Considering Concept Constraint Combinators

Authors: Douglas Gregor, Indiana University
Andrew Lumsdaine, Indiana University
Document number: N2161=07-0021
Date: 2007-01-12
Project: Programming Language C++, Evolution Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

1 Introduction

The concepts proposal (N2081) contains several ways of combining different constraints within a where clause. One can use "and" constraints (both concept requirements must be met), "not" constraints (the concept requirement must not be met), and "or" constraints (either one, but not both, of the concept requirements must be met). Now that we have significant experience using and implementing concepts, this document analyzes the uses of the "not" and "or" constraints, and provides concrete recommendations for changes to the concept proposal.

2 "Not" Constraints

"Not" constraints are used very rarely, for two reasons. The first reason is that "not" constraints convey very little actual information. A statement such as "I am not a crook" says very little, merely excluding the speaker from having a single, specific property. From this statement we cannot conclude, for instance, that the speaker is a fine, upstanding citizen. In a more technical sense, while regular constraints such as LessThanComparable<T> imply that certain operations and capabilities are available (in this case, we have a less-than operator that provides an ordering on elements of type T), the corresponding "not" constraint (!LessThanComparable<T>) only says that those specific capabilities are not guaranteed to be provided. T might still have a less-than operator with different semantics or might only be lacking a suitable concept map. Thus, "not" constraints provide very little information for users or for the compiler's type checker.

The second reason "not" constraints are rarely used is that their most obvious usage tends to be better expressed with another mechanism: concept-based overloading. "Not" constraints often come up when the user wants to write "if the given type meets the requirements of concept X, do this; if it does not meet the requirements of concept X, do this other thing." The user might then write two overloads to cover those cases:

```
template<CopyConstructible T> where X<T> void foo(T); // #1a
template<CopyConstructible T> where !X<T> void foo(T); // #2a
```

Here, the use of the "not" requirement is redundant, because one could have written a similar, simpler overload set with essentially the same properties (#1b will be chosen if the requirements of X are met, #2b will be chosen otherwise):

```
template<CopyConstructible T> where X<T> void foo(T); // #1b
template<CopyConstructible T> void foo(T); // #2b
```

When are "not" constraints necessary? Theoretically speaking, they are never necessary, because one can always use overloading (or class template partial specialization) to express the same notion. For example, say we have a concept Small that indicates that a type should be stack-allocated when possible (because it's small!). Then, we might write up two versions of a function wibble(), one that uses stack allocation and one that uses heap allocation:

```
template<typename T> where Small<T> void wibble(const T&); // #3
template<typename T> where HeapAllocatable<T> void wibble(const T&); // #4
```

When T is Small, we just allocate it on the stack in #3. Otherwise, we need T to be HeapAllocatable so that we can allocate it on the heap in #4. This overload set works properly when the type instantiating T is either Small or HeapAllocatable, but not both. If the type meets the requirements of both concepts, we get an ambiguity when we try to call wibble() because neither #3 nor #4 is more specialized than the other.

To solve this problem, we could introduce a new wibble() that has both constraints:

```
template<typename T> where Small<T> && HeapAllocatable<T>
  void wibble(const T&); // #5
```

This "tie-breaker" function would implement the appropriate semantics for this case (e.g., the same semantics as #3). In the worst case, making an overload set unambiguous could require an exponential number of tie-breaker functions, making this approach infeasible for real-world use.

Using "not" constraints provides an alternative solution. Instead of introducing tie-breaker functions, one can introduce a "not" constraint into one of the functions to exclude that function when another function is known to be a better match. For instance, if we want to prefer the Small overload (#3) for small, heap-allocatable types, we would replace #4 with:

```
template<typename T> where HeapAllocatable<T> && !Small<T>
  void wibble(const T&); // #4b
```

This use of "not" constraints can be emulated, at the cost of a bit more template metaprogramming, e.g.,

```
template<typename T> struct not_small { static const bool value = true; };
template<Small T> struct not_small<T> { static const bool value = false; };

template<typename T>
  where HeapAllocatable<T> && std::True<not_small<T>::value>
  void wibble(const T&); // #4c
```

In the specification of the C++ Standard Library, we have found that "not" constraints can save quite a bit of work in certain cases. The most interesting case is the unique_copy() algorithm, which requires several "not" constraints to ensure that its rather unique overloading requirements are satisfied:

```
template<InputIterator InIter, class OutIter>
  where OutputIterator<OutIter, InIter::value_type> && EqualityComparable<InIter::value_type> &&
        Assignable<InIter::value_type> && CopyConstructible<InIter::value_type> &&
        !ForwardIterator<InIter> && !MutableForwardIterator<OutIter>
  OutIter unique_copy(InIter first, InIter last, OutIter result);

template<ForwardIterator InIter, class OutIter>
  where OutputIterator<OutIter, InIter::value_type> &&
        EqualityComparable<InIter::reference>
  OutIter unique_copy(InIter first, InIter last, OutIter result);
```

```
template<InputIterator InIter, MutableForwardIterator OutIter>
  where EqualityComparable<OutIter::reference, InIter::value_type> &&
        Assignable<OutIter::reference, InIter::reference> &&
        !ForwardIterator<InIter>
  OutIter unique_copy(InIter first, InIter last, OutIter result);
```

While "not" constraints are not going to be used very commonly, when they are used they save a tremendous amount of effort and redundant code. In addition, "not" constraints are trivial to implement and specify, so they have a very low cost of inclusion.

3 "Or" Constraints

"Or" constraints allow a constrained template to accept types that meet one of two different requirements. For example, the following template will work with either Integral or Floating types:

```
template<typename T>
  where Integral<T> || Floating<T>
  T negate(T x) { return -x; }
```

One can think of "or" constraints as implicitly representing a set of template definitions with different constraints. For example, the negate() function template above can be viewed as two separate function templates:

```
template<typename T> where Integral<T> T negate(T x) { return -x; }
template<typename T> where Floating<T> T negate(T x) { return -x; }
```

If negate() is called with a type that is Integral, the first definition will be selected; if the type is Floating, the second definition will be selected. The two function bodies are independent, primarily because the - operator binds to different concepts and therefore could have different implementations, e.g., Integral<T>::**operator**- for the first template could have a different implementation from Floating<T>::**operator**- in the second template. For this reason, when the type T is both Integral and Floating, we get an error.

This formulation leads to the most simple and direct implementation of "or" constraints. In the implementation, the compiler will essentially produce a template with several sets of requirements, then parse the definition of that template several times, one for each set of requirements. In the negate() example, we need to process the definition of the template twice, once for Integral and once for Floating. In the general case, we need to process the template once for each term in the *disjunctive normal form* of the where clause. The disjunctive normal form breaks the where clause into sets of requirements; within each set, we "and" all of the requirements, then we "or" the sets themselves together. For example, the where clause

```
(A<T> || B<T>) && (C<T> || D<T>)
```

would have the following disjunctive normal form:

```
(A<T> && C<T>) || (A<T> && D<T>) || (B<T> && C<T>) || (B<T> && D<T>)
```

Thus, a template with this where clause would need to be parsed and type-checked four different times. There is some concern about compilation times with "or" constraints, since the size of the disjunctive normal form is exponential in the number of "or" constraints, but without an implementation we cannot verify whether this will truly be a problem in practice.

Implementing the conversion to disjunctive normal form (DNF) proved harder than expected. While the algorithm for converting a logical expression to DNF is well-known, the introduction

of constraints in where clauses also affects name lookup, complicating the DNF computation for where clauses. For example, consider the following template:

```
template<DefaultConstructible T>
where (Addable<T> || Plus<T>) && Fooable<decltype(T() + T())>
void bar(T x , T y) {
  foo(x + y);
}
```

In this example, the + in the decltype expression will refer to Addable<T>::**operator**+ or Plus<T>::**operator**+, depending on whether the first requirement is Addable<T> or Plus<T>, respectively. Thus, the Fooable requirement will need to be parsed twice. The situation is complicated further when the "or" constraints are hidden inside the concepts themselves, because the parser must perform significant semantic analysis while parsing the where clause. Due to the inherent complexity in the parsing and computation of where clauses with "or" constraints, there are no known implementations of this feature.

Since we have no implementation of "or" constraints, we have found several alternatives using the other features of concepts. For example, rather than parsing a template for either Integral<T> or Floating<T>, we could create a new concept that contains only the requirements common to both concepts, e.g.,

```
concept Numeric<typename T> {
  T operator-(T);

  // other numeric operations...
}

template<typename T> where Numeric<T> T negate(T x) { return -x; }
```

Then, when we define the Integral and Floating concepts, we make them refinements of the Numeric concept:

```
concept Integral<typename T> : Numeric<T> { /* ... */ }
concept Floating<typename T> : Numeric<T> { /* ... */ }
```

Now, Integral and Floating types can both be used with negate(). However, we might not have the luxury to redefine the Integral and Floating concepts. In this case, we can write concept map templates to make each Integral or Floating type Numeric:

```
template<typename T> where Integral<T> concept_map Numeric<T> { }
template<typename T> where Floating<T> concept_map Numeric<T> { }
```

In our experience, we have not come across any uses of "or" constraints that cannot also be expressed with one of these workarounds. Additionally, within the C++ Standard Library, we have not found many uses for "or" constraints.

4 Recommendations

We recommend the following two changes to the concepts proposal:

1. *Remove "or" constraints*: Their implementation cost has turned out very high, and the workarounds are good enough for the use cases we've seen.
2. *Replace "&&" with "," for "and" constraints*: If users are writing "A<T> && B<T>", it begs the question of where the "||" constraints are.