C++20

Overview

Many of these descriptions and examples come from various resources (see Acknowledgements section), summarized in my own words.

C++20 includes the following new language features: - concepts - designated initializers - template syntax for lambdas - range-based for loop with initializer - likely and unlikely attributes - deprecate implicit capture of this - class types in non-type template parameters - constexpr virtual functions - explicit(bool) - char8_t - immediate functions - using enum

C++20 includes the following new library features: - concepts library - synchronized buffered outputstream - std::span - bit operations - math constants - std::is_constant_evaluated

C++20 Language Features

Concepts

Concepts are named compile-time predicates which constrain types. They take the following form:

```
template < template-parameter-list >
concept concept-name = constraint-expression;
```

where `constraint-expression` evaluates to a constexpr Boolean. *Constraints* should model semantic requirements, such as whether a type is a numeric or hashable. A compiler error results if a given type does not satisfy the concept it's bound by (i.e. `constraint-expression` returns `false`). Because constraints are evaluated at compile-time, they can provide more meaningful error messages and runtime safety.

```cpp
// `T` is not limited by any constraints.
template <typename T>
concept always_satisfied = true;
// Limit `T` to integrals.
template <typename T>
concept integral = std::is_integral_v<T>;
// Limit `T` to both the `integral` constraint and signedness.
template <typename T>
concept signed_integral = integral<T> && std::is_signed_v<T>;
// Limit `T` to both the `integral` constraint and the negation of the `signed_integral` co
template <typename T>
concept unsigned_integral = integral<T> && !signed_integral<T>;
```

There are a variety of syntactic forms for enforcing concepts:

```cpp
// Forms for function parameters:
// `T` is a constrained type template parameter.
template <my_concept T>
void f(T v);

// `T` is a constrained type template parameter.
template <typename T>
  requires my_concept<T>
void f(T v);

// `T` is a constrained type template parameter.
template <typename T>
void f(T v) requires my_concept<T>;

// `v` is a constrained deduced parameter.
void f(my_concept auto v);

// `v` is a constrained non-type template parameter.
template <my_concept auto v>
void g();

// Forms for auto-deduced variables:
// `foo` is a constrained auto-deduced value.
my_concept auto foo = ...;

// Forms for lambdas:
// `T` is a constrained type template parameter.
auto f = []<my_concept T> (T v) {
  // ...
};
// `T` is a constrained type template parameter.
auto f = []<typename T> requires my_concept<T> (T v) {
  // ...
};
// `T` is a constrained type template parameter.
auto f = []<typename T> (T v) requires my_concept<T> {
  // ...
};
// `v` is a constrained deduced parameter.
auto f = [](my_concept auto v) {
  // ...
};
// `v` is a constrained non-type template parameter.
auto g = []<my_concept auto v> () {
  // ...
};
```

The `requires` keyword is used either to start a requires clause or a requires expression:

```
template <typename T>
  requires my_concept<T> // `requires` clause.
void f(T);

template <typename T>
concept callable = requires (T f) { f(); }; // `requires` expression.

template <typename T>
  requires requires (T x) { x + x; } // `requires` clause and expression on same line.
T add(T a, T b) {
  return a + b;
}
```

Note that the parameter list in a requires expression is optional. Each requirement in a requires expression are one of the following:

- **Simple requirements** - asserts that the given expression is valid.

```
template <typename T>
concept callable = requires (T f) { f(); };
```

- **Type requirements** - denoted by the `typename` keyword followed by a type name, asserts that the given type name is valid.

```
struct foo {
  int foo;
};

struct bar {
  using value = int;
  value data;
};

struct baz {
  using value = int;
  value data;
};

// Using SFINAE, enable if `T` is a `baz`.
template <typename T, typename = std::enable_if_t<std::is_same_v<T, baz>>>
struct S {};

template <typename T>
using Ref = T&;

template <typename T>
```

```
concept C = requires {
                      // Requirements on type `T`:
  typename T::value; // A) has an inner member named `value`
  typename S<T>;     // B) must have a valid class template specialization for `S`
  typename Ref<T>;   // C) must be a valid alias template substitution
};

template <C T>
void g(T a);

g(foo{}); // ERROR: Fails requirement A.
g(bar{}); // ERROR: Fails requirement B.
g(baz{}); // PASS.
```

- **Compound requirements** - an expression in braces followed by a trailing return type or type constraint.

```
template <typename T>
concept C = requires(T x) {
  {*x} -> typename T::inner; // the type of the expression `*x` is convertible to `T::inner
  {x + 1} -> std::same_as<int>; // the expression `x + 1` satisfies `std::same_as<decltype(
  {x * 1} -> T; // the type of the expression `x * 1` is convertible to `T`
};
```

- **Nested requirements** - denoted by the `requires` keyword, specify additional constraints (such as those on local parameter arguments).

```
template <typename T>
concept C = requires(T x) {
  requires std::same_as<sizeof(x), size_t>;
};
```

See also: concepts library.

Designated initializers

C-style designated initializer syntax. Any member fields that are not explicitly listed in the designated initializer list are default-initialized.

```
struct A {
  int x;
  int y;
  int z = 123;
};

A a {.x = 1, .z = 2}; // a.x == 1, a.y == 0, a.z == 2
```

Template syntax for lambdas

Use familiar template syntax in lambda expressions.

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

Range-based for loop with initializer

This feature simplifies common code patterns, helps keep scopes tight, and offers an elegant solution to a common lifetime problem.

```
for (auto v = std::vector{1, 2, 3}; auto& e : v) {
  std::cout << e;
}
// prints "123"
```

likely and unlikely attributes

Provides a hint to the optimizer that the labelled statement is likely/unlikely to have its body executed.

```
int random = get_random_number_between_x_and_y(0, 3);
[[likely]] if (random > 0) {
  // body of if statement
  // ...
}

[[unlikely]] while (unlikely_truthy_condition) {
  // body of while statement
  // ...
}
```

Deprecate implicit capture of this

Implicitly capturing `this` in a lamdba capture using `[=]` is now deprecated; prefer capturing explicitly using `[=, this]` or `[=, *this]`.

```
struct int_value {
  int n = 0;
  auto getter_fn() {
    // BAD:
    // return [=]() { return n; };

    // GOOD:
    return [=, *this]() { return n; };
  }
};
```

Class types in non-type template parameters

Classes can now be used in non-type template parameters. Objects passed in as template arguments have the type `const T`, where `T` is the type of the object, and has static storage duration.

```
struct foo {
  foo() = default;
  constexpr foo(int) {}
};

template <foo f>
auto get_foo() {
  return f;
}

get_foo(); // uses implicit constructor
get_foo<foo{123}>();
```

constexpr virtual functions

Virtual functions can now be `constexpr` and evaluated at compile-time. `constexpr` virtual functions can override non-`constexpr` virtual functions and vice-versa.

```
struct X1 {
  virtual int f() const = 0;
};

struct X2: public X1 {
  constexpr virtual int f() const { return 2; }
};

struct X3: public X2 {
  virtual int f() const { return 3; }
};

struct X4: public X3 {
  constexpr virtual int f() const { return 4; }
};

constexpr X4 x4;
x4.f(); // == 4
```

explicit(bool)

Conditionally select at compile-time whether a constructor is made explicit or not. `explicit(true)` is the same as specifying `explicit`.

```
struct foo {
  // Specify non-integral types (strings, floats, etc.) require explicit construction.
  template <typename T>
  explicit(!std::is_integral_v<T>) foo(T) {}
};

foo a = 123; // OK
foo b = "123"; // ERROR: explicit constructor is not a candidate (explicit specifier evalua
foo c {"123"}; // OK
```

char8_t

Provides a standard type for representing UTF-8 strings.

```
char8_t utf8_str[] = u8"\u0123";
```

Immediate functions

Similar to `constexpr` functions, but functions with a `consteval` specifier must produce a constant. These are called `immediate functions`.

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

constexpr int r = sqr(100); // OK
int x = 100;
int r2 = sqr(x); // ERROR: the value of 'x' is not usable in a constant expression
                 // OK if `sqr` were a `constexpr` function
```

using enum

Bring an enum's members into scope to improve readability. Before:

```
enum class rgba_color_channel { red, green, blue, alpha };

std::string_view to_string(rgba_color_channel channel) {
  switch (channel) {
    case rgba_color_channel::red:   return "red";
    case rgba_color_channel::green: return "green";
    case rgba_color_channel::blue:  return "blue";
    case rgba_color_channel::alpha: return "alpha";
  }
}
```

After:

```
enum class rgba_color_channel { red, green, blue, alpha };
```

```cpp
std::string_view to_string(rgba_color_channel my_channel) {
  switch (my_channel) {
    using enum rgba_color_channel;
    case red:   return "red";
    case green: return "green";
    case blue:  return "blue";
    case alpha: return "alpha";
  }
}
```

C++20 Library Features

Concepts library

Concepts are also provided by the standard library for building more complicated concepts. Some of these include:

Core language concepts: - `same_as` - specifies two types are the same. - `derived_from` - specifies that a type is derived from another type. - `convertible_to` - specifies that a type is implicitly convertible to another type. - `common_with` - specifies that two types share a common type. - `integral` - specifies that a type is an integral type. - `default_constructible` - specifies that an object of a type can be default-constructed.

Comparison concepts: - `boolean` - specifies that a type can be used in Boolean contexts. - `equality_comparable` - specifies that `operator==` is an equivalence relation.

Object concepts: - `movable` - specifies that an object of a type can be moved and swapped. - `copyable` - specifies that an object of a type can be copied, moved, and swapped. - `semiregular` - specifies that an object of a type can be copied, moved, swapped, and default constructed. - `regular` - specifies that a type is *regular*, that is, it is both `semiregular` and `equality_comparable`.

Callable concepts: - `invocable` - specifies that a callable type can be invoked with a given set of argument types. - `predicate` - specifies that a callable type is a Boolean predicate.

See also: concepts.

Synchronized buffered outputstream

Buffers output operations for the wrapped output stream ensuring synchronization (i.e. no interleaving of output).

```cpp
std::osyncstream{std::cout} << "The value of x is:" << x << std::endl;
```

std::span

A span is a view (i.e. non-owning) of a container providing bounds-checked access to a contiguous group of elements. Since views do not own their elements they are cheap to construct and copy – a simplified way to think about views is they are holding references to their data. Spans can be dynamically-sized or fixed-sized.

```
void f(std::span<int> ints) {
    std::for_each(ints.begin(), ints.end(), [](auto i) {
        // ...
    });
}

std::vector<int> v = {1, 2, 3};
f(v);
std::array<int, 3> a = {1, 2, 3};
f(a);
// etc.
```

Example: as opposed to maintaining a pointer and length field, a span wraps both of those up in a single container.

```
constexpr size_t LENGTH_ELEMENTS = 3;
int* arr = new int[LENGTH_ELEMENTS]; // arr = {0, 0, 0}

// Fixed-sized span which provides a view of ‘arr‘.
std::span<int, LENGTH_ELEMENTS> span = arr;
span[1] = 1; // arr = {0, 1, 0}

// Dynamic-sized span which provides a view of ‘arr‘.
std::span<int> d_span = arr;
span[0] = 1; // arr = {1, 1, 0}
```

```
constexpr size_t LENGTH_ELEMENTS = 3;
int* arr = new int[LENGTH_ELEMENTS];

std::span<int, LENGTH_ELEMENTS> span = arr; // OK
std::span<double, LENGTH_ELEMENTS> span2 = arr; // ERROR
std::span<int, 1> span3 = arr; // ERROR
```

Bit operations

C++20 provides a new `<bit>` header which provides some bit operations including popcount.

```
std::popcount(0u); // 0
std::popcount(1u); // 1
std::popcount(0b1111'0000u); // 4
```

Math constants

Mathematical constants including PI, Euler's number, etc. defined in the <numbers> header.

```
std::numbers::pi; // 3.14159...
std::numbers::e; // 2.71828...
```

std::is_constant_evaluated

Predicate function which is truthy when it is called in a compile-time context.

```
constexpr bool is_compile_time() {
    return std::is_constant_evaluated();
}

constexpr bool a = is_compile_time(); // true
bool b = is_compile_time(); // false
```

Acknowledgements

- cppreference - especially useful for finding examples and documentation of new library features.
- C++ Rvalue References Explained - a great introduction I used to understand rvalue references, perfect forwarding, and move semantics.
- clang and gcc's standards support pages. Also included here are the proposals for language/library features that I used to help find a description of, what it's meant to fix, and some examples.
- Compiler explorer
- Scott Meyers' Effective Modern C++ - highly recommended book!
- Jason Turner's C++ Weekly - nice collection of C++-related videos.
- What can I do with a moved-from object?
- What are some uses of decltype(auto)?
- And many more SO posts I'm forgetting...

Author

Anthony Calandra

Content Contributors

See: https://github.com/AnthonyCalandra/modern-cpp-features/graphs/contributors