A Tour of the Concepts Wording

Author: Douglas Gregor, Indiana University
Document number: N2399=07-0259
Date: September 9, 2007
Project: Programming Language C++, Core Working Group
Reply-to: Douglas Gregor <doug.gregor@gmail.com>

This paper gives a tour of the concepts wording, N2398=07-0258. Readers unfamiliar with concepts are encouraged to read the introductory and tutorial material in the concepts proposal (N2081=06-0151). Each page is split into two parts, with example source code followed by a discussion of the compilation of that source code and how it relates to the specification of concepts in N2398. Certain important points in the source code are marked with circles in numbers, e.g., ①, which correspond to items of explanation on the right-hand side. The explanations refer to specific sections and paragraphs in N2398. I recommend that one have a copy of N2398 at hand when browsing this paper.

```
auto concept LessThanComparable<typename T> { ①
  bool operator<(T, T); ②
}

template<typename T>
requires LessThanComparable<T> ③
const T& min(const T& x, const T& y) { ④
  return x < y? x : y; ⑤ ⑬
}

struct X { int member; };

concept_map LessThanComparable<X> ⑥ {
  bool operator<(const X& x1, const X& x2) { ⑦
    return x1.member < x2.member; ⑧
  }
} ⑨

concept_map LessThanComparable<int> { } ⑩

concept_map LessThanComparable<int X::*> { } ⑪

void f(X x1, X x2, int i1, int i2, float f1, float f2) {
  min(x1, x2); ⑫
  min(i1, i2); ⑭
  min(f1, f2); ⑮
  min(&X::member, &X::member); ⑯
}
```

① This is the definition of a new concept LessThanComparable ([concept.def]p1). It is an implicit concept ([concept.implicit]).

② This is an associated function ([concept.fct]), stating that LessThanComparable requires a < operator that accepts two values of type T and returns a **bool**.

③ This requirements clause ([temp.req]) contains a single *concept-id* requirement ([temp.req]p3) that states that T must meet the requirements of the LessThanComparable concept. The presence of the requirements clause means that min is a *constrained template* ([temp.constrained]).

④ The compiler synthesizes an archetype T′ for the template type parameter T ([temp.archetype]p1).

⑤ To type-check the expression x < y, we use the non-dependent archetype T′ in lieu of T ([temp.archetype]p2). Thus, we need to find a suitable **operator**<. We search for an **operator**< in the requirements scope ([basic.scope.req]), and we find it in the *concept instance* LessThanComparable<T′> that is generated from the requirements clause ([temp.archetype]p14). This provides the following declaration, used to type-check x < y:

```
bool operator<(T′ const&, T′ const&);
```

```
auto concept LessThanComparable<typename T> { ①
  bool operator<(T, T); ②
}

template<typename T>
requires LessThanComparable<T> ③
const T& min(const T& x, const T& y) { ④
  return x < y? x : y; ⑤ ⑬
}

struct X { int member; };

concept_map LessThanComparable<X> ⑥ {
  bool operator<(const X& x1, const X& x2) { ⑦
    return x1.member < x2.member; ⑧
  }
} ⑨

concept_map LessThanComparable<int> { } ⑩

concept_map LessThanComparable<int X::*> { } ⑪

void f(X x1, X x2, int i1, int i2, float f1, float f2) {
  min(x1, x2); ⑫
  min(i1, i2); ⑭
  min(f1, f2); ⑮
  min(&X::member, &X::member); ⑯
}
```

⑥ This line defines a concept map LessThanComparable<X> ([concept.map]), which states that (and how) the type X meets the requirements of the LessThanComparable concept.

⑦ This is an associated function definition ([concept.map.fct]), which provides an implementation for the **operator**< in the LessThanComparable concept when applied to X. This definition satisfies the requirement for **operator**< in LessThanComparable ([concept.map.fct]p1). Note that the associated function definition passes by reference-to-const, while the concept definition does not: the signatures of these two **operator**<s still match because all arguments to associated function definitions are passed by reference ([concept.map.fct]p3).

⑧ When a function template uses the LessThanComparable concept to compare two values of type X, it will use this implementation, comparing the member fields. This does *not* expose a global **operator**< for type X.

⑨ At this point, the compiler verifies that all of the requirements of concept LessThanComparable have been met ([concept.map]p3) and checks that there are no declarations in the concept map that have not been used to meet a requirement in the concept ([concept.map]p5).

```
auto concept LessThanComparable<typename T> { ①
  bool operator<(T, T); ②
}

template<typename T>
requires LessThanComparable<T> ③
const T& min(const T& x, const T& y) { ④
  return x < y? x : y; ⑤ ⑬
}

struct X { int member; };

concept_map LessThanComparable<X> ⑥ {
  bool operator<(const X& x1, const X& x2) { ⑦
    return x1.member < x2.member; ⑧
  }
} ⑨

concept_map LessThanComparable<int> { } ⑩

concept_map LessThanComparable<int X::*> { } ⑪

void f(X x1, X x2, int i1, int i2, float f1, float f2) {
  min(x1, x2); ⑫
  min(i1, i2); ⑭
  min(f1, f2); ⑮
  min(&X::member, &X::member); ⑯
}
```

⑩ This concept map does not explicitly define an associated function to match the requirement for **operator**<. Therefore, the following associated function definition is implicitly defined ([concept.map.implicit]):

```
bool operator<(int const & x, int const& y) {
  return x < y;
}
```

The associated function definition's signature and function body are created based on the associated function (from the LessThanComparable) concept, substituting in the concept arguments (**int**) and using the < operator for the implementation ([concept.implicit]p3).

⑪ Like ⑩, the compiler will synthesize the following associated function definition to meet the requirement for **operator**< ([concept.implicit]p3):

```
bool operator<(int X::* const& x, int X::* const& y) {
  return x < y;
}
```

However, in this case the expression x < y is ill-formed. Therefore, the concept map LessThanComparable<**int** X::*> is ill-formed.

```
auto concept LessThanComparable<typename T> { ①
  bool operator<(T, T); ②
}

template<typename T>
requires LessThanComparable<T> ③
const T& min(const T& x, const T& y) { ④
  return x < y? x : y; ⑤ ⑬
}

struct X { int member; };

concept_map LessThanComparable<X> ⑥ {
  bool operator<(const X& x1, const X& x2) { ⑦
    return x1.member < x2.member; ⑧
  }
} ⑨

concept_map LessThanComparable<int> { } ⑩

concept_map LessThanComparable<int X::*> { } ⑪

void f(X x1, X x2, int i1, int i2, float f1, float f2) {
  min(x1, x2); ⑫
  min(i1, i2); ⑭
  min(f1, f2); ⑮
  min(&X::member, &X::member); ⑯
}
```

⑫ With the call to `min`, template argument deduction ([temp.deduct]) proceeds as normal, determining that the template type parameter `T` is `X`. Once template argument deduction is complete, we check that all of the template's requirements are satisfied ([temp.deduct]p2, last bullet). The *concept-id* requirement `LessThanComparable<T>` (where `T` is `X`) can be satisfied by a concept map ([temp.req]p3). The concept map defined at ⑥ satisfies this requirement, so the call to `min` is well-formed.

⑬ When instantiating the constrained function template `min<X>`, function calls that resolve to members of a concept instance will instantiate to use the corresponding concept map definition ([temp.constrained.inst]p2). Thus, when instantiating the expression `x < y`, we use `LessThanComparable<X>::`**`operator`**`<`, so that `min<X>` compares the `X` values based on their `member` fields.

⑭ Template argument deduction determines that `T` is **`int`**, and the `LessThanComparable` requirement is satisfied by the concept map at ⑩. The instantiation of `min<`**`int`**`>` uses the implicitly-defined `LessThanComparable<`**`int`**`>::`**`operator`**`<`, and the compiler should optimize this use of < to a simple use of the integer < ([concept.implicit]p4).

```
auto concept LessThanComparable<typename T> { ①
  bool operator<(T, T); ②
}

template<typename T>
requires LessThanComparable<T> ③
const T& min(const T& x, const T& y) { ④
  return x < y? x : y; ⑤ ⑬
}

struct X { int member; };

concept_map LessThanComparable<X> ⑥ {
  bool operator<(const X& x1, const X& x2) { ⑦
    return x1.member < x2.member; ⑧
  }
} ⑨

concept_map LessThanComparable<int> { } ⑩

concept_map LessThanComparable<int X::*> { } ⑪

void f(X x1, X x2, int i1, int i2, float f1, float f2) {
  min(x1, x2); ⑫
  min(i1, i2); ⑭
  min(f1, f2); ⑮
  min(&X::member, &X::member); ⑯
}
```

⑮ There is no concept map `LessThanComparable<float>`. However, `LessThanComparable` is an implicit concept ([concept.implicit]), so the compiler will attempt to implicitly generate a concept map to satisfy `min`'s requirements. To generate this concept map, the compiler will effectively generate and attempt to type-check the `operator<` in the concept map ([concept.implicit]p3), e.g.,

```
bool operator<(float const& x, float const& y) {
  return x < y;
}
```

Since the expression `x < y` is well-formed, all of the concept's requirements are satisfied, and the compiler completes the definition of the concept map `LessThanComparable<float>`. This definition is used to satisfy the requirements of `min`.

⑯ The compiler attempts to implicitly define the concept map `LessThanComparable<int X::*>` (we assume the ill-formed concept map at ⑪ does not exist). However, this implicit definition fails for the same reason that the code at ⑪ is ill-formed. Therefore, the `LessThanComparable` requirement is not satisfied, and `min<int X::*>` does not enter the overload set. Thus, there is no `min` function that can be called here.

```
concept SignedIntegral<typename T> {
  T::T(const T&); ①
}
concept_map SignedIntegral<std::ptrdiff_t> { } ②
concept InputIterator<typename Iter> {
  SignedIntegral difference_type; ③
  Iter& operator++(Iter&); ④
}
concept RandomAccessIterator<typename Iter>
   : InputIterator<Iter> ⑤ {
  difference_type⑥ operator-(Iter, Iter);
}
template<typename T>
  concept_map RandomAccessIterator<T*> ⑦ {
    typedef std::ptrdiff_t difference_type; ⑧
  } ⑨
template<InputIterator Iter> ⑩
  Iter::difference_type⑪ distance(Iter first, Iter last);
template<RandomAccessIterator Iter>
  Iter::difference_type distance(Iter first, Iter last); ⑫
void f(int* first, int* last) {
  distance(first, last); ⑬
} ⑭
```

① This is an associated member function ([concept.fct]p5), which states that T must have a copy constructor.

② The compiler verifies that all of the requirements of concept SignedIntegral have been met by the concept map ([concept.map]p3). The checks that the type std::ptrdiff_t does in fact have an accessible, non-deleted copy constructor ([concept.map.fct]p6).

③ This declares an associated type ([concept.assoc]p1) named difference_type. This declaration uses the "simple form" of requirements to both declare the associated type and place a concept requirement on it ([concept.assoc]p4), and is therefore equivalent to

```
typename difference_type;
requires SignedIntegral<difference_type>;
```

The **requires** line provides an associated requirement ([concept.req]) that describes the requirements on difference_type.

④ This associated function declares a requirement for a prefix increment operator. Note that we write the operator as a free function, despite the fact that outside of concepts **operator**++ cannot be written as a free function ([concept.fct]p4).

```
concept SignedIntegral<typename T> {
  T::T(const T&); ①
}
concept_map SignedIntegral<std::ptrdiff_t> { } ②

concept InputIterator<typename Iter> {
  SignedIntegral difference_type; ③
  Iter& operator++(Iter&); ④
}
concept RandomAccessIterator<typename Iter>
   : InputIterator<Iter> ⑤ {
  difference_type⑥ operator-(Iter, Iter);
}
template<typename T>
  concept_map RandomAccessIterator<T*> ⑦ {
    typedef std::ptrdiff_t difference_type; ⑧
  } ⑨

template<InputIterator Iter> ⑩
  Iter::difference_type⑪ distance(Iter first, Iter last);
template<RandomAccessIterator Iter>
  Iter::difference_type distance(Iter first, Iter last); ⑫
void f(int* first, int* last) {
  distance(first, last); ⑬
} ⑭
```

⑤ The RandomAccessIterator concept is a refinement of the InputIterator concept ([concept.refinement]). Thus, every type that meets the requirements of RandomAccessIterator also meets the requirements of InputIterator.

⑥ The difference_type type is found in the refined concept InputIterator<Iter> ([concept.member.lookup]p4). It's fully-qualified name is InputIterator<Iter>::difference_type.

⑦ This is a concept map template ([temp.concept.map]), which can be instantiated to produce concept maps.

⑧ This associated type definition ([concept.map.assoc]p2) satisfies the requirement for an associated type difference_type. The compiler verifies that the associated requirements of the concept are met ([concept.map]p6). In this case, the compiler satisfies the requirement for SignedIntegral<difference_type> with the concept map ②.

⑨ The compiler verifies that all of the requirements of concept RandomAccessIterator have been met by the concept map ([concept.map]p3). This involves the implicit definition of **operator-** (for RandomAccessIterator) and **operator++** (for InputIterator). This concept map definition implicitly defines a concept map template InputIterator<T*> ([concept.implicit.maps]).

```
concept SignedIntegral<typename T> {
  T::T(const T&); ①
}
concept_map SignedIntegral<std::ptrdiff_t> { } ②
concept InputIterator<typename Iter> {
  SignedIntegral difference_type; ③
  Iter& operator++(Iter&); ④
}
concept RandomAccessIterator<typename Iter>
   : InputIterator<Iter> ⑤ {
  difference_type⑥ operator-(Iter, Iter);
}
template<typename T>
  concept_map RandomAccessIterator<T*> ⑦ {
    typedef std::ptrdiff_t difference_type; ⑧
  } ⑨
template<InputIterator Iter> ⑩
  Iter::difference_type⑪ distance(Iter first, Iter last);
template<RandomAccessIterator Iter>
  Iter::difference_type distance(Iter first, Iter last); ⑫
void f(int* first, int* last) {
  distance(first, last); ⑬
} ⑭
```

⑩ This template header declares a template type parameter Iter and the requirement InputIterator<Iter> using the "simple form" of concept requirements ([temp.param]p18). It is equivalent to:

```
template<typename Iter> requires InputIterator<Iter> // ...
```

⑪ Name lookup into a template type parameter looks into the requirements clause for an associated type ([basic.lookup.qual]). Thus, Iter::difference_type resolves to InputIterator<Iter>::difference_type.

⑫ This function template is an overload of the previous distance function template. It is distinct from the previous declaration because the requirements clause is part of the signature of a function template ([defs.signature]).

⑬ This call to distance eventually resolves to the function template marked ⑫. With that in mind, let us begin. Name lookup finds both overloads of distance. Template argument deduction of the first distance determines that Iter is bound to **int***. Template argument deduction requires that the template arguments meet the requirements of the first function template, i.e., we need a concept map InputIterator<**int***> ([temp.deduct]p2, last bullet).

⑬ *(continued from previous page)*
There is no such concept map, but we do find the concept map template InputIterator<T∗> that was implicitly defined ([concept.implicit.maps]) by the concept map template RandomAccessIterator<T∗>. We apply concept map matching ([temp.concept.map]p4) to determine that this concept map template does, in fact, work. Thus, the requirements of the first distance overload are met and this function template specicalization enters into the set of candidate functions.

We follow a similar pattern with the second overload of distance. Again template argument deduction determines that Iter is bound to **int**∗. This time, we satisfy the requirement for RandomAccessIterator<**int**∗> with the concept map template, and this function template specialization enters into the set of candidate functions.

Partial ordering of function templates ([temp.func.order]) determines which of the two function templates is more specialized. First, we determine whether the first distance (call it T_1) is at least as specialized as the second distance (call it T_2). We use the archetype of T_1's Iter, Iter′, as the synthesized type to produce the transformed template T_1 ([temp.func.order]p3). Then, template argument deduction determines that Iter is bound to Iter′, and then we determine whether the requirement RandomAccessIterator<Iter′> is satisfied ([temp.deduct]p2, last bullet). Since there is no concept map or concept instance to satisfy this requirement, T_1 is not as specialized as T_2.

Now, the other direction. We use the archetype Iter″ of Iter from T_2 as the unique type for the transformed template of the second distance. We also synthesize concept maps for each requirement in T_2, using the archetype Iter″ ([temp.func.order]p3). Thus, we have the concept maps RandomAccessIterator<Iter″> and (through refinement) InputIterator<Iter″>. Template argument deduction determines that Iter is bound to the archetype Iter″, and the compiler checks the requirements of T_1 (the first distance). The requirement for InputIterator<Iter″> is satisfied by the synthesized concept map. Therefore, T_2 is at least as specialized as T_1 and, since T_1 is not as specialized as T_2, T_1 is more specialized than T_2. Hence, function template partial ordering selects the more-specific RandomAccessIterator version of distance.

⑭ The compiler, exhausted after a long day of partial ordering, heads to the bar. So should you.