Concepts in C++20

A Brief Introduction to Concepts

CppMaryland

2020-01-08

Eric Richardson

Concepts

Concepts are a way to constrain templates, producing a compile error if instantiated with an invalid type

When used with templated functions, concepts are a middle ground between normal functions with concrete parameter types and unconstrained function templates

Normal C++ functions

```
// Pass an integer
void TestConcrete(int i) { /* ... */ }

// Pass a const reference to a base class
void TestConcrete(const BaseClass& base) { /* ... */ }

// Pass a pointer to a const base class
void TestConcrete(const BaseClass* base) { /* ... */ }
```

- Function arguments are limited to distinct types
- These can be built-in types:

  `int`, `char*`, `void`, etc.
- They can also be user-defined types:

  `BaseClass*`, `DerivedClass&`, etc.

Templated functions

```
// Pass an unconstrained template parameter T
template <typename T>
void TestTemplate(T t) { /* ... */ }

// Template specialization for int
template <>
void TestTemplate<int>(int i) { /* ... */ }

// Template specialization for pointer to a base class
template <>
void TestTemplate<BaseClass*>(BaseClass* base) { /* ... */ }
```

- No type-checking of function arguments (similar to `auto`)
- An attempt to instantiate a template on an illegal type can produce super long compile errors

Constrained with concepts

```
// Same unconstrained function template
template <typename T>
void TestConcepts(T t) { /* ... */ }

// Require that T is an integral type
template <typename T>
        requires Concepts::integral<T>
void TestConcepts(T t) { /* ... */ }

// Require that T is a pointer to something
template <typename T>
        requires IsPointer<T>
void TestConcepts(T t) { /* ... */ }
```

- Concepts make it easier to specify constraints on parameter types

Simple example concept

```
#include <type_traits>

// Define a concept to see if a type is a pointer
template <typename T>
        concept IsPointer =
        std::is_pointer_v<T>;
```

- Try not to define your own concepts
- If you do, they shouldn't be as simple as the ones above (which just uses a single type trait directly)

Multiple ways to specify concepts

Pre-P1084, this was how concepts had to be specified:

```
requires {
  T::value;
  requires Same<decltype(T::value), const typename T::value_ty
};
```

This is a more concise way to specify the same thing:

```
requires {
  { T::value } -> Same<const typename T::value_type&>;
};
```

Definitions

Requires clauses and expression

Requires **clauses** are used to specify constraints:

```
template<typename T> requires Addable<T>
T add(T a, T b) { return a + b; }
```

Requires **expressions** are used to define concepts:

```
template<typename T>
concept Addable = requires (T x) { x + x; };
```

They return bool, and describe the constraints

There is also an "ad-hoc constraint", in which the `requires` clause is used twice:

```cpp
template<typename T>
    requires requires (T x) { x + x; }
T add(T a, T b) { return a + b; }
```

SFINAE

"Substitution Failure Is Not An Error"

Before concepts, it was still possible to constrain templates using SFINAE

However, concepts are easier to read, produce better error messages, and are subsumable

Subsumption

Subsumption is a relationship that defines partial order of constraints, which is used to determine:

- the best viable candidate for a non-template function in overload resolution
- the address of a non-template function in an overload set
- the best match for a template template argument
- partial ordering of class template specializations
- partial ordering of function templates

`random_access_iterator` subsumes `bidirectional_iterator`, which subsumes `forward_iterator`, etc.

Compiler and libraries used

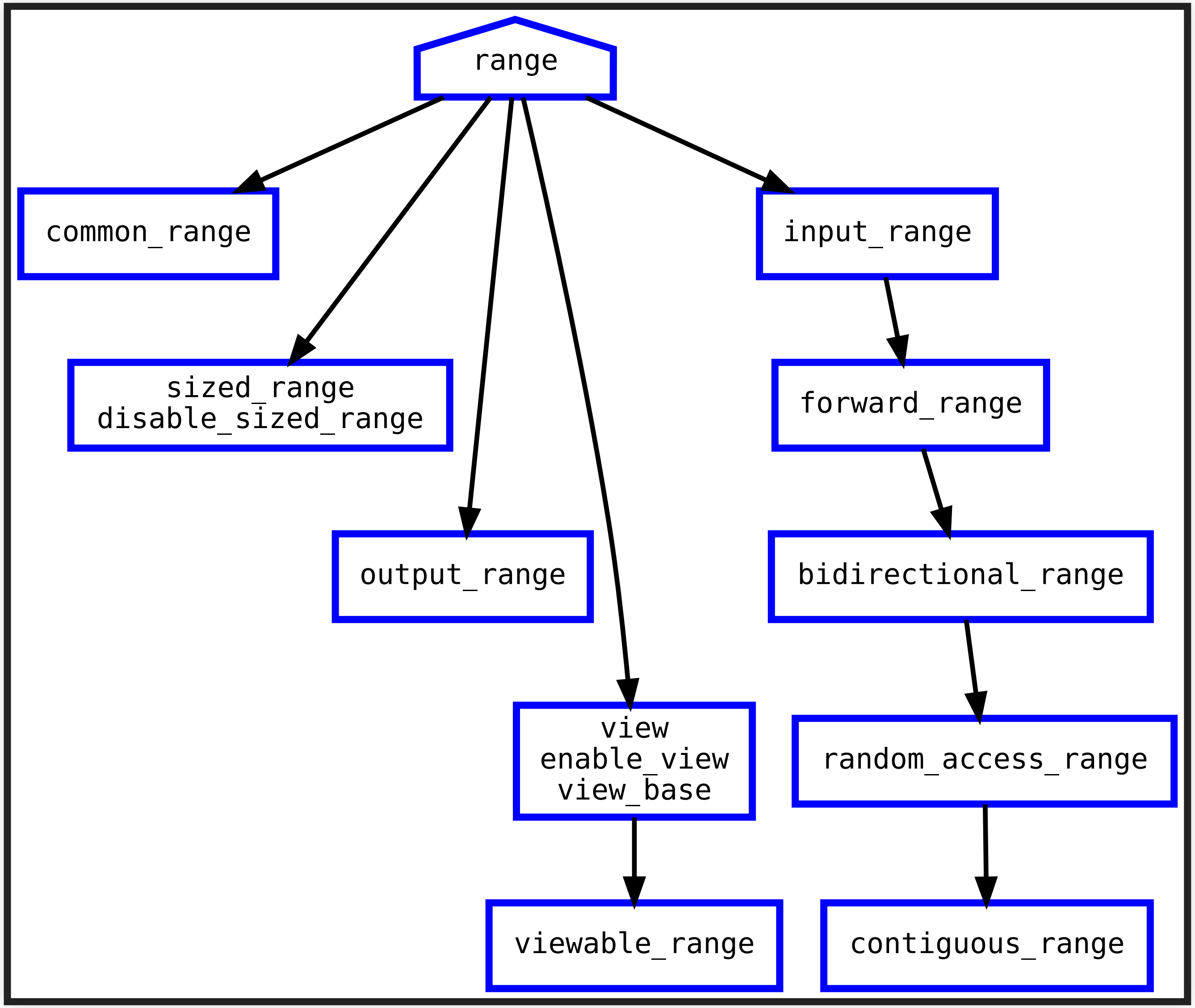
Couldn't use `clang`, because it doesn't yet support the Concepts TS (though there is a fork that does)

I used `g++ 9.2.0`, but you can probably use `g++-6` or later (it just needs to support `-fconcepts`)

No, this isn't truly C++20, but apparently true C++20 Concepts are available in `g++-10`

Available Concepts

Range Concept Hierarchy



Core language concepts

`same_as<T, U>`

```
namespace detail {
        template<class T, class U>
        concept SameHelper =
                std::is_same_v<T, U>
}

template<class T, class U>
concept same_as =
        detail::SameHelper<T, U> &&
        detail::SameHelper<U, T>
```

`true` iff types `T` and `U` are the same

More language concepts

Concept Name	Description
`convertible_to<From, To>`	`true` iff `From` is convertible to `To`, both implicitly and via `static_cast<To>`
`derived_from<Derived, Base>`	`true` iff `Derived` is derived from `Base`, and `Derived*` is convertible to `Base*`
`common_with<T, U>`	`true` iff `T` and `U` share a common type to which both can be converted
`common_reference_with<T, U>`	`true` iff `T` and `U` share a common **reference** type to which both can be converted

Numerical concepts

Concept Name	Description
`integral<T>`	`true` iff `T` is an integral type, e.g. `int` or `size_t`, but not `float`
`signed_integral<T>`	`true` iff `T` is a signed `integral`, e.g. `int`, but not `uint32_t` or `size_t`
`unsigned_integral<T>`	`true` iff `T` is an **unsigned** `integral`, e.g. `uint16_t`, but not `int64_t` or `int`
`floating_point<T>`	`true` iff `T` is a floating point type, e.g. `double` or `float`, but not `int`

Assignment and swap concepts

Concept Name	Description
`assignable_from<LHS, RHS>`	`true` iff an expression of type `RHS` can be assigned to an lvalue of type `LHS`
`swappable<T>`	`true` iff lvalues of type `T` are swappable
`swappable_with<T, U>`	`true` iff expressions of types `T` and `U` are swappable with each other

Constructible/destructible concepts

Concept Name	Description
`destructible<T>`	`true` iff `T` is `nothrow` destructible
`constructible_from<T, ... Args>`	`true` iff `destructible<T>` returns `true`, and `T` can be initialized with the given `Args`
`default_constructible<T>`	`true` iff `constructible_from<T>` returns `true`, without any arguments
`move_constructible<T>`	`true` iff both `constructible_from<T, T>`, and `convertible_to<T, T>` return `true`
`copy_constructible<T>`	`true` iff `move_constructible<T>` is true, and `T` is convertible to `T` (with all sorts of referenceness and constness)

Comparison concepts

Concept Name	Description
`boolean<B>`	`true` iff `B` can be used in boolean contexts
`equality_comparable<T>`	`true` iff `T` can be compared to itself
`equality_comparable_with<T, U>`	`true` iff `T` and `U` can be compared
`totally_ordered<T>`	`true` if expressions of type `T` can be compared using operators `==`, `!=`, `<`, `>`, `<=`, and `>=`
`totally_ordered_with<T, U>`	`true` iff `T` and `U` can be compared using the same 6 operators above

Object concepts

Concept Name	Description
`movable<T>`	`true` iff `T` is an object type that can be moved
`copyable<T>`	`true` iff `T` is a `movable` object type that can also be copied
`semiregular<T>`	`true` iff `T` is `copyable` and `default_constructible`
`regular<T>`	`true` iff `T` is `semiregular` and `equality_comparable`

Callable concepts

Concept Name	Description
`invocable<F, ... Args>`	`true` iff callable type `F` can be called with a set of arguments `Args`
`regular_invocable<F, ... Args>`	`true` iff `F` is `invocable`, and `F` doesn't modify the `Args`
`predicate<F, ... Args>`	`true` iff `F` is `regular_invocable`, and produces a boolean result
`relation<R, T, U>`	`true` iff `R` is a binary predicate, and can be called with the types `T` and `U`
`strict_weak_order<R, T, U>`	`true` iff `relation<R, T, U>` returns `true`, and `R` imposes a strict weak ordering

Code and Presentation

The source code is available at my GitHub page:

https://github.com/ejricha/examples/tree/master/concepts

The presentation is available as well:

https://github.com/ejricha/presentations

Also posted on the CppMaryland GitHub page:

https://github.com/cppmaryland/presentations

References

- wg21.link/p557
- https://en.cppreference.com/w/cpp/header/concepts
- https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-the-usage-of-concepts
- https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-the-usage-of-concepts-2
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-concepts

YouTube Videos

CppCon 2018: Andrew Sutton
Concepts in 60: Everything you need to know and nothing you don't

CppCon 2018: Arthur O'Dwyer
Concepts As She Is Spoke

CppCon 2018: Bjarne Stroustrup
Concepts: The Future of Generic Programming (the future is here)

C++ Concepts and Ranges
Mateusz Pusz - Meeting C++ 2018

Questions?