Introduction to C++20's Concepts and Constraints

Robert Zavalczki

March 22, 2019

In this talk

- Motivating Examples
- Concepts Overview
- Concepts in Action
- Advice

Motivating Examples

We Need Better Compiler Diagnostics

```
std::list<int> ns = { 3, 2, 1 };
std::sort(ns.begin(), ns.end());
```

```
在.../c++/8.3.0/algorithm:62中包含的文件中，
                 来自<source>：2：
<source>：7：35：从这里要求
.../c++/8.3.0/bits/stl_algo.h:1969:22：错误：'operator-'不匹配
（操作数类型是'std :: _ List_iterator < int>'和'std :: _ List_iterator
<int>'）
     std :: __ lg（__ last  -  __first）* 2,
               ~~~~~~~ ^ ~~~~~~~~
.../c++/8.3.0/bits/stl_iterator.h:392:5：注意：候选人：'template <class
_IteratorL, class _IteratorR> decltype（（__y.base （） -
__x.base（））） std :: operator-（const std :: reverse_iterator
<_Iterator>&, const std :: reverse_iterator <_IteratorR>&）'
     operator-（const reverse_iterator <_IteratorL>&__ x,
     ^ ~~~~~~~
.../c++/8.3.0/bits/stl_iterator.h:392:5：注意：模板参数扣除/替换失败：
编译返回：1 You Fail
```

Compiler says in 30+ lines that std::sort can't be called:

```
examples.cpp:30:35:   required from here
.../stl_algo.h:1969:22: error: no match for
  ‘operator-’ (operand types are
  ‘std::_List_iterator<int>’ and
  ‘std::_List_iterator<int>’)
    std::__lg(__last - __first) * 2,
              ~~~~~~~~~~~~~~~~~
```

We Need Better Compiler Diagnostics

Example 1

```cpp
std::vector<int&> v;
```

Example 2

```cpp
struct IntBox {
    int value;
};

std::vector<IntBox> v = { {2}, {1} };
std::sort(v.begin(), v.end());
```

How to Write Overload Sets Easily?

A library writer wants to write an overload set:

```
// general implementation
template<typename Iterator>
void doWork(Iterator first,
            Iterator last);

// More efficient implementation for
// random access iterators?
template<typename RandomAccessIterator>
void doWork(RandomAccessIterator first,
            RandomAccessIterator last);
```

How to Provide Semantic Clues to Users?

```
template<typename T>
T badlyNamedFunction(
    T withBadlyNamedArgument);
```

- the programmer has no clue what T is supposed to be and how it should behave
- "**typename** T" is the "**void**∗" of template meta programming

Constrained Genericity in Java or C#

```
class ListWidget<
        T extends Widget & Comparable<T>> {

    // here the type parameter 'T' is
    // constrained to be a 'Widget'
    // implementing the 'Comparable'
    // interface
}
```

Typeclasses in Haskell or Scala, Rust's Traits

Example in Haskell

```haskell
-- if the type parameter 'a' belongs to the
-- 'Num' typeclass then the function 'sum'
-- is defined

sum :: (Num a) => a -> a -> a
sum x y = x + y
```

Concepts Overview

What is a Concept?

"A concept is an abstract or general idea inferred or derived from specific instances." (WordNet 3.0, 2006)

What is a Concept in C++?

A C++ concept is a named, compile time, predicate that defines syntactic and semantic requirements on type parameters.

Example:

```
// the concept 'Same' is satisfied if the
// type parameters `T` and `U` are the same
// type. Then it evaluates to `true`.

template <typename T, typename U>
concept Same = std::is_same_v<T, U>;
```

What are Constraints?

- A concept is a named set of requirements, but it is defined using *constraints*.
- A *constraint* is a sequence of logical operations and operands that specify requirements on template arguments.
- A constraint can be an atomic constraint or be built up using conjunctions (&&) or disjunctions (||) from other constraints.

Example

```
EqualityComparable<T> &&
    std::is_convertible_v<T, bool>
```

in this case T is the constrained entity.

Constraints

Constraints appear in the definition of a concept or within a *requires clause*:

```
// constraint on a function template type
// parameter
template<typename T>
    requires std::is_arithmetic_v<T>
T sum(T x, T y) {
    return x + y;
}
```

- Constraints are an elegant alternative to *std::enable_if*.
- the *requires* keyword in this context introduces a *requires clause*

Anatomy of a Concept

Syntactically a concept is expressed as a constraint on template arguments:

```
// a creature is an organism that has
// the methods 'eat' and 'breed'
template <typename T>
concept LivingCreature =
   MulticellularOrganism<T>
                  && requires(T a) {
     a.eat();
     a.breed();
}
```

Using a Concept

```
// these 4 forms are equivalent
template <typename T>
        requires LivingCreature<T>
void feed(T& creature);

template <LivingCreature T>
void feed(T& creature); // terse notation

template <typename T>
void feed(T& creature)
        requires LivingCreature<T>;

void feed(auto LivingCreature& creature);
```

Other Uses - Constraining *auto*

```
// deduced and costrained return type
Number auto f(); // not a function template

// variable type
Number auto x = f();
```

Concept Definition Example 1

A named concept can be defined using a primary value of type *bool*:

```
// Integrals are an algebraic ring of
// enumerable, ordered, numbers, etc.
template <class T>
concept Integral =
        std::is_integral<T>::value;

// same as above, using C++17
template <class T>
concept Integral = std::is_integral_v<T>;
```

Concept Definition Example 2

A named concept can be defined using a *requires* expression:

```
// a type `T` is EqualityComparable if the
// operator '==' can be used with that type
// simplified, for exposition only
template <class T>
concept EqualityComparable =
    requires(T t) {
        t == t -> bool;
        t != t -> bool;
    };
```

Complex Concept Definition Example 3

A named concept can be defined by using other concepts and constraints

```
template <class T> // from Ranges TS
concept Semiregular =
    DefaultConstructible<T> &&
    CopyConstructible<T> &&
    Destructible<T> &&
    CopyAssignable<T> &&
    requires(T a, size_t n) {
        requires Same<T*, decltype(&a)>;
        { a.~T() } noexcept;
        // ...
    };
```

Concepts in Action

Improved Compiler Diagnostics

If using concepts, the compiler diagnostic message for the previous *std::list* sort example could be:

```
error: cannot call std::sort with
      std::_List_iterator<int>
note:  concept RandomAccessIterator<
      std::_List_iterator<int>> was not satisfied
```

Semantic Clues to Library Users

```
template<Number T>
T someFunction(T x);
```

- Here, the developer of someFunction provided precious information about the type T: it is a Number.

Concepts in Overload Resolution

Concepts can help in the task to select the most appropriate function overload and template specialization.

The constraints representing the concepts form a partially ordered set that can be used by the compiler to derive the best candidate in overload resolution.

Overload Resolution Example

Consider the problem of travel: moving from one point to another.

Almost all *'types'* can *'crawl'*, but a few *'types'* can also *'fly'*.

How can we implement a generic ' travel ' overload set that takes advantage of flying?

A Zoo of Creatures

```
template <typename T>
concept Creature = requires(T a) {
    a.crawl();
};

template <typename T>
concept AirborneCreature =
    Creature<T> &&
    requires(T a) {
        a.fly();
    };
```

The Overload Set

```
// generic and slow implementation
template <Creature T>
void travel(T&& t) {
    t.crawl();
}

// optimized implementation for flying
// creatures
template <AirborneCreature T>
void travel(T&& t) {
    t.fly();
}
```

A Zoo of Creatures

```
struct Reptile {
    void crawl() {}
};

struct Bird {
    void crawl() {}
    void fly() {}
};

struct Spaceship {
    void fly() {}
    void teleport() {
        puts("instant teleportation");
    }
};
```

A Zoo of Creatures

```
int main()
{
    travel(Reptile()); // Reptile::crawl
    travel(Bird()); // OK, Bird::fly
    travel(Spaceship()); // Error:
        // constraint not satisfied
        // Spaceship doesn't crawl
        // type deduction is constrained by
        // using concepts
}
```

Advice

Advice on C++ Concepts

- The intent of C++ concepts is to model semantic categories like *Same*, *Number*, *Function*, *Iterator* etc. rather than syntactic restrictions like *Addable* or *HasPlus*.
- Creating good concepts is not easy. Better use well established concept libraries such as the standard library, the *Ranges* library etc.
- Beware: concepts are part of the interface of a template where they are used as a constraint.