C++20 Concepts

Hayk Khachatryan

Aug 3, 2022

Concepts

Concepts are named Boolean predicates on template parameters, evaluated at compile time which enable

- constraining automatic and template type deduction
- well-defined parametric polymorphism
- modular, reusable type-checking
- simplified compiler diagnostics, shorter error messages for failed template instantiations
- selecting function template overloads and class template specializations based on type properties
- zero overhead abstraction mechanism using template metaprogramming

usage of Concepts

- default concepts built on top of `type traits`
- constrained algorithms, `ranges`
- iterator concepts `random_access_iterator` replaces `LegacyRandomAccessIterator`
- etc.

problem: unconstrained template parameters

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

int main() {
    add(1, 5.7);
    // oops
    add(std::string("hello "), std::string("world\n"));
    add(true, 7); // maybe unexpected
}
```

problem: unconstrained template parameters (cont'd)

```
template<>
int add<bool, int>(bool a, int b) {
  std::cout << __PRETTY_FUNCTION__ << '\n';
  return static_cast<int>(a) + b;
}
// auto add(T, K) [with T = bool; K = int]
...
add(true, 7);
```

solution using type traits

```
template <typename T, typename K>
auto add(T a, K b) {
    static_assert(!std::is_same_v<T, bool> &&
                  (std::is_integral_v<T>   ||
                   std::is_floating_point_v<T>));
    static_assert(!std::is_same_v<K, bool> &&
                  (std::is_integral_v<K>   ||
                   std::is_floating_point_v<K>));

    return a + b;
}
// template<> auto add(int, bool) = delete;
int main() {
    add(1, 5.7);
    // static assertion fails
    add(8, true);
    add(std::string("hello "), std::string("world\n"));
}
```

solution using concepts

```
template <typename T>
concept Number = !std::same_as<T, bool> &&
                 (std::integral<T>      ||
                 std::floating_point<T>);

template <Number T, Number K>
auto add(T a, K b) {
    return a + b;
}

int main() {
    add(1, 5.7);
    // rejected at compile time
    add(8, true);
    add(std::string("hello "), std::string("world\n"));
}
```

syntax

```
template <Number T, Number K>
auto add(T a, K b)

auto add(Number auto a, Number auto b)

template <typename T, typename K>
    requires Number<T> && Number<K>
auto add(T a, K b)
```

Concepts

```
// can be substituted with any value
template <typename RandomAccessIterator>
...

// type checking is enforced by the compiler
template <std::random_accesss_iterator It>
...
```

Standard Concepts

```
template <class T>
concept Any = true;

Defined in header <concepts>

template <class T>
concept integral = std::is_integral_v<T>;

template <class T>
concept movable =
  std::is_object_v<T> &&
  std::move_constructible<T> &&
  std::assignable_from<T&, T> &&
  std::swappable<T>;
```

Standard Concepts

```
Defined in header <ranges>

template<class T>
concept range = requires(T& t) {
  ranges::begin(t);
  ranges::end(t);
};

template<class T>
concept view = ranges::range<T> && std::movable<T>
            && ranges::enable_view<T>;

void func(std::range auto Range ...)
```

examples

```
ranges.cpp    // demonstration of ranges library
concepts.cpp  // some standard concepts
algorithm.cpp // constrained/unconstrained algo comparison
```

C++2x core guidelines on Concepts

T.10: Specify concepts for all template arguments

```
template<input_iterator Iter, typename Val>
    requires equality_comparable_with<
    iter_value_t<Iter>, Val>
Iter find(Iter b, Iter e, Val v)
{
    // ...
}
```

T.11: Whenever possible use standard concepts

C++2x core guidelines on Concepts

T.12: Prefer concept names over auto for local variables

```
auto& x = v.front();        // bad
String auto& s = v.front(); // good

Sortable auto x;
Comparable auto x;
for (ForwardIterator auto it = l.begin() ...
```

C++2x core guidelines on Concepts

T.21: Require a complete set of operations for a concept

```
    Arithmetic: +, -, *, /, +=, -=, *=, /=
    Comparable: <, >, <=, >=, ==, !=

template<typename T>
concept Subtractable = requires(T a, T b) { a - b; };
// bad

Subtractable auto x;
...
x + y; // maybe error
```

C++2x core guidelines on Concepts

```
template<typename T>
    concept Arithmetical = requires(T a, T b) {
        {a + b} -> std::convertible_to<T>;
        {a - b} -> std::convertible_to<T>;
        {a * b} -> std::convertible_to<T>;
        {a / b} -> std::convertible_to<T>;
        ...
    };
```

C++2x core guidelines on Concepts

T.40: Use function objects to pass operations to algorithms

```
auto res = find_if(v, [](auto x) { return x > 7; });
```

T.48: If your compiler does not support concepts, fake them with enable_if

```
template<typename T>
enable_if_t<is_integral_v<T>>
f(T v)
{
    // ...
}
// Equivalent to:
template<Integral T>
void f(T v)
{
    // ...
}
```