Document number: N3580
Date: 2013-03-17
Study group: Concepts
Reply to: Andrew Sutton <asutton@cs.tamu.edu>
Bjarne Stroustrup <bs@cs.tamu.edu>
Gabriel Dos Reis <gdr@cs.tamu.edu>

Concepts Lite: Constraining Templates with Predicates

Andrew Sutton, Bjarne Stroustrup, Gabriel Dos Reis

Texas A&M University
Department of Computer Science and Engineering
College Station, Texas 77843

1 Introduction

In this paper, we introduce template constraints (a.k.a., "concepts lite"), an extension of C++ that allows the use of predicates to constrain template arguments. The proposed feature is minimal, principled, and uncomplicated. Template constraints are applied to enforce the correctness of template use, not the correctness of template definitions. The design of these features is intended to support easy and incremental adoption by users. More precisely, constraints:

- allow programmers to directly state the requirements of a set of template arguments as part of a template's interface,
- support function overloading and class template specialization based on constraints,
- fundamentally improve diagnostics by checking template arguments in terms of stated intent at the point of use, and
- do all of this without any runtime overhead or longer compilation times.

This work is implemented as a branch of GCC-4.8 and is available for download at http://concepts.axiomatics.org/~ans/. The implementation includes a compiler and a modified standard library that includes constraints. Note that, as of the time of writing, all major features described in this report have been implemented.

This paper is organized like this:

- Tutorial: introduces the basic notions of constraints, shows examples of their use, and gives examples of how to define constraints.

- Discussion: explains what constrains are not. In particular, we try to outline constraints's relation to concepts and to dispel some common misconceptions about concepts.
- User's guide: provides many more tutorial examples and demonstrate the completeness of the constraints mechanism.
- Implementation: gives an overview of our GCC compiler support for constraints.
- Extensions: we discuss how constraints might be extended to interact with other proposed features.
- Language definition: presents a semi-formal definition of constraints

2 Constraints Tutorial

This section is a tutorial of the template constraints language feature. We present the basic concepts of the feature and describe how to constrain templates, and show what constraint definitions are.

2.1 Introducing Constraints

A template constraint is part of a template parameter declaration. For example, a generic `sort` algorithm might be declared as:

```
template<Sortable Cont>
void sort(Cont& container);
```

Here, `Sortable` is a constraint that is written as the "type" of the template parameter `Cont`. The constraint determines what kinds of types can be used with the `sort` algorithm. Here, `Sortable` specifies that any type template argument for `sort` must be "sortable," that is, be a random-access container with an element type with a <. Alternatively, we can introduce constraints using a **`requires`** clause, in which constraints are explicitly called:

```
template<typename Cont>
  requires Sortable<Cont>()
void sort(Cont& cont)
```

These two declarations of `sort` are equivalent. The first declaration is a shorthand for the second. We generally prefer the shorthand since it is often more concise and resembles the conventional type notation.

The **`requires`** clause is followed by a Boolean expression that evaluates predicates. There is no magic to the definition of `Sortable`: it is just a **`constexpr`** function returning **`true`** when its type argument is a permutable random access container with a totally ordered element type. The predicate is evaluated at compile time and constrains the use of the template.

Trying to use the algorithm with a `list` does not work since `std::sort` is not directly implemented for bidirectional iterators in the standard library.

```
list<int> lst = ...;
sort(lst); // Error
```

In C++11, we might expect a fairly long error message. It depends how deeply in the sequence of nested function calls the `sort` algorithm tries to do something that a bidirectional iterator does not support, like adding `n` to an iterator. The error messages tend to be somewhat cryptic: "`no 'operator[]' available`". With constraints, we can get much better diagnostics. Then program above results in the following error.

```
error: no matching function for call to 'sort(list<int>&)'
   sort(l);
        ^
note: candidate is:
note: template<Sortable T> void sort(T)
```

```
    void sort(T t) { }
         ^
note: template constraints not satisfied because
note:   'T' is not a/an 'Sortable' type [with T = list<int>] since
note:     'declval<T>()[n]' is not valid syntax
```

Please note that this is real computer output, rather than a mere conjecture about what we might be able to produce. If people find this too verbose, we plan to provide a compiler option to suppress the "notes".

Constraints violations are diagnosed at the point of use, just like type errors. C++98 (and C++11) template instantiation errors are reported in terms of implementation details (at instantiation time), whereas constraints errors are expressed in terms of the programmer's intent stated as requirements. This is a fundamental difference. The diagnostics explain which constraints were not satisfied and the specific reasons for those failures.

The programmer is not required to explicitly state whether a type satisfies a template's constraints. That fact is computed by the compiler. This means that C++11 applications written against well-designed generic libraries will continue to work, even when those libraries begin using constraints. For example, we have put constraints on almost all STL algorithms without having to modify user code.

For programs that do compile, template constraints add no runtime overhead. The satisfaction of constraints is determined at compile time, and the compiler inserts no additional runtime checks or indirect function calls. Your programs will not run more slowly if you use constraints.

Constraints can be used with any template. We can constrain and use class templates, alias templates, and class template member function in the same way as function templates. For example, the `vector` template can be declared using shorthand or, equivalently, with a **`requires`** clause.

```
// Shorthand constraints
template<Object T, Allocator A>
class vector;

// Explicit constraints
template<typename T, typename A>
  requires Object<T>() && Allocator<A>()
class vector;
```

When we have constraints on multiple parameters, they are combined in the **`requires`** clause as a conjunction. Using `vector` is no different than before, except that we get better diagnostics when we use it incorrectly.

```
vector<int> v1;  // Ok
vector<int&> v2; // Error: 'int&' does not satisfy the constraint 'Object'
```

Constraints can also be used with member functions. For example, we only want to compare `vector`s for equality and ordering when the value type can be compared for equality or ordering.

```
template<Object T, Allocator A>
class vector
{
  requires Equality_comparable<T>()
    bool operator==(const vector& x) const;

  requires Totally_ordered<T>()
    bool operator<(const vector& x) const;
};
```

The **requires** clause before the member declaration introduces a constraint on its usage. Trying to compare two vectors of a type that are not equality comparable or totally ordered results an an error at the point of use, not from inside `std::equal` or `std::lexicographical_compare`, which is what happens in C++98 and C++11.

2.1.1 Multi-type Constraints

Constraints on multiple types are essential and easily expressed. Suppose we want a `find` algorithm that searches through a `sequence` for an element that compares equal to `value` (using ==). The corresponding declaration is:

```
template<Sequence S, Equality_comparable<Value_Type<S>> T>
Iterator_type<S> find(S&& sequence, const T& value);
```

`Sequence` is a constraint on the template parameter `S`. Likewise, `Equality_comparable<Value_type<S>>` is a constraint on the template parameter `T`. This constraint depends on (and refers to) the previously declared template parameter, `S`. It's meaning is that the parameter `T` must be equality comparable with the value type of `S`. We could alternatively and equivalently express this same requirement with a **requires** clause.

```
template<typename S, typename T>
  requires Sequence<S>() && Equality_comparable<T, Value_type<S>>()
Iterator_type<S> find(S&& sequence, const T& value);
```

Why have two alternative notations? Some complicated constraints are best expressed by a combination of the shorthand notation and **requires** expressions. For example:

```
template<Sequence S, typename T>
  requires Equality_comparable<T, Value_type<S>>()
Iterator_type<S> find(S&& sequence, const T& value);
```

The choice of style is up to the user. We tend prefer the concise shorthand. In "Concept Design for the STL" (N3351=12-0041) we showed that the shorthand notation is sufficiently expressive to handle most of the STL [1].

2.1.2 Overloading

Overloading is fundamental in generic programming. Generic programming requires semantically equivalent operations on different types to have the same

name. In C++11, we have to simulate overloading using conditions (e.g., with `enable_if`), which results in complicated and brittle code that slows compilations.

With constraints, we get overloading for free. Suppose that we want to add a `find` that is optimized for associative containers. We only need to add this single declaration:

```
template<Associative_container C>
Iterator_type<C> find(C&& assoc, const Key_type<C>& key)
{
  return assoc.find(key);
}
```

The `Associative_container` constraint matches all associative containers: `set`, `map`, `multimap`, ... basically any container with an associated `Key_type` and an efficient `find` operation. With this definition, we can generically call `find` for any container in the STL and be assured that the implementation we get will be optimal.

```
vector<int> v { ... };
multiset<int> s { ... };

auto vi = find(v, 7); // calls sequence overload
auto si = find(s, 7); // calls associative container overload
```

At each call site, the compiler checks the requirements of each overload to determine which should be called. In the first call to `find`, `v` is a `Sequence` whose value type can be compared for equality with `7`. However, it is not an associative container, so the first overload is called. At the second call site `s` is not a `Sequence`; it is an `Associative_container` with `int` as the key type, so the second overload is called.

Again, the programmer does not need to supply any additional information for the compiler to distinguish these overloads. Overloads are automatically distinguished by their constraints and whether or not they are satisfied. Basically, the resolution algorithm picks the unique best overload if one exists, otherwise a call is an error. For details, see Section 4.3.

In this example, the requirements are largely disjoint. It is unlikely that we will find many containers that are both `Sequence`s and `Association`s. For example, a container that was both a `Sequence` and an `Association` would have to have both a `c.insert(p,i,j)` and a `c.equal_range(x)`. However, it is often the case that requirements on different overloads overlap, as with iterator requirements. To show how to handle overlapping requirements, we look at a constrained version of the STL's `advance` algorithm in all its glory.

```
template<Input_iterator I>
void advance(I& i, Difference_type<I> n)
{
  while (n--) ++i;
}
```

```
template<Bidirectional_iterator I>
void advance(I& i, Difference_type<I> n)
{
  if (n > 0) while (n--) ++i;
  if (n < 0) while (n++) --i;
}

template<Random_access_iterator I>
void advance(I& i, Difference_type<I> n)
{
  i += n;
}
```

The definition is simple and obvious. Each overload of `advance` has progressively more restrictive constraints: `Input_iterator` being the most general and `Random_accesss_iterator` being the most constrained. Neither type traits nor tag dispatch is required for these definitions or for overload resolution.

Calling `advance` works as expected. For example:

```
list<int>::iterator i = ...;
advance(i, 2); // Calls 2nd overload
```

As before, some overloads are rejected at the call site. For example, the random access overload is rejected because a `list` iterator does not satisfy those requirements. Among the remaining requirements the compiler must choose the most specialized overload. This is the second overload because the requirements for bidirectional iterators include those of input iterators; it is therefore a better choice. We outline how the compiler determines the most specialized constraint in 4.3.

Note that we did not have to add any code to resolve the call of `advance`. Instead, we computed the correct resolution from the constraints provided by the programmer(s).

A conventional (unconstrained C++98) template parameter act as of "catch-all" in overloading. It is simply represents the least constrained type, rather than being a special case . For example, a `print` facility may have:

```
template<typename T>
void print(const T& x);

template<Container C>
void print(const C& container);

// ...
vector<string> v { ... };
print(v); // Calls the 2nd overload

complex<double> c {1, 1};
print(c); // Calls the 1st overload.
```

An unconstrained template is obviously less constrained than a constrained template and is only selected when no other candidates are viable. This implies

that older templates can co-exist with constrained templates and that a gradual transition to constrained templates is simple.

Note that we do not need a "late check" notion or a separate language constructs for constrained and unconstrained template arguments. The integration is smooth.

2.2 Defining Constraints

We now look at the definition of constraints. What do they look like? Here is a declaration of `Equality_comparable`.

```
template<typename T>
constexpr bool Equality_comparable();
```

A constraint is simply an unconstrained, **constexpr** function template that takes no arguments and returns **bool**. It is—in the most literal sense—a predicate on template arguments. This also means that the evaluation of constraints in a **requires** clause is the same as **constexpr** evaluation.

The `Equality_comparable` constraint can be defined like this:

```
template<typename T>
concept Equality_comparable()
{
  return has_eq<T>::value && is_convertible<eq_result<T>, bool>::value
      && has_ne<T>::value && is_convertible<ne_result<T>, bool>::value;
}
```

Here, `has_eq` and `has_neq` are type traits that determine whether arguments of type `T` can be used with the `==` and `!=` operators. These can be implemented in C++11 using advanced metaprogramming techniques. `is_convertible` is also a type trait, defined in the Standard Library. `eq_result` and `ne_result` are alias templates referring to the result type of the corresponding expressions.

The use of traits and aliases to define syntactic requirements does not scale. For every conceivable expression, a library writer may require several template facilities to evaluate different properties. This can dramatically increase the size of a library and reduce compile times.

To aid in the implementation of constraints, we introduce a new feature aimed at minimizing these overheads. We can express exactly the same requirements using a **requires** expression.

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

The **requires** expression introduces a conjunction of *syntactic requirements*, properties of types that can be checked at compile time. The expression can

introduce local parameters which are used in the writing of nested requirements. Here, `a` and `b` can be used to denote values or expressions of type `T` for the purpose of writing constraints.

Each statement nested in a **`requires`** expression denotes a conjunction of requirements on a *valid expression* or *associated type*. A valid expression is an expression that must compile when instantiated with template arguments. A statement can also include requirements on the result type of a valid expression. For example, the statement **`bool`** `= a == b` includes two requirements:

- `a == b` must be a valid expression for all arguments of type `T`, and
- the result of that expression must be convertible to **`bool`**.

The result type could also be constrained to be the same as another type by writing e.g., **`bool`** `== {a == b}`, or the result type requirement can be omitted by simply writing the valid expression e.g., `a == b`. If the expression cannot be compiled when instantiated, or if the result type requirement returns **`false`**, then the syntactic requirement is not satisfied and also returns **`false`**.

An associated type requirement is also a syntactic requirement. For example, the `Readable` concept requires an associated `Value_type`, which represents the type of object referenced. We can implement a constraint as:

```
template<typename T>
constexpr bool Readable()
{
  return requries (T i) {
    typename Value_type<I>;
    const Value_type<I>& = {*i};
  }
}
```

The statement **`typename`** `Value_type<I>` requires that the alias `Value_type<I>` must compile when instantiated. If not, the requirement is not satisfied and returns **`false`**.

We further explain features of the **`requires`** expression in Section 4.7.

3 Constraints and Concepts

Template constraints (concepts-lite) provide a mechanism form constraining template arguments and overloading functions based on constraints. Our long-term goal is a complete definition of concepts, which we see as a full-featured version of constraints. With this work, we aim to take that first step. Constraints are a dramatic improvement on `enable_if`, but they are definitely not complete concepts.

First, constraints are not a replacement for type traits. That is, libraries written using type traits will interoperate seamlessly with libraries written using constraints. In fact, the motivation for constraints is taken directly from existing practice—the use of `enable_if` and type traits to emulate constraints on templates. Many constraints in our implementation are written directly in terms of existing type traits (e.g., `std::is_integral`).

Second, constraints do not provide a concept or constraint definition language. We have not proposed any language features that simplify the definition of constraints. We hold this as future work as we move towards a complete definition of concepts. Any new language features supporting constraint definition would most likely be obviated by concepts in the future. That said, our implementation does provide some compiler intrinsics that support the implementation of constraints and would ease the implementation of concepts. This feature is detailed in Section 5.

Third, constraints are not concept maps. Predicates on template arguments are automatically computed and do not require any additional user annotations to work. A programmer does not need to create a specialization of `Equality_comparable` in order for that constraint to be satisfied. Also unlike C++0x concepts, constraints do not change the lookup rules inside concepts.

Finally, constraints do not constrain template definitions. That is, the modular type checking of template definitions is not supported by template constraints. We expect this feature to be a part of a final design for concepts.

The features proposed for constraints are designed to facilitate a smooth transition to a more complete concepts proposal. The mechanism used to evaluate and compare constraints readily apply to concepts as well, and the language featurese used to describe requirements (type traits and compiler intrinsics) can be used to support various models of separate checking for template definitions.

The constraints proposal does not directly address the specification or use of semantics; it is targetted only at checking syntax. The constraint language described in this papers has been designed so that semantic constraints can be readily integrated in the future.

However, we do note that virtually every constraint that we find to be useful has associated semantics (how could it not?). Semantics should be documented along with constraints in the form of e.g., comments or other external definitions. For example, we might document `Equality_comparable` as:

```
template<typename T>
constexpr bool Equality_comparable()
{
```

```
    ... // Required syntax
}
// Semantics:
// For two values a and b, == is an equivalence relation that
// returns true when a and b represent the same entity.
//
// The != operator is equivalent to !(a == b).
```

Failing to document the semantics of a constraint leaves its intent open to different interpretations. Work on semantics is ongoing and, for the time being, separate from constraints. We hope to present on the integration of these efforts in the future. We see no problems including semantic information in a form similar to what was presented in N3351 [1].

4 User's Guide

This section expands on the tutorial and gives more examples of how constraints interact with various language features. In particular, we look more closely at constraints, discuss overloading concerns, examine constraints on member functions, partial class template specializations. This section also describes constraints on non-type arguments and the interaction of constraints with variadic templates. We begin with a thorough explanation of constraints.

A constraint is simply a C++11 constant expression whose result type can be converted to 0. For example, all of the following are valid constraints.

```
Equality_comparable<T>()
requires (T a) { bool = {a < a}; }
!is_void<T>::value
is_lvalue_reference<T>::value && is_const<T>::value
is_integral<T>::value || is_floating_point<T>::value
N == 2
X < 8
```

A constraint is satisfied when the expression evaluates, at compile-time to `true`. This is effectively everything that a typical user (or even an advanced user) needs to know about constraints.

However, in order to solve problems related to redeclaration and overloading, and to improve diagnostics, the compiler must reason about the content of these constraints.

4.1 Anatomy of a Constraint

The following section describes the compiler's view of a constraint and is primarily intended as an introduction to the semantics of the proposed features.

In formal terms, constraints are written in a constraint language over a set of atomic propositions and using the logical connectives and (`&&`) and or (`||`). For those interested in the logical aspects of the language, it is a subset of propositional calculus.

In order to reason about the equivalence and ordering of constraints the compiler must decompose a constraint expression into sets of atomic propositions.

An atomic proposition is a C++ constant expression that evaluates to either `true` or `false`. These terms are called *atomic* because the compiler can only evaluate them. They are not further analyzed or decomposed. These include things like type traits (`is_integral<T>::value`), relational expressions (`N == 0`), and some `constexpr` functions are also atomic (e.g., `is_prime(N)`).

The reason that expressions like `is_integral<T>::value` and `is_prime(N)` are atomic is that there they may be multiple definitions or overloads when instantiated. `is_integral` could have a number of specializations, and `is_prime` could have different overloads for different types of `N`. Specializations or overloads could also be defined after the first use in a constraint. Trying to decompose these declarations would be unsound. However, they can still be used and

evaluated as constraints. Some functions are given special meaning, which we describe in the next section.

Negation (e.g., `!is_void<T>::value`) is also an atomic proposition. These expressions can be evaluated but are not decomposed. While negation has turned out to be fairly common in our constraints (see Section 5.3), we have not found it necessary to assign deeper semantics to the operator.

Atomic propositions can be also be nested and include arithmetic operators, calls to **constexpr** functions, conditional expressions, literals, and even compound expressions. For example, `(3 + 4 == 8, 3 < 4)` is a perfectly valid constraint, and its result will always be **true**.

4.1.1 Constraint Predicates

While some function calls in constraints are atomic propositions, calls to simple functions like `Equality_comparable` and `Convertible` are decomposed into their constituent parts. We call these kinds functions are *constraint predicates*. A function is constraint predicate only if it is

- a function template,
- has no function parameters,
- returns **bool**,
- is **constexpr**,
- and has a single definition for all template arguments.

A call to any other kind of function is an atomic proposition.

Recall that the definition of `Equality_comparable` from Section 2:

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

It is a constraint predicate that is a conjunction of four requirements: the validity of the expressions `a == b` and `a != b` and the convertibility of each expression to **bool**.

When included in a template's **requires** clause, a call to a constraint predicate is called a *constraint check*. Check expressions are recursively expanded, inlining the definition of the predicate into the expression. For example, suppose we have this:

```
template<Equality_comparable T>
bool distinct(T a, T b) { return a != b; }
```

The shorthand `Equality_comparable` requirement is first transformed into a constraint expression: `Equality_comparable<T>()`. Because `Equality_comparable` is a constraint predicate, it is recursively expanded (as is the nested check of `Convertible`). Ultimately, the previous declaration is equivalent to having written this:

```
template<typename T>
  requires has_eq<T>::value
        && is_convertible<eq_result<T>, bool>::value
        && has_ne<T>::value
        && is_convertible<ne_result<T>, bool>::value
bool distinct(T a, T b)
```

Note that the representation of syntactic constraints is internal to the compiler. Here, we have chosen to represent the corresponding syntactic requirements as type traits.

Obviously, it is far more concise to express constraints in terms of constraint predicates than formulas comprised of atomic propositions. They are the basic building block of conceptual abstractions. Concepts like `Input_iterator`, `Range`, and `Relation` are defined through the composition of constraint predicates.

Recursively breaking constraint predicates into their constituent parts allows us much better analysis, more flexibility, and greatly simplifies the definition of overloading and ambiguity.

4.1.2 Overloading Constraints

Constraint predicates can be overloaded based on the number of type parameters. For example, in [1], we found it useful to define cross-type concepts that extended the usual definitions of equality and ordering to operands of different (but related) types.

```
template<typename T>
constexpr bool Equality_comparable();

template<typename T, typename U>
constexpr bool Equality_comparable();
```

A number of algorithms require the equality comparison of two value types that are not necessarily related.

Note that specializing or refining constraint predicates is allowed, but it will change the meaning of the predicate. For example, if a user provides a new declaration of `Equality_comparable` like this

```
template<typename T>
  requires is_same<T, my_type>::value
constexpr bool Equality_comparable();
```

Any subsequent use of `Equality_comparable` will be an atomic proposition. That is, the compiler will no longer be able to order constraints based on its individual syntactic requirements.

It has been requested that we provide some annotation on constraint predicates that would make this kind of overloading an error. However, no syntax could be agreed upon as of the time of writing. We further discuss annotations for constraints in Section 6.

4.1.3 Connectives

Once we have broken predicates up into atomic propositions, we can use straightforward classical logic and logical algorithms. Constraints are composed of propositions joined by the logical connectives `&&` (conjunction, and) and `||` (disjunction, or). These have the usual meanings, but cannot be overloaded. Parentheses can also be used for grouping.

In order to solve problems related to redeclaration and overload resolution, the compiler must decompose constraints into sets of atomic propositions based on the connectives in the constraint expression.

Conjunction (and) results in the union of requirements into a single set. For example, the `distinct` function in the previous has a set comprised of four requirements:

```
has_eq<T>::value
is_convertible<eq_result<T>, bool>::value
has_neq<T>::value
is_convertible<neq_result<T>, bool>::value
```

Constraints in the sets can be differentiated syntactically. That is, if two expressions have the same syntax, then only one needs to be retained in the set.

Disjunction (or) results in the creation of alternatives. Suppose we have constraints describing the distinct requirements for `Container`s and `Range`s, where a container has value semantics and ranges have reference semantics.

```
template<typename T>
constexpr bool Container()
{
  return value_semantic<T>::value
      && Equality_comparable<T>()
      && has_begin<T>::value
      && has_end<T>::value
      && has_size<T>::value; // Probably more..
}

template<typename T>
constexpr bool Range()
{
  return reference_semantic<T>::value
      && Equality_comparable<T>()
      && has_begin::value
      && has_end::value;
}
```

The `value_semantic` and `reference_semantic` type traits are hypothetical, but could possibly be implementing using traits classes or associated types or values. The remaining traits are similar to the `has_eq` and `has_neq` traits used earlier. The two concepts have some syntax in common, but are otherwise disjoint. It should not be possible to define a type that implements both value and reference semantics.

Nearly every algorithm in the STL can be extended to require a disjunction of these requirements. For example:

```
template<typename T>
  requires Container<T>() || Range<T>()
auto find(const T& x) -> decltype(begin(x))
{
  return find(begin(x), end(x));
}
```

The algorithm is written in the shared syntax of the different constraints. Either constraint may be satisfied, but it would be incorrect (for example) to call `size(x)` since it is not required by both constraints.

The decomposition of these requirements creates alternative sets of requirements. They are:

```
// Alternative 1 (Container<T>)
value_semantics<T>::value
has_eq<T>::value
is_convertible<eq_result<T>, bool>::value
has_neq<T>::value
is_convertible<neq_result<T>, bool>::value
has_begin<T>::value
has_end<T>::value
has_size<t>::value

// Alternative 2 (Range<T>)
reference_semantics<T>::value
has_eq<T>::value
is_convertible<eq_result<T>, bool>::value
has_neq<T>::value
is_convertible<neq_result<T>, bool>::value
has_begin<T>::value
has_end<T>::value
```

Finally, we note that the decomposition of constraints can be used to improve diagnostics. The error messages shown in Section 2 are derived from the decomposed requirements of the `Sortable` constraint. Specific messages can be crafted for specific kinds of requirements, especially those written using intrinsic functions.

The reason that negation (`!`) is not included as a logical connective in the constraints language has to do with the evolution of these features as we move towards a fuller definition of concepts. In particular, we assume that a template definition will be checked against sets of requirements included by a constraint.

As of this writing, it is not clear what a "negative requirement" means in that context. This does not mean, however, that a programmer cannot use `!` in a constraint. It simply means that the expression will be treated as atomic, even if the operand is a constraint predicate.

4.1.4 Relations on Constraints

In order to support redeclaration, overloading, and partial specialization, constraints must be compared for equivalence and ordering. This is done by comparing sets of propositions. Note that propositions are compared syntactically.

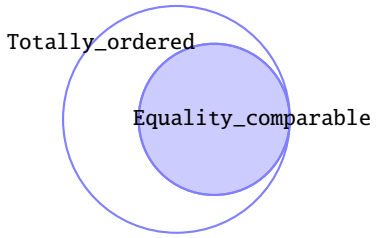
Two constraints are equivalent when the contain the same propositions. Because equivalence is based on the decomposed sets of propositions, two **`requires`** clauses may have different spellings, but may require the same things.

Constraints are partially ordered by the *subsumes* relation. Specifically, one constraint subsumes another when its requirements include those of the other. The subsumes relation is actually a generalization of the subset relation on sets that can accommodate alternatives. When neither constraint contains alternatives, the relations are the equivalent.

For example, we define `Totally_ordered` like this:

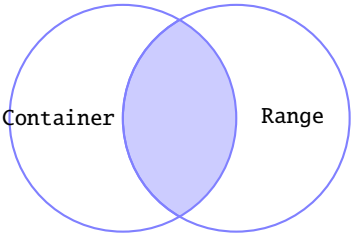
```
template<typename T>
constexpr bool Totally_ordered()
{
  return Weakly_ordered<T>() && Equality_comparable<T>();
}
```

The relationship between the requirements of `Totally_ordered` and `Equality_comparable` can be pictured like this:



`Totally_ordered` subsumes `Equality_comparable` because its requirements include those of the latter. This relation holds for any constraint predicate that explicitly includes another.

It is often the case the case that constraints overlap, with neither subsuming the other. For example, this is true of the `Container` and `Range` concepts described in the previous section. The relationship between those constraints can be pictured this way:



The subsumes relation is used to determine which of two templates is *more constrained*. In particular, a template T is more specialized than another, U iff they have the same generic type and the requirements of T subsume those of U. This relation is used to different templates with the same type when computing which is *more specialized*. Note that a constrained template is always more specialized than an unconstrained template.

This concludes the logical foundation of the constraints language and its associated relations. The remaining sections of this chapter describe how constraints interact with the C++ programming language.

4.1.5 Decomposition and Subsumption

The algorithms used to decompose a constraint into sets of atomic propositions and to determine the subsumption of constraints are simple algorithms used to support automated deduction in propositional logic. They are based on the application of *sequent calculus* to decompose complex expressions into simpler search problems. A good introduction to the underlying theory and its application can be found in "ML for the Working Programmer" [10].

Given two constraints, P and Q, determining if P subsumes (or includes) Q is equivalent to determining logically if $P \vdash Q$ is a valid statement. In other words, if we assume P to be true, does Q necessarily follow as a consequence?

For example, if P is the conjunction of requirements of `Totally_ordered` and Q is conjunction of requirements of `Equality_comparable`, is $P \vdash Q$ a valid statement? Yes. Whenever P is true, it must also be the case that Q is true. Therefore P subsumes Q, and any declaration requiring only `Totally_ordered` is more constrained than a similar declaration requiring only `Equality_comparable`.

To solve this problem, both P and Q must be decomposed into lists of atomic propositions. This is achieved by the application of the left and right logical rules of sequent calculus. The result is a list of subproblems, each having the form $P_i \vdash Q_i$ where P_i is a list of atoms derived from P and Q_i is a list of atoms derived from Q. A subproblem is valid when any atom in Q_i matches another in P_i.

Atomic propositions are compared syntactically; two propositions match iff they have the same spelling.

The initial statement $P \vdash Q$ is valid only when all subproblems are valid. If valid, then P subsumes Q. If the statement is not valid, the P does not subsume Q.

We can also determine if Q subsumes P. Note that if both $P \vdash Q$ and are $Q \vdash P$ valid, then $P = Q$. That is, the constraints are equivalent.

Formalism aside, the algorithms are easily implemented and efficiently executed. Although from the user's perspective, it may be easier to think of the problem as it relates to sets rather than formal logic.

4.2 Declarations, Redeclarations, and Overloading

Constraints are a part of a declaration, and that affects the rules for declarations, definitions, and overloading.

First, any two declarations having the same name, equivalent types, and equivalent constraints declare the same element. For example:

```
template<Floating_point T>
class complex; // #1

template<typename T>
  requires Floating_pont<T>()
class complex; // #2

template<typename T>
  requires is_same<T, float>::value
        || is_same<T, double>::value
        || is_same<T, long double>::value
class complex; // #3
```

The first two declarations introduce the same type, since the shorthand constraint in #1 is equivalent to writing #2. If `Floating_point` is defined as a disjunction of same-type constraints, then all three declarations would introduce the same type since their sets of propositions are the same.

This holds for functions as well:

```
template<Totally_ordered T>
const T& min(const T&, const T&); // #1

template<Totally_ordered T>
const T& min(const T& a, const T& b) { ... } // #2
```

Here, #2 gives a definition for the function declaration in #1.

When two functions have the same name and type but different constraints, they are overloads.

```
template<Input_iterator I>
ptrdiff_t distance(I first, I last); // #1

template<Random_access_iterator I>
ptrdiff_t distance(I first, I last); // #2

int* p = ...;
int* q = ...;
auto n = distance(p, q);
```

When `distance` is called, the compiler determines the best overload. The process of overload resolution is described in 4.3. In this case, this is determined by the most constrained declaration. Because `Random_access_iterator` subsumes `Input_iterator`, the compiler will select #2.

Defining two functions with identical types and identical constraints is an error.

Classes cannot be overloaded. For example:

```
template<Floating_point T>
class complex; // #1

template<Integral T>
class complex; // Error, redeclaration of #1 with different constraints
```

The reason this is not allowed is that C++ does not allow the arbitrary overloading of class templates. This language extension does not either. However, constraints can be used in class template specializations.

```
template<Arithmetic T>
class complex;

template<Floating_point T>
class complex<T>; // #1

template<Integral T>
class complex;    // #2

complex<int> g; // Selects #2
```

As with function overloads, the specializations are differentiated by the equivalence of their constraints. Choosing among constrained specializations is similar to the selection of constrained overloads: choose the most constrained specialization.

Suppose `Arithmetic` has the following definition:

```
template<typename T>
constexpr bool Arithmetic()
{
  return Integral<T>() || Floating_point<T>();
}
```

The reason that the compiler selects #2 is that a) `int` is not a floating point type, and b) `Integral` subsumes the set of requirements denoted by `Integral<T>() || Floating_point<T>()`.

Note that partial specializations must be more specialized than the primary template (see Section 4.3 for more information). The reason is simply that if this is not the case, then the partial specialization will never be selected.

4.3 Overloading and Specialization

The overload resolution process is extended to support constraints. Briefly, for some call expression, that process is:

1. Construct a set of candidate functions
   - If a candidate is a template, deduce the template arguments.
   - If the template is constrained, instantiate and check constraints.
   - If the constraints are satisfied, instantiate the declaration.
2. Exclude non-viable candidates
3. Select the best of the viable candidates.
   - If there is one viable candidate, select it.
   - If there are multiple viable candidates, select the most specialized.

Constructing the candidate set entails the instantiation of function templates declarations. This is done by first deducing the template arguments from the function arguments. If the template is constrained, then those constraints must also be checked. This is done immediately following template argument deduction. Once all template arguments have been deduced, those arguments are substituted into the declaration's constraints and evaluated as a constant expression. If the constraints expression evaluates to **true**, then the declaration is instantiated (but not the definition).

A (template) function is not a viable candidate if a) template argument deduction fails, b) the constraints are not satisfied, or c) instantiating the declaration results in a substitution failure.

If there are multiple viable candidates in the candidate set, the compiler must choose the most specialized. When the candidates are both template specializations, having equivalent types, we compare the templates to see which is the most constrained.

Consider the following:

```
template<Container C>
void f(const C& c); // #1

template<typename S>
  requires Container<S>() || Range<S>()
void f(const S& s); // #2

template<Equality_comparable T>
void f(const T& x); // #3

...
vector<int> v { ... };
f(v)                // calls #1
f(filter(v, even)); // calls #2
f(0);               // calls #3
```

The first call of `f` resolves to #1. All three overloads are viable, but #1 is more constrained than both #2 and #3. Assuming `filter` returns a range adaptor (as in `boost::filter`), the second call to `f` resolves to #2 because a range adaptor is not a `Container` and `Equality_comparable` is subsumed by `Container<S>() || Range<S>()`. The third call resolves to #3 since `int` is neither a `Container` nor a `Range`.

Selecting partial specializations is a similar process. As with overload resolution, determining which specialization is to be instantiated requires the compiler to:

1. Construct a set of candidate specializations
    - If the candidate is a partial specialization, deduce the template arguments.
    - If the candidate is constrained, instantiate and check the constraints.
    - If the constraints are satisfied, instantiate the non-deduced specialization arguments
2. Exclude non-viable candidates
3. Select the best viable specialization
    - If there is one one viable candidate, select it.
    - If there are multiple viable candidates, select the most specialized.

When collecting candidates for instantiation, the compiler must determine if the specialization is viable. This is done by deducing template arguments and checking that specializations constraints.

A specialization is not viable if template argument deduction fails, constraints are not satisfied, or the instantiation of the non-deduced arguments results in a substitution failure.

If there are multiple viable specializations, the compiler must select the most specialized template. When no other factors clearly distinguish two candidates, we select the most constrained, exactly as we did during overload resolution.

For example, we can implement the `is_signed` trait using constraints.

```
template<typename T>
struct is_signed : false_type { };

template<Integral T>
struct is_signed<T> : integral_constant<bool, (T(-1) < T(0))> { };

template<Floating_point T>
struct is_signed<T> : true_type { };
```

The definitions corresponding to `Integral` and `Floating_point` types are partial specializations of the primary template. Note that they are also more specialized, since any constrained template is more constrained than an equivalently

typed unconstrained template. Because of this, the instantiation of this trait will always select the correct evaluation for its type argument. That is, the result is computed for integral types, and trivially `true` for floating point types. For any other type, the result is `false`.

4.4 Non-Type Constraints

Thus far, we have only constrained type arguments. However, predicates can just as easily be used for non-type template arguments as well.

For example, in some generic data structures, it is often more efficient to locally store objects whose size is not greater than some maximum value, and to dynamically allocate larger objects.

```
template<size_t N, Small<N> T>
class small_object;
```

Here, `Small<N>` is just like any other type constraint except that it takes an integer template argument, `N`. The equivalent declaration written using a `requires` clause is:

```
template<size_t N, typename T>
  requires Small<T, N>()
class small_object;
```

The constraint is `true` whenever the `sizeof T` is smaller than `N`. It could have the following definition:

```
template<typename T, size_t N = sizeof(void*)>
constexpr bool Small()
{
  return sizeof(T) <= N;
}
```

The parameter `N` defaults to `sizeof(void*)` . Default arguments can be omitted when using shorthand. We might, for example, provide a facility for allocating small objects:

```
template<Small T>
class small_object_allocator { ... };
```

Shorthand constraints can also introduce non-type parameters. Suppose we define a `hash_array` data structure that has a fixed number of buckets. To reduce the likelihood of collisions, the number of buckets should be prime. The `Prime` constraint has the following declaration:

```
template<size_t N>
constexpr bool Prime() { return is_prime(N); }
```

Note that the expression `is_prime(N)` does not denote a constraint check since the `is_prime` function takes an argument (it may also be overloaded) so it is an atomic proposition.

The hash table's can declared like this:

```
template<Object T, Prime N>
class hash_array;
```

or equivalently:

```
template<typename T, size_t N>
  requires Object<T>() && Prime<N>()
class hash_array;
```

Because constraints are **constexpr** functions, we can evaluate any property that can be computed by **constexpr** evaluation, including testing for primality. Obviously, constraints that are expensive to compute will increase compile time and should be used sparingly.

Note that the kind of the template parameter `N` is `size_t`, not **typename**. A shorthand constraint declares the same kind of parameter as the first template parameter of the constraint predicate.

The proposed language does not currently support refinement based on integer ranges. That is, suppose we have the two predicates:

```
template<int N>
constexpr bool Non_negative() { return N >= 0; }

template<int N>
constexpr bool Positive() { return N > 0; }
```

Both `N >= 0` and `N > 0` are atomic propositions. Neither constraint subsumes the other, nor do they overlap.

4.5 Template Template Parameters

Template template parameters may both use constraints and be constrained. For example, we could parameterize a `stack` over an object type and some container-like template:

```
template<Object T, template<Object, Allocator>> class Cont>
class stack
{
  Cont<T> container;
};
```

Any argument substituted for the `Cont` must have a conforming template "signature" (same number and kinds of parameters) and also be at least as constrained as that parameter. This is exactly the same comparison of constraints used to differentiate overloads and partial specializations. For example:

```
template<Object T, Allocator A>
class vector;

template<Regular T, Allocator A>
class my_list;

template<typename T, typename A>
```

```
class my_vector;

stack<int, vector> a;      // OK: same constraints
stack<int, list> b;        // OK: more constrained
stack<int, my_vector> c;   // Error: less constrained.
```

The `vector` and `list` templates satisfy the requirements of `stack Cont`. However, `my_vector` is unconstrained, which is not more constrained than `Object<T>() && Allocator<T>()`.

Template template parameters can also be introduced by shorthand constraints. For example, we can define a constraint predicate that defines a set of templates that can be used in a policy-based designs.

```
template<template<typename> class T>
constexpr bool Checking_policy()
{
  return is_checking_policy<T>::value;
}
```

Below are the equivalent declarations of a policy-based `smart_ptr` class using a constrained template template parameter.

```
// Shorthand
template<typename T, Checking_policy Check>
class smart_pointer;

// Explicit
template<typename T, template<typename> class Check>
  requires Checking_policy<Check>()
class smart_pointer;
```

This restricts arguments for `Check` to only those unary templates for which a specialization of `is_checking_policy` yields `true`.

4.5.1 Variadic Constraints

Constraints can also be used with variadic templates. For example, an algorithm that computes an offset from a stride descriptor and a sequence of indexes can be declared as:

```
template<Convertible<size_t>... Args>
void offset(descriptor s, Args... indexes);
```

The name `Convertible<size_t>` is just like a normal constraint. The `...` following the constraint means that the constraint will be applied to each type in the parameter pack `Indexes`. The equivalent declaration, written using a `requires` clause is:

```
template<typename... Args>
  requires Convertible<Args, size_t>()...
void offset(descriptor s, Args... indexes);
```

The meaning of the requirement is that every template argument in the pack `Args` must be convertible to `size_t`. When instantiated, the argument pack expands to a conjunction of requirements. That is, `Convertible<Args, size_t>()...` will expand to:

```
Convertible<Arg1, size_t>() && Convertible<Arg2, size_t>() && ...
```

For each $\texttt{Arg}_i$ in the template argument pack `Args`. The constraint is only satisfied when every term evaluates to `true`.

A constraint can also be a variadic template. These are called *variadic constraints*, and they have special properties. Unlike the `Convertible` requirement above, which is applied to each argument in turn, a variadic constraint is applied, as a whole, to an entire sequence of arguments.

For example, suppose we want to define a slicing operation that takes a sequence of indexes and `slice` objects such that an index requests all of the elements in a particular dimension, while a `slice` denotes a sub-sequence of elements. A mix of indexes and slices is a "slice sequence", which we can describe using a variadic constraint.

```
template<typename... Args>
constexpr bool Slice_sequence()
{
  return is_slice<Args...>::value;
}
```

It is a variadic function template taking no function arguments and returning `bool`. The definition delegates to a metafunction that computes whether the property is satisfied.

Our function that computes a matrix descriptor based on a slice sequence has the following declaration.

```
template<Slice_sequence... Args>
descriptor sub_matrix(const Args&... args);
```

Or equivalently:

```
template<typename... Args>
  requires Slice_sequence<Args...>()
descriptor sub_matrix(const Args&... args);
```

Note the contrast with the `Convertible` example above. When the constraint declaration is not variadic, the constraint is applied to each argument, individually (the expansion is applied to constraining expression). When the constraint is variadic, the constraint applies to all of the arguments together (the pack expansion is applied directly to the template arguments).

4.6 Constrained Members

Member functions, constructors, and operators can be constrained and overloaded just like regular function templates. For example, the constructors of the `vector` class are declared like this:

```
template<Object T, Allocator A>
class vector
{
  requires Movable<T>()
    vector(vector&& x); // Move constructor

  requires Copyable<T>()
    vector(const vector& x); // Copy constructor

  // Iterator range constructors
  template<Input_iterator I>
    vector(I first, I last);

  template<Forward_iterator I>
    vector(I first, I last);
};
```

Each constrained member has as its requirements, the conjunction of all constraints in the enclosing scope (all enclosing scopes) in addition to the member-specific constraints introduced by the **requires** clause.

Definitions could be written outside the class declaration by re-stating the requirements. For example:

```
template<Object T, Allocator A>
  requires Copyable<T>()
vector<T, A>::vector(const vector& x) { ... }

template<Object T, Allocator A>
template<Input_iterator I>
vector<T, A>::vector(I first, I last) { ... }
```

The out-of-class declarations are matched against the members declared inside the class based on their types and constraints.

4.7 Writing Requirements

This proposal includes new syntax directly aimed at the specification of syntactic requirements in the form of the **requires** expression. A **requires** expression is a constant, Boolean expression that introduces a conjunction of syntactic requirements. For example, `Totally_ordered` could be defined as:

```
template<typename T>
constexpr bool Totally_ordered()
{
  return Equality_comparable<T>()
      && requires (T a, T b) {
           bool = {a < b};
           bool = {a > b};
           bool = {a <= b};
           bool = {a >= b};
         };
```

```
}
```

The **requires** clause introduces local parameters `a` and `b`, which can be used as notation in nested syntactic requirements. The requires clause is followed by an enclosing block: a sequence of syntactic requirements.

A *syntactic requirement* introduces a conjunction of requirements involving valid expression. Syntactic requirements are interpreted as Boolean expressions; they evaluate to **true** or **false**.

The full range of syntactic requirements is summarized below. Here `e` is an expression, and `T` is a type.

```
e;                      // e is a valid expression
constexpr e;            // e is a valid constant expression
e noexcept;             // e does not propagate exceptions

T = {e}                 // decltype(e) is convertible to T
T == {e};               // decltype(e) is the same as T
constexpr T == {e};     // e is a constant expression
T == {e} noexcept;      // e does not propagate exceptions
```

An expression by itself is simply a requirement for a valid expression: an expression that must compile when instantiated. If the expression cannot be instantiated (due to a substitution failure), the requirement evaluates to **false**.

Writing **constexpr** before `e` requires that `e` must be compile-time evaluable. Writing `noexcept` after `e` indicates that `e` must not propagate exceptions.

A syntactic requirement of the form `T = e` adds the requirement that **decltype**`(e)` must be convertible to `T`. A syntactic requirement of the for `T == e` requires that **decltype**`(e)` is the same as `T`. The **constexpr** and `noexcept` keywords can be used in conjunction with result type conversions as well.

Note that valid expressions written as syntactic requirements are never evaluated at runtime. They are only instantiated to determine validity.

A **requires** expression can also introduce *associated type requirements*, either as member types or alias templates. Associated type requirements are introduced by the **typename** keyword. They can be written as:

```
typename A<T>;
typename T::x;
```

The requirement **typename** `X<T>` requires the alias `A` to compile when instantiated with a template argument `T`. The requirement evaluates to **true** when this is the case.

Note that requiring the valid instantiation of a class template is not currently supported. Unfortunately, the instantiation of a class template may fail in a different instantiation context, resulting in unexpected compiler errors.

The requirement **typename** `T::x` requires `T` to define a nested type named `x`. The requirement is true when instantiation succeeds (i.e., does not result in a substitution failure).

4.8 Designing Concepts

Constraints provide use-site checking for template arguments, which is only one part of what a full definition of concepts will do. But constraints are an important stepping stone in that direction. They provide a basis for experimenting with the required interfaces of concepts moving forward. In this section, we discuss what makes a good concept based on our experience from our concept design experience [1, 8], and our work with constraints.

4.8.1 Intensional and Extensional Definitions

The first observations we make is that good concepts are defined *intensionally* by specifying all of the properties that are required to model that concept. Concepts defined in this way can be readily refined to define more specialized abstractions simply by adding more requirements.

The opposite is to define concepts *extensionally* by providing a list of types known to be models of that concept. Many of the type traits in the Standard Library are defined extensionally (e.g., `Integral`, `Floating_point`). These extensional definitions are rigid and difficult to extend. For example, the `Arithmetic` constraint in our implementation is defined as:

```
template<typename T>
constexpr bool Arithmetic()
{
  return Integral<T>() || Floating_point<T>();
}
```

However, any program that would wish to use `complex<double>` with an `Arithmetic` algorithm would be unable to do so. The programmer would have to create a new constraint and define a new algorithm (probably with the same syntax) to accommodate `complex` numeric types.

We note that the design of effective concepts for mathematic structures has not proven to be an easy task and requires a good understanding of abstract algebra.

4.8.2 Expressivity

There has been an unfortunate tendency in the generic programming community over the past decade to reduce the requirements of algorithms to a minimal kernel of valid expressions. For example, an algorithm comparing values for equality, or more specifically its inverse, must be written as `!(a == b)` instead of the more natural expression `a != b`.

This reduction has been made for the sake of users, so they don't have to implement the full set of overloads for inherently related operations, only ==, <, +=, etc. But this reduction has also been made at the expense of expressiveness and specifiability within templates. Library implements may not be able to use syntax that is natural to the expression of an algorithm, and its requirements must be made in terms of the least syntactic units.

In *Elements of Programming*, Stepanov and McJones state that a computational basis must be efficient and expressive [5]. This ideal was used throughout the design of concepts in both [8] and [1]. Concepts must not be reduced to the least syntactic requirement of a set of related operations.

We think it is both possible and desirable to have expressive concepts and also a mechanism for simplifying implementations. We have not yet thoroughly investigated how such a mechanism might be provided, but we see it as being separate from the design and specification of concepts.

5 Implementation

We have implemented the most features as a branch of GCC 4.8. A few features are currently still incomplete or being refined. In particular, we are revamping the implementation to use the new **requires** expression

In this section, we describe the implementation and some extensions we have provided to simplify the writing of constraints.

5.1 Compiler Support

One of the goals of this implementation is to decrease compile times by providing facilities to help eliminate complexity in the definition of type traits and constraints. In particular, we have internalized several of the Standard type traits to support efficient computation and extensions to the logical rules for constraints.

The `__is_same` intrinsic is a compiler implementation of the `is_same` type trait. This helps reduce the number of template instantiations and specializations needed to define type traits. For example, our definition of `is_floating_point` is:

```
template<typename T>
struct is_floating_point
  : integral_constant<bool,
      __is_same(T, float)       ||
      __is_same(T, double)      ||
      __is_same(T, long double)
    >
{ };
```

The `_is_convertible_to` trait determines whether a user-defined conversion sequence can be found from one type `T` to another `U`. Currently, this is implemented as a library feature with a complex implementation.

These traits, together with the `__is_base_of` intrinsic, have special logical rules in our implementation of the subsumes algorithm. In particular, the following statements are always valid:

- `__is_same(T, U)` $\vdash$ `__is_convertible_to(T, U)`
- `__is_same(T, U)` $\vdash$ `__is_base_of(U, T)`
- `__is_base_of(U, T)` $\vdash$ `__is_convertible_to(T, U)`

That is, whenever the left side of $\vdash$ is **true**, the right side is also **true**. This has the nice property of ordering these relationships by their "strength". These properties are used in some STL implementations to select optimized algorithms when comparing values of the same type vs. those that are simply convertible.

5.2 Syntactic requirements

Suntactic requirements are represented as a conjunction of intrinsic expressions. Valid expression requirements is represented using the `__is_valid_expr` intrinsic, and associated type requirements are represented using the `__is_valid_type` intrinsic.

For example, this sequence of requirements

```
requies (T a, T b) {
  typename T::additive_id;
  T = {a + b} noexcept;
}
```

is decomposed into the following conjunction of atoms:

```
   __is_valid_type(typename T::additive_id)
&& __is_valid_expr(a + b)
&& __is_convertible_to(decltype(a + b), T)
&& noexcept(a + b);
```

Note that the primitives are currently exposed and available to be used wherever a **`requires`** expression can be used. As of the time of writing, support for the **`requires`** expression has not yet been implemented. It is highly likely that these intrinsics will not be available in the future.

Checking for **`constexpr`** requirements has not yet been implemented.

5.3 Standard Library

With our implementation, we have also introduced constraints to a small subset of the standard library. This is not a straightforward proposition because virtually every component in the standard library is a template. This section serves primarily to document our experience with these constraints. The declarations and constraints described herein should not be considered as part of this proposal.

We modified the `<type_traits>`, `<iterator>`, and `<algorithm>` headers to include new constraint definitions and applied them to the required interfaces in those modules. Details and discussion follow.

5.3.1 Type Traits

There are two major changes to the this module. First, we rewrote all of the standard type trait implementations to use intrinsics and constraints where possible, and replaced the use of logical metafunctions with the usual logical C++ operators. The goal is to reduce complexity and compile times. The result is about 25% less code. We haven't measured compile-times yet, but we expect a reasonable improvement due to the smaller number of instantiations required to evaluate those properties.

Second, we added constraint predicates for all of the unary type predicates, and aliases for many of the type transformations (this is a work in progress). The constraint predicates allow the use of standard type traits as constraints:

```
template<Floating_point T>
class complex;
```

Some of the type properties, especially those related to construction and destruction have been implemented in a way that supports ordering for overload resolution. In particular, it must be the case that all constructible types are destructible, and that all copy operations are also valid move operations. In the latter case, this means that copy constructible and copy assignable types are also move constructible and move assignable, respectively.

We also added constraints for the foundational and function concepts found in [1]. In the <type_traits> header, this includes:

- Equality_comparable
- Totally_ordered
- Movable
- Copyable
- Semiregular
- Regular
- Function
- Regular_function
- Predicate
- Relation

Their definitions follow from those given in [1].

5.3.2 Iterator

The iterator header is extended with new constraints and aliases. The aliases provide access to the associated types of an iterator. There are three:

- Iterator_category
- Value_type
- Difference_type

The pointer type and reference type are not used in this iterator design; neither is the iterator_traits traits class. The reason is that the reference type is always the same as decltype(*i), and the pointer type is never needed by any standard algorithms. The use of auto further reduces the need for these names.

For reference, the implementation of Value_type is:

```
template<typename T>
struct __value_type
{ using type = __subst_fail; };

template<typename T>
  requires __is_valid_type(typename T::value_type)
struct __value_type<T>
{ using type = typename T::value_type; };

template<typename T>
  requires __is_valid_type(typename T::value_type)
        && __is_same(typename T::value_type, void)
struct __value_type<T>
{ using type = __subst_fail; };

template<typename T>
struct __value_type<T*>
{ using type = T; };

template<typename T>
struct __value_type<const T*>
{ using type = T; };

template<typename T>
using Value_type = typename __value_type<T>::type;
```

We need a specialization of `__value_type` to accommodate the fact that the `output_iterator` template sets the value type to `void`. This prevents substitution failures when writing type names like, `const Value_type<I>&`.

The `__subst_fail` type indicates substitution failure. Determining whether `Value_type<I>` is defined for requires us to test that the alias is not a name for `__subst_fail`. For example, the `Input_iterator` constraint includes this test:

```
!__same(Value_type<I>, __subst_fail).
```

There are a number of support constraints in the library module. Most of these are defined in [1].

- `Readable`
- `Writable`
- `Permutable`
- `Mutable`
- `Advanceable` (was `WeaklyIncrementable`)
- `Incrementable`

The standard iterator hierarchy is unchanged.

- `Input_iterator`
- `Output_iterator`
- `Forward_iterator`
- `Bidirectional_iterator`
- `Random_access_iterator`

The design in [1] did not include an output iterator. In truth, the supporting concepts (esp., `Writable` and `Advanceable`) largely eliminate the specific need for the concept. However, we have retained it in this design for parity with input iterators.

5.3.3 Algorithm

We constrained all of the standard algorithms, except those taking random number generators as arguments. To help simplify the constraints, which can get fairly verbose, we introduced a number of algorithmic abstractions. Some of these were used in [1], others are new.

- `Indirectly_movable`
- `Indirectly_copyable`
- `Indirectly_swappable`
- `Indirectly_equal`
- `Indirectly_ordered`
- `Indirectly_comparable`
- `Sortable`
- `Mergeable`

The "indirectly" constraints describe operations between two pairs of differently typed iterator parameters (e.g., copying between iterators, comparing elements of two iterators for equality). The names might be improved; these are primarily intended for convenience. We are not proposing that they should be part of the Standard Library.

6 Extensions

The following sections describe features that we have considered, but have not yet implemented, or have just begun to experiment with.

6.1 Concepts

Eventually, we hope to see a full concept design, with full checking of template bodies and semantics. For now, we are convinced that it can be done (see [1]), but do not have a complete design.

6.2 Terse Templates

We consider generic lambdas and constraints very closely related. In this section, we present an extension for "terse templates", an extension of the proposal aimed at providing a uniform notation for constraining both templates and generic lambdas.

The purpose of this extension is to provide a very terse notation to constrain simple templates and lambdas. This design aims for minimalism, not completeness. If you want completeness, write a template. We see this as an exercise in making simple things simple.

The aim is to come up with a uniform notation that can be used to constrain both templates and generic lambdas. For lambdas, we are convinced that we want a terse syntax, primarily for relatively simple sets of template argument types. We would hate to see an "unconstrained lambda" subculture grow up as a result of a clumsy syntax for constraints. It is this potential problem with generic lambdas that caused us to move ahead now (and encouragement from the SG8).

6.2.1 The basics

We use `Number` as an example of a constraint or concept. We can define a constrained lambda like this:

```
[](Number n) ...
```

This means that `Number` must be a type for which some numeric constraint is true; `n` is a constrained parameter. Let us first explain how this works for templates, and then come back to lambdas.

For a template, we can write:

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

Here, `Cont` must be known to be a constrained parameter, and this is shorthand notation for

```
template<Container Cont>
void sort(Cont& c);
```

which again is a shorthand for

```
template<typename Cont>
   requires Container<Cont>()
void sort(Cont& c);
```

Somewhere, somehow (described below), `Cont` must be defined to be the name of a type than satisfies the constraint/concept `Container`.

We note that when programmers first see something new, they clamor for "heavy" syntax, such as the last example. Later, they complain about verbosity, and prefer the terser forms. Later generations of programmers typically fail to understand why the long form exists at all. If you like the "heavy syntax", you can simply use that.

6.2.2 Type compatibility

What if we need two argument types of the same concept? Consider

```
void sort(Ran p, Ran q);
```

For this to make sense, `p` and `q` must be of the same type, and that is the rule. By default, if you use the same constrained parameter name for two arguments, the types of those arguments must be the same. We chose to make repeated use of a constrained parameter name imply "same type" because that (in most environments) is the most common case and the aim here is to optimize for terse notation of the simplest case. Also, a constrained parameter is the name of a type, and having two type names refer to different types in the same scope would cause chaos.

When we need to make the requirement for commonality explicit, we do so by introducing a name for a type that must meet a concept. For example:

```
using Randon_access_iterator{Ran};
// ...
void sort(Ran p, Ran q);
```

This *using-declaration* introduces the constrained parameter name `Ran` and associates it with the constraint named `Random_access_iterator`. When `Ran` is used as a parameter of a function, the function is a function template using the terse form, and a function argument used in a call of the must be of a type that satisfies `Ran`'s concept, here `Random_access_iterator`. The "long hand" declaration of `sort` could be equivalently written:

```
template<typename Ran>
  requires Random_access_iterator<Ran>
void sort(Ran p, Ran q);
```

To our eyes, this latter version is beginning to look verbose. Note that we can place the *using-declaration* for `Ran` in a namespace and/or as part of a library header to avoid namespace pollution and gain uniformity of use across the library.

Now, what do we do when we want two argument types of the same concept that may differ? Consider `merge()`:

```
template<typename For, typename For2, typename Out>
void merge(For p, For q, For2 p2, For2 q2, Out p);
```

Merge has been a long-standing example of (necessary) complexity of specification, but the mechanism for naming types that meet a concept, handles this:

```
using Forward_iterator{For};
using Forward_iterator{For2};
using Output_iterator{Out};
// ...
void merge(For p, For q, For2 p2, For2 q2, Out p);
```

`For` and `For2` are simply different names for types that must meet the `Forward_iterator` concept but types that match different names may differ.

We can do better still. In reality, the three template argument types for this `merge()` are not independent, but must meet some fairly intricate constraints that can be expressed as a concept taking three type arguments. For details, see the Palo Alto TR. At a minimum, we need to write:

```
template<Forward_iterator For, Forward_iterator For2, Output_iterator Out>
  requires Mergeable<For,For2,Out>
void merge(For p, For q, For2 p2, For2 q2, Out p);
```

Using the terse syntax, this becomes:

```
using Mergeable{For,For2,Out};
// ...
void merge(For p, For q, For2 p2, For2 q2, Out p);
```

There are five algorithms in the STL that requires `Mergeable`, so this notation is even more economical than it appears when used in a single example.

The syntax of this *using-declaration* is:

```
using name-of-concept { identifier-list };
```

So, we have a very terse syntax that minimizes repetition and can distinguish between two uses of the same type of a concept and (potentially) two different type of a concept. In case, you lost track of the reason for going in this direction: This terse syntax can be used for lambdas, where the old (verbose) syntax is either impossible or so verbose that it would encourage the use of unconstrained lambdas. For example:

```
[] (For p, For q, For2 p2, For2 q2, Out p) { /* do some merging ... */ }
```

6.2.3 The "Dual Name" Problem

We used `Ran` for a constrained parameter name for `Random_access_iterator`s, For for `Forward_iterator`s, and `Number` for some numeric concept. We actually like such short names, but they will have to be in namespace scope to be usable for many terse templates and not everybody likes short names. It seems that we need two names for every concept: one for the concept/constraint itself plus

one for its constrained parameter name. It would be nice if the constrained parameter names were conventional and standard, so that when we see one it is obvious what it is. We struggled trying to find a suitable naming scheme. Consider

1. Randon_access_iterator Ran first letters)
2. Randon_access_iterator RandomAccessIterator (CamelCase)
3. Random_access_iterator Random_access_iterator1 (index)

After trying those, and more, we hated them all. Our solution is that the name of a constraint doubles as the name of its constrained parameter. This then implies that a constraint that is to be used for the terse syntax must be syntactically distinguished. In other words, it must be defined to be a concept using a special syntax or a keyword. We use **concept**.

From comments on the reflectors and in the phone meeting, we are under the impression that many would like to have at least some constraints explicitly declared to be concepts anyway, so this may be a popular choice.

So, when used as a type in a function declaration, the name of a concept denotes its constrained parameter. For example, we can write

```
template<class T>
concept bool Number() { /* what it takes to be a number */ }
```

Simply replacing **constexpr** with concept and allowing a bit of extra checking. Give that we can write

```
[](Number n) ...

void f(Number n) ...
```

as desired and without introducing new syntax, potentially confusing dual names, or conventions.

From a language-technical point of view, the dual name solution is marginally simpler, but from a user's point of view, having to introduce a second name only when a second name is needed (e.g., `For2` above) or wanted for terse notation (e.g., `For` above) is far more convenient.

6.2.4 Technicalities

This simple design leaves many questions for language experts, such as

- How do we define a constraint so that it is recognized as a concept?
- How do we define two names for a concept?
- How do we define return values that depend on a template argument?

The answers are essential, but not fundamental as long as we preserve the simplicity of use. We are not married to any particular syntax.

6.3 How do we recognize a parameter type?

We suggest that a `constexpr` function declared with concept as its keyword rather than `constexpr` is recognized as a parameter type in the implicit template shorthand syntax. For example

```
template<typename T> constexpr bool One(); // a constraint
template<typename T> concept bool Two(); // a concept

void algo1(One t) ...
void algo2(Two t) ..
```

We could shorten the definition of `Two()` by letting `concept` mean `constexpr bool`, but left such further elaboration to the future. I note that we reserved concept as a keyword for future use.

6.4 How do we define two names for a concept?

We need to define an alias for a concept that is recognized as distinct by the terse template syntax. For example:

```
Using Forward_iterator{For};
```

This would be sufficient to make the merge example work. I don't think we need anything more elaborate.

6.5 What about return types?

Expressing a return type without access to template arguments or function arguments can be rather tricky. For example:

```
Forward_iterator find(Forward_iterator p, Forward_iterator q,
Equality_comparable<Value_type<Forward_iterator> v);
```

Here, `Forward_iterator` is used as a return type before it is recognized as a parameter type. This might work, but I suspect problems, notably lookup problems. This was one problem that stumped us years ago. However, Daveed pointed out that now we have the suffix return type notation, so let's use it

```
auto find(Forward_iterator p, Forward_iterator q,

Equality_comparable<Value_type<Forward_iterator> v) -> Forward_iterator;
```

But Wait! C++14 will almost certainly allow deduction of the return value so we can write

```
auto find(Forward_iterator p, Forward_iterator q,
Equality_comparable<Value_type<Forward_iterator> v)
{
  // ...
}
```

This reduces the problem to a previously solved one. If you don't like the shorthand notation or don't like the suffix return type syntax, don't use them. This is all shorthand.

If you like short names, you can get:

```
auto find(For p, For q, Equality_comparable<Value_type<For> v)
{
  // ...
}
```

6.6 Constraints on Non-templates

It should also be possible to add constraints to non-template declarations, although this is purely speculative discussion. One use of this feature would be to conditionally define functions based on the compiler vendor.

```
requires Compiler == MSVC
  string get_type_name();

requires Compiler == GCC
  string get_type_name();
```

Since the constraints are non-dependent, they should be evaluated prior to the parsing of the declaration. It is unclear whether the declaration should be parsed or not if the constraints are not satisfied. Reasonable arguments could be made for either choice. The definition should not be parsed, however.

It's not hard to imagine the application of constraints to virtually any declaration (e.g., aliases, namespaces, variables, etc). This may prove to be a rich source of new features for C++. However, careful consideration must be given to the semantics of those constraints and how they can be used in a principled way. We would like to avoid the use of constraints as a low-level mechanism for controlling compilation.

7 Standard Wording

The proposed standard wording covers only template constraints. It does not describe the features discussed in Section 6.

7.1 5.1.1 [expr.prime.general]

primary-expression::
literal
`this`
...
requires-expression

7.2 5.1.3 [expr.prim.requires]

A requires expression provides a concise way to define syntactic requirements for template constraints. [*Example:*

```
template<typename T>
constexpr bool Readable() {
  return requires (T i) {
    typename Value_type<T>;
    const Value_type<T>& = {*i};
  };
}
```

— *end example*]

requires-expression:
`requires` *parameter-declaration-clause*$_{opt}$ `{` *requirement-list* `}`

requirement-list:
requirement-statement$_{opt}$
requirement-list `;` *requirement-statement*$_{opt}$

requirement-statement:
syntactic-requirement
associated-type-requirement

syntactic-requirement:
`constexpr`$_{opt}$ *valid-expression-requirement* `noexcept`$_{opt}$

valid-expression-requirement:
expression
type-id `(` *expression* `)` *type-id* `=` `{` *expression* `}`
type-id `==` `{` *expression* `}`

associated-type-requirement:
`typename` *type-id*

A *requires-expression* shall only appear inside a template.

The type of a *requires-expression* is **bool**; it is a constant expression.

The *requires-expression* may be introduce local objects via a *parameter-declaration-clause*. These parameters have no linkage, storage, or lifetime; they are used as notation for the purpose of writing requirements.

The body of *requires-expression* is a conjunction of requirements. A *requirement-statement* introduces one or more requirements, also as a conjunction.

The presence of **constexpr** in a syntactic requirement that precedes a *valid-expression-requirement* denotes the requirement that the expression must be compile-time evaluable. The presence of `noexcept` following a *valid-expression-requirement* denotes the requirement that the expression must not propagate exceptions.

A *valid-expression-requirement* started by a *type-id* denotes an additional type requirement. The use of parentheses indicates that the specified *type-id* must be constructible from the result of the required *expression*. The = denotes a requirement that the **decltype** of the required expression is convertible to the *type-id*. The == denotes a requirement that the **decltype** of the required expression is the same type as the *type-id*.

A substitution failures occurring from the instantiation of a *requires-expression* expression causes the entire expression to evaluate to **false**. If instantiation succeeds, the result of the *requires-expression* is the result of the conjunction of sub-requirements. The *expression* in *valid-expression-requirement* is never evaluated.

7.3 14 [temp]

A template defines a family of classes or functions or an alias for a family of types.

template-declaration:
`template` < *template-parameter-list* > *requires-clause*$_{opt}$ *declaration*

[*Note:* ... — end note]

7.4 14.2 [temp.param]

The syntax for *template-parameter*s is:

template-parameter:
type-parameter
parameter-declaration
constrained-template-parameter

constrained-template-parameter:
    constraint-id `...`$_{opt}$ *identifier*
    constraint-id `...`$_{opt}$ *identifier* `=` *constrained-default-argument*

constraint-id:
    identifier simple-template-id

constrained-default-argument:
    type-id
    template-name
    expression

A *constrained-template-parameter* is introduced by a *constraint-id*.

A *constraint-id* refers to a *constraint predicate* 7.8. The template parameter introduced by the constraint has the same properties as the first template parameter of the constraint. The id is used to construct a requirement on the introduced parameter, `X`.

- If the *constraint-id* is an *identifier*, `C`, the constructed requirement is `C<X>()`.
- If the *constraint-id* is a *simple-template-id*, `C<Args>`, then requirement `C<X, Args>()`.

The kind of *constrained-default-arg* shall match the kind of parameter introduced by the *constrained-id*.

7.5 14.5.6 Template template arguments [tmp.arg.template]

(3) A *template-argument* matches ..., and the `P` is more constrained than `A`.

7.6 14.5.6 Function templates [tmp.func]

(6) Two function templates are *equivalent* when ..., and have equivalent constraints.

7.7 14.5.6.2 Partial ordering of function templates [tmp.func]

(5) If neither template is more specialized than the other based on template argument deduction, the most specialized template is the most constrained 7.8.2.

7.8 14.9 [temp.cons]

If a *requires-clause* is present, the template is a *constrained* template. Otherwise, it is an *unconstrained template*.

requires-clause:
constant-expression

A *requires-clause* introduces template constraints in the form of a **bool** *constant-expression*. A constrained template can only be instantiated when the constraints are satisfied.

If the *requires-clause* contains a call to *constraint predicate*, the body of that function is expanded, in-line into the required *constant-expression*. No other call expressions are expanded.

A *constraint predicate* is a function template that:

- is **constexpr**,
- returns **bool**,
- has function arguments, and
- has exactly one definition for all type arguments.

[*Note: One purpose for inlining requirements is that syntactic requirements can be lifted out of a function definition, and bringing them into the local instantiation context. Constraint predicates are inherently SFINAE-friendly. — end note*]

7.8.1 14.9.1 Constraint Satisfaction [temp.cons.sat]

A constrained templates constraints are checked just after template argument deduction and before the declaration is instantiated.

The deduced arguments are substituted into the required *constant- expression*. If a substitution failure occurs, the expression evaluates to *false*. Otherwise, the resulting expression is **constexpr** evaluated, returning either **true** or **false**.

If the template constraints are not satisfied, the declaration is treated as if template argument deduction had failed. [*Note: Whether or not this results in diagnostics depends on the context in which instantiation is requested. — end note*]

7.8.2 14.9.3 Constraint Relations [temp.cons.rel]

To determine if one declaration is more constrained than another, we determine if its constraints *subsume* the other's.

A constraint P subsumes a constraint Q iff $P \vdash Q$ is logically valid statement.

[*TODO: Write a formal definition of subsumes*].

An unconstrained template has an empty set of constraints. [*Note: This implies that an constrained template is always more constrained than an unconstrained template — end node*].

A constrained template `T` is *more constrained* than a similarly typed template `S` iff `T`'s constraints subsume those of the `S`.

Two constraints P and Q are equivalent when $P \vdash Q$ and $Q \vdash P$.

Acknowledgements

We are grateful for the input, comments, and corrections from Jason Merril, Greg Marr, Chris Jefferson, Daveed Vandevoorde, Matt Austern, Herb Sutter, Tony Van Eerd, and Michael Lopez

References

[1] Bjarne Stroustrup, Andrew Sutton, *et al.*, *A Concept Design for the STL*, Technical Report N3351=12-0041, ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++, Jan 2012.

[2] Pete Becker, *Working Draft, Standard for Programming Language C++* Technical Report N2914=09-0104, ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++, Jun 2009.

[3] Gabriel Dos Reis, Bjarne Stroustrup, and Alisdair Meredith, *Axioms: Semantics Aspects of C++ Concepts* Technical Report N2887=09-0077, ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++, Sep 2009.

[4] Stephan Du Toit (ed), *Working Draft, Standard for Programming Language C++* Technical Report N3337=12-0027, ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++, Nov 2012.

[5] Alexander Stepanov and Paul McJones, *Elements of Programming*, Addison Wesley, 2009, pp. 250.

[6] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine, "Concepts: Linguistic Support for Generic Programming in C++", *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA'06), Oct 22-26, 2006, Portland, Oregon, pp. 291-310.

[7] Gabriel Dos Reis and Bjarne Stroustrup, "Specifying C++ concepts", In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL'06), Jan 11-13, 2006, Charleston, South Carolina, pp. 295-308.

[8] Andrew Sutton and Bjarne Stroustrup "Design of Concept Libraries for C++" In *Proceedings of the 4th International Conference on Software Language Engineering* (SLE'11), Jul 3-4, 2011, Braga, Portugal, pp. xxx-yyy.

[9] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine, "Concept-Controlled Polymorphism", *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering* (GPCE'03), Sep 22-25, 2003, Erfurt, Germany, pp. 228-244.

[10] Larry Paulson, *ML for the Working Programmer*, Cambridge University Press, 1996, pp. 500.