Concepts

Presented by
Rich Seibel

3 Apr 2020

Concepts

- What are Concepts
  - Definition
  - Origin
  - Goals
  - How to use
  - How to define
- Design of Concepts
  - Issues
  - Ideals
  - Constraints
- Musings by Bjarne Stroustrupt

Section 1

What are Concepts

Definition

- Concepts are requirements of a template on its template arguments

  In other words: A way at compile time (no runtime overhead) to specify requirements for a parameter or an argument to be valid for a function template or class template and its member functions

  - Applies to templates
  - Compile time only

Origin of Templates

- In 1987, Bjarne Stroustrup wanted concepts, but was unable to define them
  - Thus: bad error messages!
- Bjarne Stroustrup's goals for templates were:
  1. Generality
  2. Zero overhead
  3. Well-defined interfaces
- Templates acheived the first two successfully, but failed the third
- **Concepts** are a solution to the third
- Alex Stepanov named them **Concepts**

Origin of Concepts

- Interest was strong for getting them into the 2008 standard, but despite the efforts of several people no traction was gained
- In 2009, Bjarne Stroustrup met with Gabriel Dos Reis and Andrew Sutton to design concepts from scratch
- The effort continued into 2010 and was called C++0x, since they didn't know when it might be finished
- In 2011, this effort was declared failed and C++11 was released without concepts
- Also in 2011, Alex Stepanov called together a larger group to restart the standards effort that resulted in Concepts-TS in 2015
- This effort did not make the cut for C++17
- The **concepts** in C++20 appear to be a revision, based on experience, of the 2015 effort

How would **Concepts** solve the third goal

- From Bjarne Stroustrup's writings (P0557r1):
  - Consider this function declaration from C++84

    ```
    double sqrt(double d);
    ```

  - If I write:

    ```
    double d = 7;
    double d2 = sqrt(d);
    ```

  - Everything compiles without any problem
  - However, if I write:

    ```
    vector<string> vs = { "Good", "old", "template" };
    double d3 = sqrt(vs);
    ```

  - I get an error with a message telling me that "vs is not a double", or something similar

How would **Concepts** solve the third goal (cont)

- Now consider this same scenario using templates (generic code)
  - this template from C++98

```
template <class T>
void sort(T& c)
{
  // implementation
}
```

  - if I write

```
sort(vs);
```

  - everything compiles without problem
  - however if I write

```
sort(d);
```

  - I get an error, possibly dozens of lines and maybe ”d doesn’t have a [ ] operator” if I’m lucky
  - or worse, pages of messages resulting in a message that I cannot relate to anything I wrote
  - what we want is an error message similar to the sqrt message

    ”d is not sortable”
  - possibly with additional information

    ”d does not have a random iterator”

    ”d references a type that does not have a $<$ operator”

How would **Concepts** solve the third goal (cont)

- To get this result, we can use **concepts**
  - we declare:

    ```
    void sort(Sortable& c);
    ```

  - what does **Sortable** really imply
    1. a type that has begin() and end()
    2. random access iterators
    3. referenced elements have < comparisons

How do I use **Concepts**

- A **concept** is a compile time predicate (that is, something that yields a boolean value)
- For example, a template argument T may need to be
  - an iterator: `iterator<T>`
  - random access: `Random_access_iterator<T>`
  - a number: `Number<T>`
- The notation is C<T>
  - C is a **concept**
  - T is a type
  - C<T> is either **true** or **false**
  - Concepts can involve more than one argument and can be combined in logical expressions

How do I use **Concepts** (cont)

- Three forms of using a **concept**, for example Sequence, are allowed
  - from most prefered to least prefered

```
template<Sequence Seq>
void algo(Seq& s);

template<typename Seq>
  requires Sequence<Seq>
void algo(Seq& s);

template<typename Seq>
void algo(Seq& s) requires Sequence<Seq>;
```

How do I use **Concepts** (cont)

- **Concept**s with more then one argument
  - consider **find** in the STL algorithm library

    ```
    template<class InputIterator, class T>
    InputIterator find(InputIterator first, InputIterator last, const T& value);
    ```

  - assume we want to simplify and make more understandable and safer by constraining the arguments

    ```
    template<Sequence S, typename T>
      requires Equality_compatable<Value_type<S>, T>
    Iterator_of<S> find(S& seq, const T& value);
    ```

  - we are going to use the STL **find** internally so:
    * the Sequence **concept** is going to require that S have **begin()** and **end()**, which return InputIterators
    * the Equality_compatable **concept** requires the values in S can be compared using the == operator with a value of type T
  - but what about Value_type and Iterator_of?
    * they are a bit of alias magic from C++11

      ```
      template<typename X> using Value_type<X> = X::value_type;
      template<typename X> using Iterator_of<X> = X::iterator;
      ```

  - Equality_compatable **concept** comes from the new **concepts** library in C++20
- An alternative declaration of find could be

```
template<typename S, typename T>
  requires Sequence<S> && Equality_compatable<Value_type<S>, T>
Iterator_of<S> find(S& seq, const T& value);
```

Defining **Concepts**

- A **concept** is a named set of requirements
- It's definition must appear at namespace scope
- Form is:

```
template<parameter_list>
concept name = constraint_expression
```

- A **constraint** is a sequence of logical operations that specify requirements on template arguments
  - anything producing a boolean value at compile time can be a constraint
  - constraints usually appear in require clauses

    ```
    requires constraint
    ```

  - for example, the Equality_comparable **concept** seen previously, could be defined as

    ```
    template<typename T>
    concept Equality_comparable =
      requires (T a, T b) {
        { a == b } -> bool;
        { a != b } -> bool;
      };
    ```

Defining **Concepts** (cont)

– another example, Sequence could be defined as

```
template<typename T>
concept Sequence = requires(T t) {
  typename Value_type<T>
  typename Iterator_of<T>

  { begin(t) } -> Iterator_of<T>;
  { end(t) } -> Iterator_of<T>;

  requires Input_iterator<Iterator_of<T>>;
  requires Same_type<Value_type<T>, Value_type<Iterator_of<T>>>;
};
```

– and, Sortable could be defined as

```
template<typename T>
concept Sortable =
  Sequence<T> &&
  Random_access_iterator<Iterator_of<T>> &&
  Less_than_comparable<Value_type<T>>;
```

Section 2

Design of Concepts

Accidental match

- Consider two classes, Shape and Cowboy, that have the same method, `draw()`
- A `draw_all(v)` function calls `draw()` on each memeber of a vector
- Did we really want draw_all to work on values of both classes?
- Adding concepts to limit applicablity may be used

Semantics

- While we cannot specify semantics with concepts, we can use our understanding of semantics to design a good **concept**, that is, taking into account the semnantics of a domain to find the set of properties we can use to create a **concept** to match the domain
- Most application areas already have this semantic understanding
  - built-in types: integers, floats, ...
  - STL concepts: iterators, containers
  - mathematical concepts: moniods, group, ...
  - graph concepts: edges, verticies

Ideals of **concept** design

- What makes one **concept** better than another?
  - we want to write algorithms that can be used for a variety of types
  - we want types that can be used with a variety of containers
- For example,
  - we write a Number **concept** that can be used with our numeric algorithms
  - we write containers that can hold types that match our Number **concept**
- How do we define our Number **concept**
  - we allow types that can perform numeric operations
  - we reject types that use the same operators to perform non-numeric operations
    * For example, `opeator+`
  - is our type copyable and/or movable
  - does our type do + and =, but not +=
- Thus the ideal is not "minimal requirements", but "requirements expressed in terms of fundmental and complete concepts"

Constraints

- What if we can't get an "ideal" **concept**?
- Some concepts are "incomplete"
- Some concepts are "simple"
- **concept**s that are to simple for general use and/or lack a clear semantics can be used as building blocks for more complete concepts
- **constraints** are useful even if incomplete **concept**s
  - as they help where applicable
  - do not hinder where inapplicable (failure mode is as before concepts)
  - they allow gradual implementation in real world situations

Testing a **Concept**

- All previous discussion of **concept**s has been associated with either a template class or template function
- **Concept**s can be tested directly
- Assume we have defined a **concept** Number

```
static_assert(Number<int>);          // should pass
static_assert(Number<string>);       // should fail
```

Section 3

Beyond Concepts

Readability

- Concepts make template declarations easier to read, hence, more understandable and less prone to incorrect usage
- Short-hand notation

```
void sort(Sortable& s);

// Short for
template<Sortable Seq>
void sort(Seq& s);

// Short for
template<typename Seq>
  requires Sortable<Seq>
void sort(Seq& s);
```

- However, the first of these did not make it into C++20

Readability (cont)

- **auto** can be thought of as the least constrained **concept**

  ```
  void f(auto x);              // take argument of any type
  ```

  or using concepts

  ```
  concept Any = true;
  void f(Any x);               // take argument of Any type is more readable
  ```

  but, there is a small difference

  ```
  void ff(auto x, auto y);  // x and y can be different
  void ff(Any x, Any y);    // x and y must be the same type
  ```

  or, a big difference

  ```
  auto x = hh(2);           // x can be anything
  Number x = hh(2);         // more readable and constrained
  ```

- However, the last of these did not make it into C++20

Additional Musings by Bjarne Stroustrupt

- Do we really need Concepts

  ”Concepts” is a fundamental feature that in an ideal world would have been present in the very first version of templates and the basis for all to use

- Definition Checking

  Concepts are not checked on template definitions

  1. We didn’t want to delay and complicate the initial design
  2. We estimate that something like 90% of the benefis of concepts are in the value of improved specification and point-of-use checking
  3. The template implementer can conpensate through normal testing techniques
  4. As ever, type errors are *always* caught, only uncomfortably late
  5. By checking definitions, we would complicate transition from older, unconstrained code to concept-based templates
  6. By checking definitions, we would be unable to insert debug aids, logging code, telementry code, performance counters, and other ”scaffolding code” into a template without affecting its interface

- Separate compliation of templates

  – Complete separate compliation presupposes definition checking and would lead to lots of indirection killing performance

  – Alternative is semi-compiled form as in a ”module” system

Additional Musings by Bjarne Stroustrupt (cont)

- Multiple notation
  - Define a **concept** either as a template variable or template function
    - Note: The template function form is not in the current reference implementation and probably not needed since logicals have been added
- Can you spot the template

```
void sort(Sortable&);
```

  - No syntatic clue that this is a template
  - Experience shows that it really isn't a problem
- Should concepts be a kind of class
  - Concepts are not classes, they do not support inheritance
    - Though you can enforce type inheritance, that is not considered proper C++

Additional Musings by Bjarne Stroustrupt (cont)

- Concepts are not type classes
  - Reference to Haskell type classes
    - Two different ways to solve similar problems
  - A **concept** is a compile-time predicate on zero or more template arguments or value arguments
    - Operational requirements for concepts are specified in terms of usage patterns (”value expressions”)
    - Concepts are specified as general predicates (Boolean expressions)
    - A type does not need to be explicitly defined to match a **concept**; the match is deduced
    - Concepts can take value arguents (rather than just type arguments)
    - Concetps can take many arguments
    - Concept functions can be overloaded
    - Algorithms can be overloaded on concepts
    - Concepts do not default constrain implementations
    - Concepts are not defined as members of hierarchies; relations among concepts are deduced
    - Concepts can constrain template arguments
- Conclusion
  - Concepts complete C++ templates as originally envisioned. I (Bjarne Stroustrup) don’t see them as extension but as a completion.
  - Concepts follow C++ design principles
    - Provide good interfaces
    - Look for semantic coherence
    - Don’t force the user to do what a machine does better
    - Keep simple things simple
    - Zero-overhead

Some closing thoughts

- Expect **concept** defined templates to be in future standards

  ```
  void sort(Sortable&);
  ```

- Expect complex rules on construction of concepts to improve (see cppreference for C++20 rules for concepts)
- More Type Classes influence
  - Concepts do not have relationships in C++ as they do in Haskell
    * Possible, but Bjarne may oppose (tried and failed)
    * Expect some borrowing
- Packages based on Units types (weigths, lengths, ...)

  ```
  velocity c = Velocity::speed_of_light;
  mass m = 1 pound;
  killowathours e = m * c**2;
  ```

  or

  ```
  #include<Enstein>

  Energy Enstein::masstoenergy(Mass m);
  ```

  where Energy and Mass are **concept**s
- C++20 concepts are not the end of the concept story