Document number P0587R0

Date 2017-02-05

Authors Richard Smith <richard@metafoo.co.uk>
James Dennett <jdennett@google.com>

Audience Evolution

Concepts TS revisited

Introduction

The Concepts TS has proven to be an invaluable resource for exploring the design space of predicate-style concepts in C++. We believe we have enough information now to know that

- Many of its ideas work well in the context of C++ — for instance, modeling concepts as type predicates, partial ordering for constrained templates through a concept refinement relationship, and allowing a constrained template to depend on constructs that are not described by its constraints;
- Some parts need syntactic and in some cases semantic adjustment to mesh well with the C++ language and existing implementations; and
- Other parts should be left behind.

This paper attempts to identify these areas.

Concept definition

"`concept bool`" is redundant: all concepts are boolean. Reusing an existing construct such as a function or variable template is an expedient way to simplify the specification for a TS, but for standard C++, concepts are sufficiently central that they should be first-class.

Modeling concepts as functions and variables leads to many unnecessary problems; some highlights:

- The choice of function versus variable is an implementation detail, but unavoidably leaks out of the interface (the syntax for a concept check is different in the two cases: `Concept<X>` vs `Concept<X>()`).
- Function and variable templates are instantiated on use and produce distinct entities; concepts must be introspected, decomposed, and substituted into the point of use to support partial ordering.

- Modeling concepts as functions permits nonsense such as taking the address of a concept.

Concept overloading creates cognitive ambiguity — is `EqualityComparable<T>` “types that are equality comparable with T” or “a check that T is equality comparable with itself”? The answer is **both**, in different contexts.

Proposed changes

=> Add first-class concept definition syntax, remove `concept bool`
=> (Optional) do not allow concept overloading

Before	**After**

`template<typename T>`
`concept bool X = Concept1<T> &&`
`                 Concept2<T>();`
	`template<typename T>`
	`concept X = Concept1<T> &&`
	`            Concept2<T>;`
	or
	`concept<typename T> X = //…`
	or even
	`concept X<typename T> = //…`

Constraint decomposition

The Concepts TS decomposes constraints into atomic constraints, even inside *requires-expression*s. This promotes textual duplication, introduces fragility and coupling between implementation details, and does not serve a useful purpose.

Example:

```
template<typename T> void f()
  requires requires { g(declval<T&>()); ++declval<T&>(); } // #1
template<typename T> void f()
  requires requires { g(declval<T&>()); } // #2
```

#1 is more specialized than #2, through brittle textual duplication. If #1 is refactored as

```
template<typename T> void f()
  requires requires(T &t) { g(t); ++t; }
```

… then it is no longer considered more specialized. We think it would be better if #1 and #2 are simply unordered, and for partial ordering to be determined by semantic concept requirements rather than arbitrary textual duplication.

The useful feature provided by constraint decomposition is the ability to specify that the constraints on one template are a refinement of those on another. “`&&`” constraints allow a concept to specify that it refines some existing concept; in practice, “`||`” constraints are used to specify the reverse: that some existing concept or constraint is a refinement of the new concept or constraint.

Proposed changes

=> Remove full decomposition and ordering based on textual duplication; only order concepts
=> Concepts explicitly specify which concepts they refine and which they are refinements of

A new syntax is provided to retroactively annotate that an existing concept is a refinement of a newly-introduced concept, in order to preserve the ability to write the common cases of `||` constraints. Such added constraints are not checked during satisfaction checks for the original concept, and only affect partial ordering. The program is ill-formed if they introduce a cycle into the concept refinement graph.

Before	**After**

`template<typename T>`
`concept bool X = Concept1<T> &&`
`                 Concept2<T>();`
`template<typename T>`
`concept bool Y = X<T> &&`
`  sizeof(T) == 4 &&`
`  requires (T t) { ++t; };`
 	
`// Integral and Floating are`
`// provided by some library.`
`template<typename T>`
`concept bool Primitive =`
`  Integral<T> || Floating<T>;`
	`template<typename T>`
	`concept X : Concept1<T>,`
	`            Concept2<T>;`
	`template<typename T>`
	`concept Y : X<T> requires (T t)`
	`{`
	`  requires sizeof(T) == 4;`
	`  ++t;`
	`};`
	
	`template<typename T>`
	`concept Primitive`
	`  requires { /*...*/ };`
	
	`// Declare that Integral and`
	`// Floating are refinements of`
	`// Primitive.`
	`template<typename T>`
	`extern concept Integral<T>`
	`  : Primitive<T>;`
	`template<typename T>`
	`extern concept Floating<T>`
	`  : Primitive<T>;`

Requires clauses

The boolean expression syntax is confusing as not all boolean operators are taken into account, and it adds novel implementation requirements for determining equivalence, with a worst-case exponential time subsumption algorithm.

Permitting arbitrary expressions within *requires-clause*s places unreasonable requirements on implementations, requiring complex constraints to be mangled into template signatures (for implementations relying on name mangling). The resulting “`requires requires`” syntax is confusing and embarrassing.

Proposed changes

=> Requires clause specifies list of required concepts, not arbitrary boolean expression.
=> Requirements of a template cannot contain arbitrary expressions, just *named* concepts.

Combined with the proposed removal of constraint decomposition, there is now a simple, reasonably efficient test for partial ordering: each template simply has a set of associated concepts, and one template is at least as specialized than another if each of the other template’s concepts is the same as, or transitively refined by, at least one of the first template’s concepts.

Before	**After**
`template<typename T>`	`template<typename T>`
`void f() requires requires`	`concept Xable requires`
` { x(T()); } {}`	`  { x(T()); };`
`template<typename T>`	`template<typename T>`
`void g() requires A<T> && B<T>;`	`void f() requires Xable<T> {}`
	or, as in Concepts TS v1,
	`template<Xable T> void f() {}`
	`template<typename T>`
	`void g() requires A<T>, B<T>;`

Summary of proposed concept definition syntax

While the above list is presented as a set of piecemeal changes, there is a big picture design here. The key idea is to model a concept definition after a type definition, rather than after a function or variable definition. C++ has a semi-uniform pattern for such cases, which we propose to follow:

```
keyword name : refined-type-1, refined-type-2 { body };
class Foo : public Bar { int n; /*...*/ };
enum E : int { a, b, c };
concept Foo : Bar<T>, Baz<T> requires (T t) { t++; };
```

The specific proposed syntax for a *concept-declaration* is:

`concept` *identifier* **:** *concept-id-list*$_{\text{opt}}$ *requires-expression*$_{\text{opt}}$ **;**

where the `requires` keyword may be omitted from the *requires-expression* when the *parameter-declaration-clause* is omitted. It should be noted that this implies a little extra syntax compared to a Concepts TS variable concept for the case where a concept is defined as a boolean expression:

```
template<typename T> concept IsSmall {
  requires sizeof(T) <= 8;
};
```

… and a little less syntax for concepts that are defined primarily as validity requirements on their type parameter, which we expect to be the common case. (And increasingly so as more code transitions from wrapping type traits in concepts to expressing type predicates as concepts directly.)

Requires expressions

The Concepts TS's *requires-expression*s are, at their heart, a first-class way to perform expression/statement/type validity ("SFINAE") checks. However, the syntax chosen to express these checks is unnecessarily inventive, providing neither a familiar way to express the desired syntactic constructs nor a system that trivially allows production of an archetype for constrained template definition checking. (Note, we're not claiming that such checks are impossible, rather that the necessary constraints would need to be carefully inferred from the set of valid constructs.)

requires-expressions are also quite limited: they do not model all of C++. While there is a syntax to specify an implicit conversion constraint, there is no syntax to check the validity of other forms of variable declaration and initialization. (Example: copy-list-initialization -- that is, the initialization form `T x = {a,b};` -- is not straightforward to model in a *requires-expression*.)

Additionally, the same syntax in a convertability requirement in a *compound-requirement* means different things depending on whether the right-hand side contains a placeholder:

```
template<typename T> concept bool C = // ...
template<typename T> concept bool X =
  requires (int x) {
    {f(x)} -> T*; // f(x) can be implicitly converted to type T*
    {f(x)} -> C*; // type of f(x) is pointer to type satisfying C
  };
```

Proposed changes

=> Replace syntax with '`requires` *parameter-declaration-clause compound-statement*'

- can use any syntax permitted in a function or lambda body

- result is `true` if substitution into body succeeds, `false` if not
- extend and reuse existing language syntax, do not invent a new mini language

=> For syntactic convenience, add *nested-requirement* form as regular declaration

- `requires x;` is equivalent to `static_assert(x);` but more natural[1]

Before
```
template<typename T>
void f() {
 bool b = requires (T t) {
   {x(t)} -> int;
   // require that range-based
   // 'for' will work on 't'
   requires requires
     (decltype(begin(t)) b,
      decltype(end(t)) e) {
     b != e;
     ++b;
     *b;
   };
 };

 static_assert(Concept<T>);
}
```

After
```
template<typename T>
void f() {
  bool b = requires (T t) {
    int n = x(t);
    for (auto v : t) {}
  };
  requires Concept<T>;
}
```

Terse template notations

The Concepts TS introduces too many syntaxes for a function template declaration. Some of those syntaxes have no clear, consistent syntactic marker for templates, which is important semantic information for the reader of the code (remembering that code is read vastly more than it is written). Consider:

```
      void f(Something x) { foo(x); }
```

Is this declaring a function template? That has deep implications for its comprehension, refactoring, and use. With the Concepts TS, we cannot tell unless we look up the definition of `Something` and determine whether it is in fact a concept name.

The above form also significantly harms any sound and consistent notion of what a *concept-name* represents, and harms the consistency of the language. Consider:

```
      // A is a value of type int.
      // typename is notionally the kind (type-of-a-type) of types,
      so
      // B is a value of type typename.
```

[1] If we wish to fully solve the issue of *concept-names* being usable at different levels of the sort hierarchy, we could further restrict the use of `Concept<T>` as a boolean value to only being permitted in a *requires-clause / nested-requirement*. However, the practical benefits of permitting this usage in an `if constexpr` statement may be sufficient to discourage this.

```
      // Regular is the kind of regular types, so
      // C is a value of type Regular.
      template<int A, typename B, Regular C>
      // a is a value of type int.
      // b is ill-formed because values of type typename are types.
      // c is unjustifiably valid, and ontologically wrong.
      void f(int a, typename b, Regular c);
```
This form must be revised. Accepting a type concept as a function parameter is every bit as wrong as accepting "typename b" as a function parameter.

Conversely, given:
```
      template<int N> concept bool Even = N % 2 == 0;
```
This seems quite reasonable (if perhaps not all that useful):
```
      template<Even N> void f() {
            constexpr Even X = N;
      }
```
Note that in this case, values of type `Even` are ontologically at the "value" level not at the "type" level because it is a concept constraining a value (non-type template parameter), not a concept constraining a type.

Consider also the *template-introduction* syntax:
```
      Concept{A, B, C}
      void f(A a, B b, C c);
```
Assuming `Concept` is not overloaded and takes three type parameters, this is shorthand for
```
      template<typename A, typename B, typename C>
      void f(A a, B b, C c) requires Concept<A, B, C>;
```
Note that this syntax is extremely limited: if the template has *any* other template parameters, it cannot be used (allowing multiple such lists introduces ambiguity into the language). It's also extremely and unnecessarily inventive; the language constructs it most closely resembles mean something wholly different (they would be uses of A, B, and C, not declarations!).

Proposed changes

=> Replace template-introduction syntax with a syntax based on decomposition declarations that permits additional template parameters to be declared

Replace
```
  Concept{A, B, C} void f(A, B, C) {}
```
with
```
  template<Concept [A, B, C]> void f(A, B, C) {}
```
(or similar). Note that this generalizes to multiple parameters:
```
  template<Concept [A, B, C], typename T, Concept2 [D, E]>
```

=> Require an explicit sigil to declare a function to be a template

Replace
```
  void f(auto a) {}
  template<typename T> void g(auto a) {}
```
with
```
  template<...> void f(auto a) {}
  template<typename T, ...> void g(auto a) {}
```

The `...` sigil in the *template-parameter-list* indicates that additional template parameters will be inferred from the declaration.

=> Remove or revise terse template notation

Remove
```
  void f(ConceptName) {}
```
… but perhaps add something like
```
  template<...> void f(ConceptName auto a) {}
  template<...> void f(ConceptName T a, T b) {}
```
… where the latter declares `T` to be a type template parameter of type `ConceptName`, and `a` and `b` to be of type `T`.

Keep
```
  template<ConceptName X> void f(X a) {}
```

Summary of proposed template definition syntax

The guiding design ideas leading to the above proposed syntax are:

- A common, uniform syntax for template definitions, where every template definition begins with the syntactic marker `template<`
- Reuse of existing C++ syntax for declarations introducing multiple template parameters (structured binding syntax)
- Each concept-id declares only values, or only types, or only kinds, and the choice is not context-dependent.

Constrained template redeclaration

The Concepts TS tries to define the various concept syntaxes as essentially syntactic desugarings of a more fundamental syntax, allowing redeclarations of the same template to use different syntaxes. This approach is poorly-conceived and highly underspecified (see various threads on the core reflector about whether and where parentheses are inserted, the order in which components of the requires-clause are emitted, the precise syntax that forms desugar into), and does not fit into the C++ template model, which uses a token-stream equivalence model for all other such checks throughout the language. For example:

```
template<C T, C U, C V> void f();
template<typename T, typename U, typename V>
  void f() requires (C<T> && C<U>) && C<V>;
template<typename T, typename U, typename V>
  void f() requires C<T> && (C<U> && C<V>);
```

Does the first `f` declare the same function as the second or the third? Neither? Both? This approach should be revisited. There is simply no reason to allow different syntactic forms to be used in different declarations of the same template.

Proposed changes

Remove the special case. Require that the same syntax must be used in constrained function template redeclarations, as is the case in C++ today.

Acknowledgements

Most of the issues and ideas here are not our own. They are primarily drawn from two sources:

- A discussion of the assembled members of the BSI national body at Jacksonville, and
- An informal meeting of committee members interested in revisiting and updating the Concepts TS at Oulu.

While this paper was written at the request of the assembled members of the BSI national body, it does not necessarily represent the official position of BSI nor of those committee members attending the Oulu meeting.

We want to thank everyone involved in those discussions, along with hallway conversations and the like during prior committee meetings, and hope that this paper will be a starting point for the refinements that we think are necessary in order to make Concepts an elegant and consistent part of C++20.