 = 1;
14 i4 >>= 1;    // 0
15
16 // "unrepresentable" conversions to signed integers(previously implementation-defined)
17 int i5 = UINT_MAX; // -1

```

`__VA_OPT__` for variadic macros

Allows more simple handlining of variadic macros. Expands to nothing if `__VA_ARGS__` is empty and to its content otherwise. It's especially useful when macro calls a function with some predefined argument(s) followed by optional `__VA_ARGS__`. In such a case,

`__VA_OPT__` allows to omit the trailing comma when `__VA_ARGS__` are empty (thanks to Jérôme Marsaguet for bringing up this issue).

```
1  #define LOG1(...) \
2      __VA_OPT__(std::printf(__VA_ARGS__);) \
3      std::printf("\n");
4
5  LOG1(); // std::printf("\n");
6  LOG1("number is %d", 12); // std::printf("number is %d", 12); std::prin
7
8  #define LOG2(msg, ...) \
9      std::printf("[ " __FILE__ " :%d] " msg, __LINE__, __VA_ARGS__)
10 #define LOG3(msg, ...) \
11     std::printf("[ " __FILE__ " :%d] " msg, __LINE__ __VA_OPT__(,) __VA_ARGS__)
12
13 // OK, std::printf("[ " file.cpp " :%d] " "%d errors.\n", 14, 0);
14 LOG2("%d errors\n", 0);
15
16 // Error, std::printf("[ " file.cpp " :%d] " "No errors\n", 17, );
17 LOG2("No errors\n");
18
19 // OK, std::printf("[ " file.cpp " :%d] " "No errors\n", 20);
20 LOG3("No errors\n");
```

Explicitly defaulted functions with different exception specifications

This fix allows exception specification of an explicitly defaulted function to differ from such specification of implicitly declared function. Until C++20 such declarations made the program ill-formed. Now it's allowed and, of course, the provided exception specification is the actual one. This is useful when you want to enforce `noexcept`-ness of some operations. For example, due to strong exception guarantee, `std::vector` *moves* its elements into a new storage only if their move constructors are `noexcept`, otherwise elements are *copied*. Sometimes it's desirable to allow this faster implementation even if elements can actually throw during move. As usual, when a function marked `noexcept` throws, `std::terminate()` is called.

```
1  struct S1{
2      // ill-formed until C++20 because implicit constructor is noexcept(true)
3      S1(S1&&)noexcept(false) = default; // can throw
```

```

4   };
5
6   struct S2{
7       S2(S2&&) noexcept = default;
8       // implicitly generated move constructor would be `noexcept(false)`
9       // because of `s1`, now it's enforced to be `noexcept(true)`
10      S1 s1;
11  };
12
13  static_assert(std::is_nothrow_move_constructible_v<S1> == false);
14  static_assert(std::is_nothrow_move_constructible_v<S2> == true);
15
16  struct X1{
17      X1(X1&&) noexcept = default;
18      std::map<int, int> m;    // `std::map(std::map&&)` can throw
19  };
20
21  struct X2{
22      // same as implicitly generated, it's `noexcept(false)` because of `std::map`
23      X2(X2&&) = default;
24      std::map<int, int> m;    // `std::map(std::map&&)` can throw
25  };
26
27  std::vector<X1> v1;
28  std::vector<X2> v2;
29  // ... at some point, `push_back()` needs to reallocate storage
30
31  // efficiently uses `X1(X1&&)` to move the elements to a new storage,
32  // calls `std::terminate()` if it throws
33  v1.push_back(X1{});
34
35  // uses `X2(const X2&)` , thus, copies, not moves elements to a new storage
36  v2.push_back(X2{});

```

Destroying `operator delete`

C++20 introduces a class-specific `operator delete()` that takes a special `std::destroying_delete_t` tag. In such a case, the compiler will not call the object's destructor before calling `operator delete()`, it should be called manually. This might be useful if object members should be used to extract information needed to free memory it occupies, for example to extract its valid size and call sized version of `delete`.

```

1 struct TrickyObject{
2     void operator delete(TrickyObject *ptr, std::destroying_delete_t){
3         // without destroying_delete_t object would have been destroyed here
4         const std::size_t realSize = ptr->GetRealSizeSomehow();
5         // now we need to call the destructor by-hand
6         ptr->~TrickyObject();
7         // and free storage it occupies
8         ::operator delete(ptr, realSize);
9     }
10    // ...
11 };

```

Conditionally `explicit` constructors

Just like `noexcept(bool)` we now have `explicit(bool)` to make constructor/conversion conditionally `explicit`.

```

1 template<typename T>
2 struct S{
3     explicit(!std::is_convertible_v<T, int>) S(T){}
4 };
5
6 void f(){
7     S<char> sc = 'x';           // OK
8     S<std::string> ss1 = "x";   // Error, constructor is explicit
9     S<std::string> ss2{"x"};    // OK
10 }

```

Feature-test macros

C++20 defines a set of preprocessor macros for testing various language and library features, the full list is [here](#).

```

1 #ifdef __has_cpp_attribute // check __has_cpp_attribute itself before using
2 #   if __has_cpp_attribute(no_unique_address) >= 201803L
3 #       define CXX20_NO_UNIQUE_ADDR [[no_unique_address]]
4 #   endif
5 #endif
6

```



```

7  #ifndef CXX20_NO_UNIQUE_ADDR
8  #   define CXX20_NO_UNIQUE_ADDR
9  #endif
10
11  template<typename T>
12  class Widget{
13      int x;
14      CXX20_NO_UNIQUE_ADDR T obj;
15  };

```

Known-to-unknown bound array conversions

Allows conversion from array of known bound to the reference to array of unknown bound. Overload resolution rules have also been updated so that overload with matching size is better than overload with unknown or non-matching size.

```

1  void f(int (&&[]){};
2  void f(int (&)[1]){};
3
4  void g() {
5      int arr[1];
6
7      f(arr);          // calls `f(int (&)[1])`
8      f({1, 2});       // calls `f(int (&&[])`
9      int(&r)[] = arr;
10 }

```

Implicit move for more local objects and rvalue references

In certain cases the compiler is allowed to replace copy with move. But it turned out that rules were too restrictive. C++17 didn't allow to move rvalue references in `return` statements, function parameters in `throw` expressions, and various forms of conversions unreasonably prevented moving. C++20 fixed these issues but some problems are still here, see [P2266R0 Simpler implicit move](#).

```

1  std::unique_ptr<T> f0(std::unique_ptr<T> && ptr) {
2      return ptr; // copied in C++17(thus, error), moved in C++20, OK
3  }

```

```
4
5 std::string f1(std::string && x) {
6     return x;    // copied in C++17, moved in C++20
7 }
8
9 struct Widget{};
10
11 void f2(Widget w){
12     throw w;    // copied in C++17, moved in C++20
13 }
14
15 struct From {
16     From(Widget const &);
17     From(Widget&&);
18 };
19
20 struct To {
21     operator Widget() const &
22     operator Widget() &&
23 };
24
25 From f3() {
26     Widget w;
27     return w;    // moved (no NRVO because of different types)
28 }
29
30 Widget f4() {
31     To t;
32     return t;    // copied in C++17 (conversions were not considered), moved in
33 }
34
35 struct A{
36     A(const Widget&);
37     A(Widget&&);
38 };
39
40 struct B{
41     B(Widget);
42 };
43
44 A f5() {
45     Widget w;
46     return w;    // moved
```

```

47 }
48
49 B f6() {
50     Widget w;
51     return w; // copied in C++17(because there's no B(Widget&&)), moved in
52 }
53
54 struct Derived : Widget{};
55
56 std::shared_ptr<Widget> f7() {
57     std::shared_ptr<Derived> result;
58     return result; // moved
59 }
60
61 Widget f8() {
62     Derived result;
63     // copied in C++17(because there's no Base(Derived)), moved in C++20
64     return result;
65 }

```

Conversion from `T*` to `bool` is narrowing

Conversions from pointer or pointer-to-member types to `bool` are narrowing now and can't be used in places where such conversions are not allowed. `nullptr` is OK when used with direct initialization.

```

1 struct S{
2     int i;
3     bool b;
4 };
5
6 void f(){
7     void* p;
8     S s{1, p};           // error
9     bool b1{p};          // error
10    bool b2 = p;          // OK
11    bool b3{nullptr};     // OK
12    bool b4 = nullptr;    // error
13    bool b5 = {nullptr};  // error
14    if(p){/*...*/}        // OK
15 }

```

Deprecate some uses of `volatile`

Deprecates `volatile` in various contexts:

- built-in prefix/postfix increment/decrement operators on volatile-qualified variables
- usage of the result of an assignment to volatile-qualified object
- built-in compound assignments in form of `E1 op= E2` (e.g. `a += b`) when E1 is volatile-qualified
- volatile-qualified return/parameter type
- volatile-qualified structured binding declarations

Note that `volatile-qualified` means top-level qualification, not just any `volatile` in a type. Something like `volatile int* px` is actually pointer-to-volatile-int, thus, not volatile-qualified.

```
1 volatile int x{};
2 x++;           // deprecated
3 int y = x = 1; // deprecated
4 x = 1;         // OK
5 y = x;         // OK
6 x += 2;        // deprecated
7
8 volatile int   //deprecated
9 f(volatile int); //deprecated
```

Deprecate comma operator in subscripts

Comma operator inside subscripts is deprecated to allow a multidimensional (variadic) subscript operator [in the future](#). Current approach for this is to have a custom `path_type` with overloaded `path_type::operator,()` and `operator[](path_type)`. Variadic `operator[]` will eliminate the need for such dirty tricks.

```
1 // current approach
2 struct SPath{
3     SPath(int);
4     SPath operator,(const SPath&); // store path somehow
5 };
6
7 struct S1{
8     int operator[](SPath); // use path
9 };
```

```

10
11 S1 s1;
12 auto x1 = s1[1,2,3];    // deprecated
13 auto x2 = s1[(1,2,3)];  // OK
14
15 // future approach
16 struct S2{
17     int operator[](int, int, int);
18     // or, as a variadic template
19     template<typename... IndexType>
20     int operator[](IndexType...);
21 };
22
23 S2 s2;
24 auto x3 = s2[1,2,3];

```

Fixes

Here I put minor fixes. Some of them have been implemented by compilers for a while but were not reflected in the Standard. Perhaps, you won't notice any major changes in practice.

Initializer list constructors in class template argument deduction

```

1 // C++17
2 std::tuple t{std::tuple{1, 2}};    // std::tuple<int, int>
3 std::vector v{std::vector{1,2,3}}; // std::vector<std::vector<int>>

```

In this example, two syntactically similar initializations result in surprisingly different CTAD-deduced types. That's because `std::vector` has and prefers `std::initializer_list` constructor, `std::tuple` doesn't have one so it prefers copy constructor.

With this fix, copy constructor is preferred to list constructor when initializing from a single element whose type is a specialization or a child of specialization of the class template under construction.

```

1 // C++20
2 std::tuple t{std::tuple{1, 2}};    // std::tuple<int, int>
3 std::vector v{std::vector{1,2,3}}; // std::vector<int>
4
5 // this example is from "C++17" book by N. Josuttis, section 9.1.1
6 // now it has consistent behavior across compilers

```

```

7 | template<typename... Args>
8 | auto make_vector(const Args&... elems)
9 | {
10 |     return std::vector{elems...};
11 | }
12 |
13 | auto v2 = make_vector(std::vector{1,2,3}); // std::vector<int>

```

const&-qualified pointers to members

The problem was that using `.*` with rvalue with reference qualified pointer to member function was not allowed. Now it's fine.

```

1 | struct S {
2 |     void f() const& {}
3 | };
4 |
5 | S{}.f(); // OK
6 | (S{}.*&S::f)(); // could be an error on some old compilers

```

Simplifying implicit lambda capture

This simplifies wording for lambda capture. Lambdas within default member initializers now officially can have capture list, their enclosing scope is the class scope:

```

1 | struct S{
2 |     int x{1};
3 |     int y{[&]{ return x + 1; }()}; // OK, captures 'this'
4 | };

```

Entities are implicitly captured even within discarded statements and `typeid`:

```

1 | template<bool B>
2 | void f1() {
3 |     std::unique_ptr<int> p;
4 |     [=]() {
5 |         if constexpr (B) {
6 |             (void)p; // always captures p
7 |         }
8 |     }();
9 | }

```

```

10  f1<false>();    // error, can't capture unique_ptr by-value
11
12  void f2() {
13      std::unique_ptr<int> p;
14      [=]() {
15          typeid(p); // error, can't capture unique_ptr by-value
16      }();
17  }
18
19  void f3() {
20      std::unique_ptr<int> p;
21      [=]() {
22          sizeof(p); // OK, unevaluated operand
23      }();
24  }

```

const mismatch with defaulted copy constructor

This fix allows type to have defaulted copy constructor that takes its argument by `const` reference even if some of its members or base classes has copy constructor that takes its argument by non-`const` reference until that constructor is actually needed:

```

1  struct NonConstCopyable{
2      NonConstCopyable() = default;
3      NonConstCopyable(NonConstCopyable&){} // takes by non-const reference
4      NonConstCopyable(NonConstCopyable&&){}
5  };
6
7  // std::tuple(const std::tuple& other) = default; // takes by const reference
8
9  void f(){
10     std::tuple<NonConstCopyable> t; // error in C++17, OK in C++20
11     auto t2 = t; // always an error
12     auto t3 = std::move(t); // OK, move-ctor is used
13 }

```

Access checking on specializations

Allows usage of `protected/private` type to be used as template arguments for partial specialization, explicit specialization and explicit instantiation.

```

1  template<typename T>
2  void f(){}
3
4  template<typename T>
5  struct Trait{};
6
7  class C{
8      class Impl; // private
9  };
10
11  template<>
12  struct Trait<C::Impl>{};    // OK
13
14  template struct Trait<C::Impl>; // OK
15
16  class C2{
17      template<typename T>
18      struct Impl;    // private
19  };
20
21  template<typename T>
22  struct Trait<C2::Impl<T>>;    // OK

```

ADL and function templates that are not visible

Unqualified-id that is followed by a `<` and for which name lookup finds nothing or finds a function is treated as a template-name in order to potentially cause argument dependent lookup to be performed.

```

1  int h;
2  void g();
3
4  namespace N {
5      struct A {};
6      template<class T> int f(T);
7      template<class T> int g(T);
8      template<class T> int h(T);
9  }
10
11  // OK: lookup of `f` finds nothing, `f` treated as a template name
12  auto a = f<N::A>(N::A{});
13  // OK: lookup of `g` finds a function, `g` treated as a template name

```



```

14 auto b = g<N::A>(N::A{});
15 // error: `h` is a variable, not a template function
16 auto c = h<N::A>(N::A{});
17 // OK, `N::h` is qualified-id
18 auto d = N::h<N::A>(N::A{});

```

In rare cases, this can break existing code if there's `operator<()` for functions but it was considered as a pathological case by committee:

```

1 struct A {};
2 bool operator <(void (*fp)(), A);
3 void f(){}
4 int main() {
5     A a;
6     f < a;      // OK until C++20, now error
7     (f) < a;    // OK
8 }

```

Specify when `constexpr` function definitions are needed for constant evaluation

This fix specifies when `constexpr` functions are instantiated. These rules are pretty tricky but most of the time everything works as expected. Instead of copy-pasting them here I will only show a couple of examples to demonstrate the problem.

```

1 struct duration {
2     constexpr duration() {}
3     constexpr operator int() const { return 0; }
4 };
5
6 // duration d = duration(); // #1
7 int n = sizeof(short{duration(duration())}); // always OK since C++20

```

Remember that special member functions are defined only when they are *used*. In C++17 terms move constructor is not used and not defined here so the program should be ill-formed. But, if line `#1` would be uncommented, move constructor would become *used* and defined so the program would be OK. It makes no sense and rules have been changed to reflect this.

Another example:

```

1  template<typename T> constexpr int f() { return T::value; }
2
3  template<bool B, typename T> void g(decltype(B ? f<T>() : 0));
4  template<bool B, typename T> void g(...);
5
6  template<bool B, typename T> void h(decltype(int{B ? f<T>() : 0}));
7  template<bool B, typename T> void h(...);
8
9  void x() {
10     g<false, int>(0); // OK
11     h<false, int>(0); // error
12 }

```

Here we have `constexpr` template function that will potentially be instantiated with type `int` and should lead to an error because `int::value` is wrong. Then there are two functions that use `B ? f<int>() : 0` where `B` is always `false` so `f<int>()` is never needed. The question is: should `f<int>` be instantiated here?

New rules clarify what's *needed for constant evaluation*, template variables or functions in such expressions are always instantiated even if they are not required to evaluate an expression. One of such cases is braced initializer list, thus, in expression `int{B ? f<T>() : 0}` `f<T>` is always instantiated which leads to an error.

Implicit creation of objects for low-level object manipulation

In C++17 an object can be created by a definition, by a new-expression or by changing the active member of a `union`. Now, consider this example:

```

1  struct X { int a, b; };
2  X *make_x() {
3      X* p = (X*)malloc(sizeof(struct X));
4      p->a = 1; // UB in C++17, OK in C++20
5      return p;
6  }

```

Although it looks natural, in C++17 this code has undefined behavior because `X` is not created according to the language rules and write to a member of a nonexistent entity is UB. Rules for such cases have been clarified by specifying what types can be created implicitly and what operations can create such objects implicitly. Types that can be created implicitly (implicit-lifetime types):

- scalar types

- aggregate types
- class types with any eligible trivial constructor and trivial destructor

Operations that can create implicit-lifetime objects implicitly:

- operations that begin the lifetime of an array of `char`, `unsigned char`, `std::byte`
- `operator new` and `operator new[]`
- `std::allocator<T>::allocate(std::size_t n)`
- C library allocation functions: `aligned_alloc`, `calloc`, `malloc`, and `realloc`
- `memcpy` and `memmove`
- `std::bit_cast`

Also, the rule for pseudo-destructor (destructor for built-in types) has been changed. Until C++20 it has no effect, now it ends object's lifetime:

```
1 | int f(){
2 |     using T = int;
3 |     T n{1};
4 |     n.~T();    // no effect in C++17, ends n's lifetime in C++20
5 |     return n;  // OK in C++17, UB in C++20, n is dead now
6 | }
```

You can find more detailed explanation in this post: [Objects, their lifetimes and pointers](#) by Dawid Pilarski.

References

[C++20 feature list](#)

[Complete and grouped list of all papers for each feature](#)

[C++ Weekly](#)



[CppCon 2019: Jonathan Müller “Using C++20’s Three-way Comparison `<=>`”](#)

[CppCon 2019: Timur Doumler “C++20: The small things”](#)

[C++ standard draft](#)

Oleksandr Koval's blog

Oleksandr Koval
oleksandr.koval.dev@gmail.com

 [oleksandrkv](#)
 [oleksandr-koval](#)

Thoughts, experiments, software, C++

