

Document number: N3701  
Date: 2013-06-28  
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

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 principled, uncomplicated, and uniform. 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, Concepts Lite:

- allows programmers to directly state the requirements of a set of template arguments as part of a template’s interface,
- supports function overloading and class template specialization based on constraints,
- seamlessly integrates a number of orthogonal features to provide uniform syntax and semantics for generic lambdas, `auto` declarations, and result type deduction,
- fundamentally improves diagnostics by checking template arguments in terms of stated intent at the point of use,
- do all of this without any runtime overhead or longer compilation times.

This work is implemented as a branch of GCC-4.9. Previous versions of the compiler are available for download at <http://concepts.axiomatics.org/>. The most recent version provides support for writing and enforcing constraints. Features related to shorthand notation and generic lambdas have not yet been implemented.

This paper is in its 2nd revision. There have been a number of substantial changes, which are summarize here.

- The syntax of constrained member functions has changed.

- The syntax of the requires expression has changed.
- We have introduced syntax for constraining generic lambdas, and using constraints wherever auto is used.
- We allow the use of overload sets as arguments to constrained functions.
- The standard wording has been more extensively developed, but does not yet include language for constrained generic lambdas.

This paper is organized like this:

- Tutorial: introduces the new features specified by Concepts Lite, their core concepts and examples.
- User's guide: provides a more extensive discussion of the proposed features and demonstrates the completeness of the constraints mechanism. We also include some high-level discussion of the language mechanics.
- Discussion: explains what constrains are not. In particular, we try to outline constraint's relation to concepts and to dispel some common misconceptions about concepts.
- Design Notes: Notes on the design of terse notation for templates and generic lambdas.
- Language definition: presents a semi-formal definition of constraints

## 2 Tutorial

We begin by presenting a tutorial of Concepts Lite and its associated language features. This section covers the core concepts of the proposal: how to constrain templates, what concept definitions look like, and where constraints can be used with other language features.

### 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 constant Boolean expression. Template constraints are simply `constexpr` expressions that are evaluated by the compiler to determine if the following declaration can be used. In fact, the `Sortable` is just a `constexpr` function that returns `true` only when its template argument can be used with the `sort` algorithm.

Trying to use the algorithm with a `list` does not work since `std::sort` is not defined for bidirectional iterators.

```
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>]
note:   failed requirement with ('list<int>::iterator i', std::size_t n)
note:     'i[n]' is not valid syntax

```

Note that this is similar to actual computer output, rather than a conjecture about what we might be able to produce. If people find this too verbose, diagnostics could be suppressed through various compiler options.

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 vectors for equality and ordering when the value type can be

compared for equality or ordering.

```
template<Object T, Allocator A>
class vector
{
    vector(const T& v)
        requires Copyable<T>();

    void push_back(const T& x)
        requires Copyable<T>();
};
```

For constrained member functions, the **requires** clause is written *after* the declaration. Trying to invoke the copy constructor or `push_back` when `T` is a `unique_ptr` will result in a diagnostic at the point of use. This is a significant improvement over C++11 where constraining (non-template) member functions is not possible.

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 to 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].

## 2.2 Defining Constraints

What do these concepts look like? Here is the declaration of `Equality_comparable`.

```
template<typename T>
concept bool Equality_comparable() { ... }
```

A constraint is a function template declared using the declaration specifier, **concept**. The **concept** specifier implies that the declaration is **constexpr**, and it enforces some other restrictions. In particular, concepts must not have function arguments, and they must return **bool**. Concepts are—in the most literal sense—predicates on template arguments, and they can be checked by compile time evaluation, just like any other **constexpr** function.

The definition of the `Equality_comparable` concept relies on a new language feature, the **requires** expression. This provides the capability to succinctly state requirements on valid expressions and associated types.

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

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 requirement `== ba == b -> bool` 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**.

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 requires (T i) {
        typename Value_type<I>;
        {*i} -> const Value_type<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 3.1.2.

Concepts definitions can, naturally, be nested. For example, the `Sortable` concept is simply a conjunction of other concepts or primitive constraints:

```
template<typename T>
concept bool Sortable()
{
    return Permutable_container<T>()
        && Totally_ordered<Value_type<T>>();
}
```

The `Permutable_container` and `Totally_ordered` concepts are also defined in terms of other concepts. Ultimately, these definitions are written in terms of syntactic requirements for valid expressions and associated types.

### 2.2.1 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 3.3.

In this example, the requirements are largely disjoint. It is unlikely that we will find many containers that are both `Sequences` and `Associations`. 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_access_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 3.3.

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 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 seamless.

The same principles apply for the partial specialization of class templates. This can be helpful, for example, in the definition of numeric traits for sets of types:

```
template<typename T>
struct numeric_traits;

template<Integral T>
struct numeric_traits<T> : integral_traits<T>
{ ... };

template<Floating_point T>
struct numeric_traits<T> : floating_point_traits<T>
{ };
```

The `numeric_traits` facility is designed as an undefined primary template, and two constrained partial specializations, each of which corresponds to a distinct set of numeric types.

Note that a specialization must be more specialized than the primary template. This is obviously the case here since the primary template is unconstrained.

## 2.3 More Shorthand

The shorthand notation thus far allows us to use concepts as the “type” of template parameters. This provides a convenient way of concisely stating the requirements for that single template argument. However, when multiple template parameters are involved, the constraints can become unwieldy.

Our experience has been that it is often useful to define concepts for groups of algorithms with related requirements and semantics. This can help reduce redundancy in declarations. For example, `all_of`, `any_of`, `none_of`, `find_if`, `find_if_not`, `count_if`, and `is_partitioned` all share the same template requirements. We can define them as:

```
template<typename I, typename P>
concept bool Input_query()
{
    return Input_iterator<I> && Predicate<P, Value_type<I>>;
}
```

We could use this concept like this:

```
template<typename I, typename P>
    requires Input_query<I, P>()
I find(I first, I last, P pred);
```

But this is a little more verbose than we want. We’d like to have the `Input_query` concept fully specify the template parameters necessary for that the declaration. We can do this by writing the declaration like this:

```
Input_query{I, P}
I find(I first, I last, P pred);
```

Here, the specifier `,PInput_queryI` is called a *concept introduction*. It is simply shorthand for writing the template parameter list and associated `requires` clause in the preceding declaration. The template parameters `I` and `P` and their constraints are used for the declaration following the introduction.

This mechanism is not simply syntactic sugar. It is necessary for writing non-trivial constraints on generic lambdas, which we describe in the following section.

## 2.4 Lambdas and Auto

C++14 will introduce generic lambdas. They allow programmers to write lambdas without specifying argument types. For example, we might search for a C-string in a vector of strings.

```
vector<string> v = {...};
const char* str = "hello";
find_if(v, [str](const auto& x) { return str == x; });
```

The use of a generic lambda prevents a conversion to `string` in each comparison. Obviously, this is a desirable features. However, the lambda is not

defined for every type of `x`, only those for which `str == x` is a valid expression. Essentially, we'd like to say that `x` must be some kind of `String`.

Concepts lite allows us to constrain generic lambdas by using a concept name in place of `auto`. We can rewrite the lambda in the program above like this:

```
[str](const String& x) { return str == x; }
```

This is the most terse way of writing of writing the lambda. Here, `String` must be a *type concept*, a concept that takes a single template argument.

There are, of course equivalent ways of expressing the same thing. For example, we can explicitly declare the constrained type parameter `T` like this:

```
[str]<String T>(const T& x) { return str == x; }
```

Here, `<String T>` is the template parameter list for the lambda expression. We can also choose to introduce the constrained template parameter using the concept introduction mechanism.

```
[str] String{T} (const T& x) { return str == x; }
```

All three declarations are precisely equivalent.

Note that the concept introduction mechanism is the only reliable way of tersely writing constraints for non-trivial generic lambdas. For example, a lambda that shares the same interface as `find_if` might be written thusly:

```
[] Input_query{I, P} (I first I last, P pred) { ... }
```

Using one of the other syntaxes would lead to (far) more verbose code, which is antithetical to the notion of lambdas to begin with.

Note that generic lambdas allow us to implicitly declare templates without first declaring that the following declaration is, in fact, a template. To provide consistency, we allow `auto` (and constrained `auto`) to be used for any function declaration. For example, we could declare `find` like this:

```
auto gcd(auto a, auto b)
```

Of course, no reasonable C++ programmer would declare a function whose interface is completely unspecified, except in the number of arguments. Obviously the algorithm is designed for a specific set of types, say `Integers`. We can use concepts to restrict that definition.

```
Integer gcd(Integer a, Integer b)
```

This is equivalent to writing:

```
template<Integer T>  
T gcd(T a, T b);
```

Note that repeated uses of the same concept name with a scope bind to the type name, whereas the unconstrained `auto` version allows those types to vary. This makes the use of concept names consistent with the usual rules for type names (i.e., they don't change between uses).

To further uniformity of these features, Concepts Lite allows programmers to use the name of a concept *wherever auto can be used in C++*.

For example, we can use concepts with `auto` variable declarations:

```
Real y = f(x);
```

The meaning is that `decltype(y)` must satisfy the requirements of the `Real` concept. This provides more context to a programmer reading the code if. This behavior could be emulated with a `static_assert`.

```
auto y = f(x);
static_assert(Real<decltype(y)>(), "");
```

Concept checking may give better diagnostics than `static_assert`; it depends on the implementation.

We can also use concept names for function result types. For example, the result of `begin` must be an `Iterator`.

```
template<typename T>
Iterator begin(T&& t) -> decltype(t.begin());
```

Or more concisely using result-type deduction:

```
template<typename T>
Iterator begin(T&& t);
```

The result type constraint becomes part of the requirements for the template. It is equivalent to writing this:

```
template<typename T>
requires Iterator<decltype(declval<T>().begin())>()
auto begin(T&& t);
```

Notes on the rationale and design of these syntactic extensions is given in Section 5.

Concepts Lite provides one more feature related to lambdas and constraints. We allow the names of overloaded functions to be used as arguments to a constrained function. For example, if we want to compute the product of a sequence of values using the `accumulate` algorithm, we could simply write it this way:

```
vector<double> v { ... };
Number n = accumulate(v.begin(), v.end(), operator*);
```

The call to `accumulate` is equivalent to writing it and using a generic lambda. That is:

```
Number n = accumulate(v.begin(), v.end(),
    [](auto a, auto b) { return operator*(a, b)});
```

Note that `accumulate` *must* be constrained for this to work because the evaluated expression is deduced from the constraints of the called function. More details are given in Sections 3.6 and 6.

The following sections in this report elaborate on the ideas presented in the tutorial. We describe basic language mechanics, the implementation and provide initial standard wording for the proposed features.

## 3 User's Guide

This section expands on the tutorial and gives more examples of how constraints interact with different 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) { {a < a} -> bool; }
!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
```

In the last two we assume `N` and `X` are also constant expressions. 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.

### 3.1 Anatomy of a Constraint

In this 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 reduce and decompose a constraint expression into sets of atomic propositions. That algorithm is sketched out in Section ??.

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 have no deeper logical structure and cannot be further further reduced or decomposed. Atomic propositions, or simply *atoms*, include things like type traits (`is_integral<T>::value`), relational expressions (`N == 0`), and `constexpr` functions (e.g., `is_prime(N)`). Calls to functions declared with the *concept* specifier are *not* atomic.

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. The `is_integral` trait could have a number of specializations, and

`is_prime` could have different overloads for different types of `N`. Trying to reduce 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 reduced. While negation has turned out to be occasionally useful in the specification of constraints, 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`.

### 3.1.1 Concept Definitions

A concept is a declaration declared with the `concept` specifier. That specifier implies that the declaration is `constexpr`, and imposes the following restrictions:

- The declaration must return `bool`.
- The declaration must have no function parameters.
- The declaration must not be constrained.
- The declaration must also be a definition.
- The definition must not be recursive.
- There must be no overload with the same name and type but different constraints.

The goal of these restrictions is to ensure that each concept has a single definition for all type arguments. That is, there is no way to find an alternative definition of requirements based on the type of a template argument. Allowing concepts to overload like regular functions would make the constraints language undecidable.

Note, however, that concepts 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>
concept bool Equality_comparable() { ... }

template<typename T, typename U>
concept bool Equality_comparable() { ... }
```

### 3.1.2 Writing Requirements

Concept definitions often specify requirements on valid expressions and associated types. These are written using the **requires** expression. A **requires** expression introduces local arguments to help ease the writing of valid expressions. This is no different than the table of names that often precedes tables of requirements in the C++ standard [?] and online library documentation [?]. Those arguments also obviate the need to use `declval` everywhere. Recall the definition of the `Equality_comparable` concept:

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

The arguments `a` and `b` are placeholders for values of type `T`. They are simply used to help the compiler resolve expressions in the individual requirements of the **requires** clause.

There are four kinds of requirements:

- A *simple-requirement* is simply a valid expression that must compile when instantiated.
- A *compound-requirement* is a set of requirements involving a valid expression that includes the validity of an expression, possibly a constraint on the result type, and other specifiers for semantic constraints. The requirements for `==` and `!=` in `Equality_comparable` are compound requirements.
- A *type-requirement* is a requirement for the formation of a valid type.
- A *nested-requirement* is a **requires** clause that evaluates additional constraints within the body of a **requires** expression.

Consider a simple concept definition for `Iterator`; its models must be incrementable and dereferenceable.

```
template<typename T>
concept Iterator()
{
    return requires (I i) {
        {++i} -> I&;
        i++;
        *i;
    };
}
```

The pre-increment requirement is a *compound-requirement*. The expression `++i` must compile when instantiated and the result of that expression must be convertible to `I&`. Note that there is an implied type requirement in the formation of `I&`. This means, for example, that any implementation returning `void` will fail to satisfy the requirements since references to `void` are not valid types.

The requirements for post-increment and dereferencing are *simple-requirements*. These expressions must compile when instantiated. There are no requirements associated with the result of those operations.

Expression requirements can include specifiers, requiring that the expression must not throw, or that the expression must be evaluable at compile-time. For example, we can define a concept for types that can be copied without propagating exceptions.

```
template<typename T>
concept bool Nothrow_copyable()
{
    return requires (T a, T b) {
        { T(a) } noexcept;
        { a = b; } noexcept;
    };
}
```

Similarly, we could define a constraint for types whose interfaces support compile-time evaluation. For example, we could define a constraint that allows the optimization certain container operations when their allocators are `constexpr` evaluable.

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

Both `constexpr` and `noexcept` can be specified. The order is immaterial.

A *type-requirement* requires the valid formation of a type. For example, the `Allocator` concept may include the following:

```
template<typename A>
concept bool Allocator()
{
    return requires () {
        typename A::value_type;
        typename A::pointer;
        typename A::const_pointer;
        ... // more requirements
    };
}
```

If, when instantiated, any of those type names cannot be resolved, the requirements are not satisfied.

Type requirements can also be formed for alias templates:

```
template<typename T>
concept bool Range()
{
    return requires () {
        typename Iterator_type<R>;
    };
}
```

If `Iterator_type<R>` results in a substitution failure, the constraints are not satisfied.

A *nested-requirement* provides the ability to evaluate additional requirements on associated types or result types inside the expression. For example, the `Allocator` concept may include the following:

```
concept bool Allocator()
{
    return requires () {
        typename A::pointer;
        requires Pointer<typename A::pointer>;
        // ...
    };
}
```

The *type-requirement* checks for validity of `A::pointer`, and the *nested-requirement* checks that the type satisfies the requirements of the `Pointer` concept.

This can be cumbersome when specifying requirements on valid expressions, especially when the name of the result type is unimportant to the design of the concept. A concept name may be used in place of a type-id in a *compound-requirement*. For example, the `Forward_iterator` concept might be written as:

```
template<typename I>
concept bool Forward_iterator()
{
    return Input_iterator<I>()
        && requires (I i) {
            {*i} -> Reference;
        };
}
```

The name `Reference` is a concept that is equivalent to `is_reference<T>::value`.

### 3.1.3 Constraint Reduction

Within a constraint, a call to a function that is declared with the `concept` specifier is called a *concept check*. Unlike other `constexpr` functions, concepts are not atomic: their definitions are recursively reduced instead of `constexpr`-evaluated.

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

```

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

```

And its use within an algorithm:

```

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

```

After the template requirements are parsed, the constraints are reduced to an expression of only atoms, connectives. The concept check `Equality_comparable<T>()` causes the function definition of that function to be recursively reduced.

The results of reducing concept checks are equivalent to writing all of the atomic propositions as if they were part of the `requires` clause. For example, if a compiler implementation exposed the mechanism for evaluating these requirements, the declaration above would be equivalent to this:

```

template<typename T>
requires __is_valid_expr(decltype<T>() == decltype<T>())
        && __is_convertible_to(decltype<T>() == decltype<T>()), bool)
        && __is_valid_expr(decltype<T>() != decltype<T>())
        && __is_convertible_to(decltype<T>() != decltype<T>()), bool)
bool distinct(T a, T b)

```

Obviously, this representation of constraints is internal to the compiler, and an implementation may not expose the primitives required to write such expressions. For example, our GCC implementation does not expose `__is_valid_expr` as an intrinsic.

Once we have reduced predicates up into their simplest form, we can use straightforward classical logic and logical algorithms to solve constraint related problems: primarily ordering and equivalence. These two relations are used to define the semantics for redeclaration, overload resolution, partial template specialization, and the substitution rules for constrained template parameters.

In their most reduced forms, constraints are composed of propositions joined by the logical connectives `&&` (conjunction, and) and `||` (disjunction, or). These have the usual meanings in C++, but cannot be overloaded. Parentheses may also be used for grouping.

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” should actually mean. 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.

In order to determine ordering and equivalence of constraints, the compiler must then decompose these reduced expression into lists of atomic propositions based on the connectives in the constraint expression. The decomposition rules are the same as the left-logical rules of sequent calculus for conjunction and disjunction. A good introduction to the underlying theory and its application can be found in “ML for the Working Programmer” [10].

Note that the description of the decomposition algorithm is intended primarily for compiler writers and tool vendors. Programmers will not need to be aware of these details.

This algorithm constructs a list of lists of atoms called *goals* (or subgoals, depending on the literature). The rules for decomposing constraint expressions modify this data structure by inserting new atoms into the current goal or creating new goals. The rules are specific to the the different kinds of operators.

Given a reduced constraint expression, we first construct a new goal (the current goal) and insert the expression as its only term (the current term). We then consider that expression.

If the current term is a conjunction of the form `a && b`, the expression is removed from the current goal, and its operands `a` and `b` are re- inserted. For example, if the current goal contains the atom `p`, and the expression `a && b`, the result of this rule will modify the current goals so that it contains `p`, `a`, `b`. The new terms `a` and `b` will need to be recursively decomposed.

If the current term is disjunction of the for `a || b`, the expression is removed from the current goal, the current is copied, and the operands are inserted into the different goals. For example, if the current goal contains the atom `p` and the expression `a || b`, the result of this rule is that the current goal will contain `p` and `a` and the copy will contain `p` and `q`. Both `a` and `b` will need to be recursively decomposed.

If the current term is an atom, no action is required, and we advance to the next term. There are no other terms, we advance to the next goal. If there are no other goals, the algorithm terminates.

As an example, suppose we define concepts for `Container` and `Range` like this:

```
template<typename T>
constexpr bool Container()
{
    return value_semantic<T>::value
        && requires (T c) {
            typename T::iterator;
            begin(c) -> typename T::iterator;
            end(c) -> typename T::iterator;
            ... // more requirements
        };
}

template<typename T>
```

```

constexpr bool Range()
{
    return reference_semantic<T>::value
        && requires (T r) {
            typename Iterator_type<T>;
            begin(r) -> Iterator_type<T>;
            end(r) -> Iterator_type<T>;
            ... // more requirements
        };
}

```

The two concepts share some syntactic requirements, namely that both `begin` and `end` are defined and return some an `Iterator` type. However, they differ in the use of hypothetical two type traits, `value_semantic` and `reference_semantic`, which characterize the intended semantics of their constituent types.

Consider the following algorithm:

```

template<typename R>
requires Range<R>
Iterator find(R range, Value_type<R>)

```

Reducing the constraints produces the following requirement:

```

requires reference_semantic<R>::value
    && __is_valid_type(Iterator_type<R>)
    && __is_valid_expr(begin(decltype<R>()))
    && __is_convertible_to(decltype(begin(decltype<R>()))), Iterator_type<R>)
    && __is_valid_expr(end(decltype<R>()))
    && __is_convertible_to(decltype(end(decltype<R>()))), Iterator_type<R>);

```

Again, we are assuming the availability of intrinsics to express these low-level requirements. The result of decomposition produces a single goal containing the following atomic propositions:

```

reference_semantic<R>::value
__is_valid_type(Iterator_type<R>)
__is_valid_expr(begin(decltype<R>()))
__is_convertible_to(decltype(begin(decltype<R>()))), Iterator_type<R>)
__is_valid_expr(end(decltype<R>()))
__is_convertible_to(decltype(end(decltype<R>()))), Iterator_type<R>);

```

Because only conjunctions were found in the constraints, all of the propositions are added to the same list of terms.

Suppose we define `find` more generally, allowing both reference- and value-semantic types to be used:

```

template<typename T>
requires Container<T>() || Range<T>()
Iterator find(const T& x);

```

Reduction is similar to that above, except that the reduced requirements for `Container<T>()` and `Range<T>()` are operands of a logical-or expression. Decomposition results in two different goals (lists of atoms).

```

// Atomic propositions for Container<T>
value_semantic<T>::value
__is_valid_type(typename T::iterator)
__is_valid_expr(begin(decltype<T>()))
__is_convertible_to(decltype(begin(decltype<T>())), typename T::iterator_type)
__is_valid_expr(end(decltype<T>()))
__is_convertible_to(decltype(end(decltype<T>())), typename T::iterator_type)

// Atomic propositions for Range<T>
reference_semantic<T>::value
__is_valid_type(Iterator_type<T>)
__is_valid_expr(begin(decltype<T>()))
__is_convertible_to(decltype(begin(decltype<T>())), Iterator_type<T>)
__is_valid_expr(end(decltype<T>()))
__is_convertible_to(decltype(end(decltype<T>())), Iterator_type<T>);

```

The two goals contain the lists of propositions required by each concept. These lists are used by algorithms to determine the ordering and equivalence of constraints.

### 3.1.4 Relations on Constraints

Constraints can be ordered (i.e., one constraint is more restrictive than another) and compared for equivalence. These relations are used to define semantics for a number of language features.

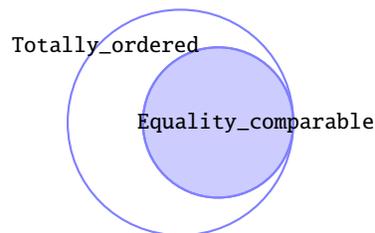
The ordering relation on constraints is called *subsumption*. Given two constraints  $P$  and  $Q$ , then logically  $P$  subsumes  $Q$  iff whenever  $P$  is satisfied (evaluates to **true**),  $Q$  is also satisfied. We are effectively computing the validity the implication  $P \implies Q$ . The idea is closely related to set theory, and the basic concepts are sometimes easier to understand in that context. For example, suppose we define `Totally_ordered` like this:

```

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

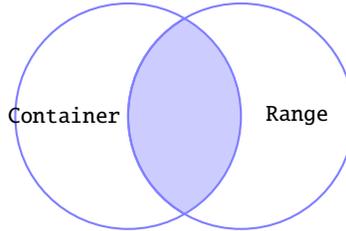
```

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



If we fully decompose each concept into its lists of atomic propositions then we find that `Totally_ordered` is a superset of `Equality_comparable`. This means that `Totally_ordered` subsumes `Equality_comparable` since there is no way to satisfy the requirements of the former without satisfying those of the latter.

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:



Here, neither concept subsumes the other. If we fully decompose both concepts into their sets of propositions, we will find that neither is a superset of the other.

An algorithm for computing the subsumption relation is given below. Again, this discussion is primarily targeted at compiler implementers and tool vendors.

Let  $p$  and  $q$  denote reduced constraints, and let  $P$  be the list of lists (goals) resulting from decomposition of  $p$ , and let  $P_i$  denote the  $i^{th}$  goal in  $P$ . Determining if  $p$  subsumes  $q$  is equivalent to determining if the atoms in each  $P_i$  contains the the atoms contained in the subexpressions of  $q$ . For each  $P_i$ , this is determined by recursively analyzing the subexpressions of  $q$  such that

- If  $q$  is of the form  $a \ \&\& \ b$  then  $P$  subsumes  $q$  iff  $P$  subsumes  $a$  and  $P$  subsumes  $b$ .
- If  $q$  is of the form  $a \ || \ b$  then  $P$  subsumes  $q$  iff  $P$  subsumes  $a$  or  $P$  subsumes  $b$ .
- Otherwise  $P$  subsumes  $q$  iff there is an expression  $p'$  in  $P$  that *matches*  $q$ .

If, for every  $P_i$ , the result of this computation is `true`, then  $p$  subsumes  $q$ .

In general two, expressions always match when they have the same syntax. It must be the case that whenever  $p'$  evaluates to `true`, so does the matched atom  $q$ . If the expressions are the same, that property is trivially satisfied—it must be since state cannot escape constant expressions.

However, there are some special cases. For example, `Same<T, U>` (as  $p'$ ) will match `Convertible<T, U>` (as  $q$ ) since the former implies the latter. Note that the converse is not a valid implication: convertibility does not imply equality. A complete list of special cases or known implications is given in Section ??.

The subsumes relation is used to determine which of two templates is *more constrained*. In particular, a template  $T$  is more constrained 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.

Two constraints  $P$  and  $Q$  are equivalent when  $P$  subsumes  $Q$  and  $Q$  subsumes  $P$ . This is analogous to two sets being supersets of each other, or two propositions being logically equivalent.

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.

## 3.2 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 Same<T, float>()
        || Same<T, double>()
        || Same<T, long double>()
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 3.3. 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 3.3 for more information). The reason is simply that if this is not the case, then the partial specialization will never be selected.

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

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

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

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

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

Member function, constructor, and operator overloads are disambiguated by their template constraints, just like regular function templates.

When a class template is instantiated, the non-template member constraints may be evaluated and the results cached, but the instantiated declarations remain as part of the template specialization, even when the constraints are not satisfied. Member constraints do not change the set of members in the instantiated class.

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

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

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

The nested name specifiers are matched class declarations that are declared with the same constraints associated with the corresponding depth in the template parameters lists. The declarator is also matched to the nested declaration having the equivalent “left over” constraints.

### 3.3 Overloading, and Specialization

In this section, we describe when and how constraints are checked for function templates, member functions, class templates, and partial specializations.

For function templates and member functions constraints are checked during overload resolution. 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).

If the candidate is a non-template member function, then the declaration has already been formed by the instantiation of its enclosing class. If that member function's template declaration is constrained, then those constraints must be instantiated with the same arguments as the class and evaluated.

A constrained 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`.

For class and alias templates, constraints are checked on lookup, prior to the instantiation of the class. This means that constraints are always checked even if the type is not required to be complete. For example:

```

template<Object T>
class vector;

vector<int&> v; // Error
vector<int&>* p; // Error

```

Clearly, `v` will result in an error since the declaration requires a complete type, and `int&` does not satisfy the `Object` requirement. The declaration of `p` is also invalid even though `vector<int&>` is not required to be a complete type. This applies even when (especially when) the template has partial specializations. It is not possible to create a partial specialization that is less constrained than the primary template.

Selecting partial specializations is a similar process to overload resolution. The compiler is required 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 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`.

### 3.4 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*)>
concept 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 be 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.

### 3.5 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`.

### 3.5.1 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 `Argi` 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).

### 3.6 Overload Arguments

The combination of constraints and lambdas allows provides C++ with the ability to use overload sets as function arguments. For example, I could implement the geometric mean like so.

```
template<typename T>
T geometric_mean(initializer_list<T> list)
{
    T p = accumulate(list.begin(), list.end(), operator*);
    return root(list.size(), p);
}
```

Note that the third argument to `accumulate` is the name of an overload set. This is shorthand notation for having written the following:

```
T p = accumulate(list.begin(), list.end(),
                 [](T a, T b) -> T { return operator*(a, b); });
```

The signature and implementation of the lambda are deduced from the requirements of the `accumulate` function.

The motivation for this particular example comes from the use of operator symbols as function (or property) arguments in *Elements of Programming* [5].

### 3.7 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.

### 3.7.1 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.

### 3.7.2 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 implementations 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.

## 4 Constraints and Concepts

Template constraints (concepts-lite) provide a mechanism for 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 ??.

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 features 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 targeted only at checking syntax. The constraint language described in this paper 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].

## 5 Terse Notation

In this section, we present design notes for a terse notation for templates and generic lambdas. This notation is aimed to make constrained lambdas sufficiently concise to be in the spirit of lambdas as a terse notation for function objects and to address concerns about the verbosity of constrained templates and of templates in general. The “terse notation” is pure “syntactic sugar.” It is defined in terms of the simple predicates and requires clauses of constraints (“concepts lite”). The terse notation provides a common notation (and semantics) for lambdas and templates.

*[This note is a major revision of the “Terse Templates” note posted to SG8 before the Bristol meeting and modified after presentations to hundreds and discussions with dozens of people. Thanks to the many who contributed in minor and major ways.]*

### 5.1 Introduction

The purpose of what has been called “terse templates” 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, possibly with a requires clause. We see this “terse notation” 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.

We also briefly discuss the interaction between generic lambdas and any constraint or concept system. Note that lambdas are a form of templates, so semantic differences in how type checking is done or types are specified are bound to lead to confusion and language-technical problems.

### 5.2 The basics

Consider `Container` as an example of a concept. We can define a constrained lambda like this:

```
[](Container& c) ...
```

This means that `c` must be a reference to a type that is a `Container`, that is a type `X` for which the constraint `Container<X>()` is true, a constrained parameter type. Let us first explain how this works for templates, and then come back to lambdas.

For a template, we can write:

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

This is shorthand notation for

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

which again is a shorthand for

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

Here, `__Container` is a compiler-generated name of a type than meets the concept `Container`. The name `__Container` is an implementation artifact (hence the leading underscores) and not available to be used directly by users. Implementers can choose how best to implement the semantics.

We note that when programmers first see something new, they clamor for “heavy” syntax, such as the last version of `sort()`. 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 prefer the traditional “heavy syntax,” you can simply use that.

Note that the meaning (semantics) of the terse notation is defined in terms of the traditional (“heavy”) notation, so it is pure syntactic sugar and does not add new semantic rules to the language.

### 5.3 Type compatibility

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

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

For this to make sense, `p` and `q` must be of the same (random-access iterator) type, and that is the rule. By default, if you use the same constrained parameter type name for two arguments, the types of those arguments must be the same. We chose to make repeated use of a constrained parameter type name imply “same type” because that (in most environments) is the most common case, it would be odd to have an identifier used twice in a scope have two different meanings, and the aim here is to optimize for terse notation of the simplest case.

This also follows from the rewrite rule (above). That `sort()` declaration is equivalent to

```
template<Random_access_iterator __Ran>
void sort(__Ran p, __Ran q);
```

or equivalently

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

To our eyes, this last version looks verbose and the first is still so “heavy” that we predict problems for lambdas. We must look for something better/terser.

Part of the problem is that concept names tend to be long (for good reasons), so that repeating them becomes a bother.

## 5.4 Concepts as Type introducers

Consider what to do when we want two argument types of the same concept that may differ? Consider the C++11 `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. Consider:

```
template<typename For, typename For2, typename Out>
    requires Forward_iterator<For>()
    && Forward_iterator<For2>()
    && Output_iterator<Out>()
    && Assignable<Value_type<For>,Value_type<Out>>()
    && Assignable<Value_type<For2>,Value_type<Out>>()
    && Comparable<Value_type<For>,Value_type<For2>>()
void merge(For p, For q, For2 p2, For2 q2, Out p);
```

We can do better still. 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 introduce the three type name and express their relations:

```
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);
```

This is still quite redundant. We first introduce the three names (`For`, `For2`, and `Out`) and then we state that they must meet some joint constraint (`Mergeable`). We can introduce the three names and at the same time require that they meet the constraint

```
Mergeable{For,For2,Out} // Mergeable is a concept requiring three types
void merge(For p, For q, For2 p2, For2 q2, Out p);
```

Here, `Mergeable` is a concept and the name of a concept followed by `{` is recognized as an *introducer* of constrained template argument names. We use `{}` rather than `<>` after the constraint to avoid confusion and ambiguities with the usual notation for specialization.

For any variant of the “terse notation” to work, concept names must be known to the parser as names of concepts (just like type names must be known to be names of types). Also, any future ideas of checking template bodies and for adding semantic information requires concepts to be known as concepts and not just constant expression. The Palo Alto TR has examples.

## 5.5 Examples

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

Simply replacing `constexpr` with `concept` allows a bit of extra checking. Given that, we can write

```
[](Number n) { return n*n; } // deduce return type
```

and

```
auto square(Number n) { return n*n; } // deduce return type
```

as desired and without introducing new syntax, potentially confusing dual names, or conventions. We assume that there will be many uses of lambdas where a concept/type is used only once. For example predicates, numbers, and strings.

Let's look at a more elaborate example, `find()`:

```
// #1 most verbose
template<class In, class V>
    requires Input_iterator<In>() && Equality_comparable<Value_type<In>,V>()
In find(In p, In q, V v);
```

---

```
// #2 use short-hand for Input_iterator
template<Input_iterator In, class V>
    requires Equality_comparable<Value_type<In>,V>()
In find(In p, In q, V v);
```

---

```
// #3 use shorthand for both arguments:
template<Input_iterator In, Equality_comparable<Value_type<In>> V>
In find(In p, In q, V v);
```

---

```
// define concept:
template<class In, class V>
concept bool Input_comparable()
{
    return Input_iterator<In>() && Equality_comparable<Value_type<In>,V>();
}
```

```
Input_comparable{In, V} // #4 Use Input_comparable
In find(In p, In q, V v);
```

These alternatives are semantically equivalent. Which notation is better depends on the example (e.g., how many template arguments? how many function arguments? What are the semantics of the concepts? How long are the concept names?) and on the aesthetic sense of the programmer. We think they all have their places and there is no significant implementation burden.

Given that, consider a lambda that calls `find()`:

```
// #0 unconstrained:
[](auto p, auto q, auto v) { return find(p,q,v); }
```

The example is not great, but equivalents will be used fairly widely. If passed as an operation to an unconstrained template, we have a problem. For example, this doesn't even say that `p` and `q` are of the same type, so we could call that

lambda with `p` being an `int*` and `q` being a `vector<string>::const_iterator` and get a much delayed error message.

So how do we constrain the types of `p`, `q`, and `v`? We have exactly the alternatives shown above for templates.

```
// #1 use shorthand:  
[](Input_iterator p, Input_iterator q, Equal_comparable<Value_type<Input_iterator> v)  
{ return find(p,q,v); }
```

---

```
// #2 Factor constraints:  
[]<Input_iterator In, Equal_comparable<Value_type<In> V> (In p, In q, V v)  
{ return find(p,q,v); }
```

---

```
// #3 use concept to shorten:  
[] Input_comparable{In, V} (In p, In q, V v) { return find(p,q,v); }
```

They are all pretty verbose, but the last seems plausible. We suspect that terse notation increase in importance as the lambda argument type constraints get simpler, so this `find()` is a relatively complicated example.

## 5.6 Why Are Constrained Lambdas Important?

We use lambdas for many things. Consider first a simple example

```
sort(p,q,[](const string& a, const string& b) { return a > b; });
```

This will, of course work, only if `*p` is “string like”, such as a `std::string` or a `const char*`, but that’s fine because in real use, we’ll know that and if we make a mistake, the compiler will catch it at the point of call of the lambda: `*p` cannot be converted to `const string&`. Now consider making the lambda generic:

```
sort(p,q,[](auto a, auto b) { return a > b; });
```

The notation is terser, which is part of the attraction of “generic” lambdas, possibly their main attraction, and the reason for people pushing for even terser syntax. Also, the exact source text will work for other inputs, say, for the case where `p` and `q` are iterators into a `vector<double>`. However, we also lost something. Errors are caught later. For example (assuming `sort()` is unconstrained):

```
void f(vector<double>& v)  
{  
  sort(v, [](double x, double y) { return x%100 < y%100; }); // error  
  sort(v, [](auto x, auto y) { return x%100 < y%100; }); // error  
};
```

The first error could (and should) be caught when the lambda is defined. The second cannot (easily) be caught until `sort()` calls its predicate, instantiates the generic lambda, and find that we have applied `%` to a `double`. Then, and only then, do we get one of the error messages that is a reason for the widespread desire for concepts and constraints.

When passing a lambda as a template argument, there are four combinations:

- Unconstrained lambda + unconstrained template argument
  - late checking, you're on your own
- Unconstrained lambda + constrained template argument
  - Use constraint from template
- Constrained lambda + unconstrained template argument
  - Use constraint from lambda
- Constrained lambda + constrained template argument
  - Use (constraint from lambda and constraint from template)

Checking is straightforward whenever there are any constraints involved. We hope that the first case (no constraints anywhere) will become rare. We suspect that it will mostly be used by people unacquainted with constraints and for expression templates where only minimal constraints can be specified. From a language-technical point of view constrained lambdas are essential because lambdas are a notation for templates, so constraining one and not the other is incoherent.

Lambdas with few arguments are likely to be very common (because function objects with few arguments are). For example:

```

[](Range& c) { ... }
auto trim = []( const String& s ) { return trim_at_both_ends(s); };
[](Unary_predicate p) { ... }
auto func = []( Forward_iterator first, Forward_iterator last ) { ... }
callback1.register = func;
callback2.register = func;

```

Here `Range`, `String`, `Unary_predicate`, and `Forward_iterator` must be concepts.

## 5.7 Concepts and auto

For lambdas, we gain precise specification (implying early checking and improved error messages) by replacing `auto` with a concept. This is logically equivalent to replacing typename with a concept in a template declaration. The equivalent can be done when `auto` is used as a return type or as a type in an object definition. Consider:

```

auto r = find(p,q,v);

```

What is the type of `r`? Well, it is the type of `p`, but we have to know `find()` to know that. Using `auto`, there is no way to express our expectation about the type of a result of a computation. Using concepts, we can express such an expectation:

```

Random_access_iterator r = find(p,q,v);

```

Now if we wanted a random-access iterator and passed we get an early error. Similarly, we could write:

```
Value_type<Range> x = [](Range& c) { ... } // we expect a value
Range x = [](Range& c) { ... } // we expect a Range
String trim = [](const String& s) { return trim_at_both_ends(s); };
```

We can write `find()` like this:

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

Here, the concept `Forward_iterator` is used as a return type. As in the case of a concept used as an argument type, it is recognized by being a concept and is used as the name of an argument type. A concept used as a return type must be defined in terms of argument types that are deduced. The declaration of `find()` above is equivalent to:

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

Alternatively, we can use the suffix return type notation:

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

Or we can rely on C++14 return type deduction:

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

We can, of course, shorten further using a suitable concept for the two parameter types:

```
Input_comparable{For,Val}
For find(For p, For q, Val v);
```

For a lambda, we get:

```
[] Input_comparable{For,Val}
(For p, For q, Val v) { ... }
```

As ever, the notation is consistent between lambdas and (other) templates.

## 5.8 Clusters of templates

We considered whether an even terser syntax (more minimal notation) was possible. It is, but we deemed it an unnecessary complication.

Consider again `Mergeable`. There are five algorithms in the STL that requires `Mergeable`, so we could use some form of recouping mechanism to make a single `Mergeable{For,For2,Out}` apply to all five.

There are two variants of this idea. A form of using-directive:

```
using Mergeable{For, For2, Out}; // from here on until the end scope
```

Given that, `For`, `For2`, and `Out` in template declarations and lambdas must meet the `Mergeable` constraint.

Alternatively, we could explicitly specify the scope of `Mergeable{For, For2, Out}` and get a form of block:

```
using Mergeable{For,For2,Out}
{
    // Here, For, For2, and Out in template declarations and lambdas
    // must meet the Mergeable
}
```

Either can be made to work. That is, there are no significant implementation problems. The question is “is it worth the effort.” Will it be used often enough to warrant the effort teaching and learning yet another construct? Together with some students, we conducted a small study clustering all STL algorithms. We need to write up that experiment, but the conclusion was clear: There are not enough clean “clusters” of algorithms with identical constraints for this kind of clustering syntax to be valuable. So, we have dropped this line of inquiry and would need new evidence of utility to revisit this question. Obviously, leaving this out gives us a simpler language.

## 5.9 But it doesn’t look like a template!

When first seeing something like

```
Number sqrt(Number n);
```

or

```
Input_comparable{In, V}
In find(In p, In q, V v);
```

Many C++ programmers exclaim “But it doesn’t look like a template!” It doesn’t, but other languages with generic functions do not require a long keyword to precede a generic function declaration. Many C++ programmers complain about the verbosity of the current template syntax, and in fact the earliest designs for C++ templates did not have the template keyword. When a compiler knows whether an identifier is a concept name or not the terse notation is simple to parse. I consider most uses of template analogous to the use of `struct` in C. Do not confuse the familiar with the simple. The proposed syntax is readable and parsable. We considered “louder”, more verbose notations, but did not find them consistently better than what is described here.

## 6 Standard Wording

The proposed wording for concepts lite is written against the current Working Paper, targeting C++14.

### 5.1.1 General [expr.prim.general]

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

### 5.1.3 Requires expressions [expr.prim.requires]

A requires expression provides a concise way to express 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 requirement-parameter-list requirement-body  
requirement-parameter-list::  
    ( parameter-declaration-clauseopt )  
requirement-body::  
    { requirement-list }  
requirement-list::  
    requirementopt  
    requirement-list ; requirementopt  
requirement::  
    simple-requirement  
    compound-requirement  
    type-requirement  
    nested-requirement  
simple-requirement::  
    expression  
compound-requirement::  
    { expression } trailing-requirements
```

```

trailing-requirements::
    constraint-specifier-seq result-type-requirementopt
constraint-specifier-seq::
    constexproptnoexceptopt
result-type-requirement::
    -> type-id
nested-requirement::
    requires-clause

```

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

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

The *requires-expression* may be introduce local arguments 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 list of requirements. If each requirement in that list is satisfied, the result of the expression is **true**, or **false** otherwise.

#### 5.1.3.1 Simple requirements [expr.req.simple]

A *simple-requirement* introduces a requirement that the *expression* is a valid expression when instantiated. If the instantiation of the constraint results in a substitution failure, the the requirement is not satisfied.

#### 5.1.3.2 Compound requirements [expr.req.compound]

A *compound-requirement* introduces a set of requirements involving a single *expression*. The *expression* must compile when instantiated.

If a *result-type-requirement* is present then the result type of the instantiated *expression* must satisfy that requirement. If the required *type-id* is a constrained placeholder type (7.1.6.5), then those constraints must also be satisfied.

If the **constexpr** specifier is present, the instantiated expression must be **constexpr**-evaluable. If the **noexcept** specifier is present, instantiated expression must not propagate exceptions.

#### 5.1.3.3 Type requirements [expr.req.type]

A *type-requirement* introduces a requirement that an associated type name can be formed when instantiated. If the instantiation of the type requirement results in a substitution failure, the requirement is not satisfied.

### 5.1.3.4 Nested requirements [expr.req.type]

A nested requirement introduces additional constraints to be evaluated as part of the *requires* expression. The *requires-clause* is *constexpr* evaluated, and the requirement is satisfied only when that evaluation yields **true**.

## 7.1 Specifiers [dcl.spec]

```
decl-specifier:  
...  
concept
```

### 7.1.6 Simple type specifiers [dlt.type.simple]

```
type-name:  
...  
concept-name  
partial-concept-id
```

```
concept-name:  
identifier
```

```
partial-concept-id:  
concept-name < template-argument-list >
```

#### 7.1.6.5 Constrained type specifiers [dcl.spec.constrained]

The type denoted by a *concept-name* or *partial-concept-id* is a constrained placeholder type. A constraint is formed from the application of the concept to the placeholder type. The placeholder type is used as the first template argument of the constraint. If the type specifier is a *partial-concept-id*, the specified arguments follow the placeholder type in the formed constraint. [Example:

```
Real y = f(x); // decltype(y) is a placeholder type,  
              // constrained by Real<decltype(y)>  
  
Same_as<T> a = f(b); // decltype(a) is a placeholder type,  
                   // constrained by Same_as<T, decltype(a)>
```

— end example]

The first use of *concept-name* or *partial-concept-id* within a scope binds that name to the placeholder type so that subsequent uses of the same name refer to the same type. [Example:

```
template<typename T>  
concept bool Number() { ... }
```

```
auto mul = [](Number a, Number b) { return a * b; }
```

The types of `a` and `b` are the same. This is equivalent to having written:

```
auto mul = []<Number N>(N a, N b) { return a * b; }
```

— *end example*]

### 7.1.7 The concept specifier

[**dcl.concept**]

The **concept** specifier shall be applied to only the definition of a function template. A function template definition having the **concept** specifier is called a *concept definition*.

Every concept definition is also a **constexpr** declaration (7.1.5).

Concept definitions have the following restrictions:

- The template must be unconstrained.
- The result type must be *bool*.
- The declaration may have no function parameters.
- The declaration must be defined.
- The function shall not be recursive.

[*Example:*

```
template<typename T>
  concept bool C() { return true; } // OK: Concept definition

template<typename T>
  concept int c() { return 0; } // error: must return bool

template<typename T>
  concept bool C(T) { return true; } // error: must have no parameters

concept bool p = 0; // error: not a function template
```

— *end example*]

If a program declares a non-concept overload of a concept definition with the same template parameters and no function parameters, the program is ill-formed. [*Example:*

```
template<typename T>
  concept bool Totally_ordered() { ... }

template<Graph G>
  constexpr bool Totally_ordered() // error: subverts concept definition
  { return true; }
```

— *end example*]

## 14 Templates

[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-clauseopt declaration  
concept-introduction declaration
```

*requires-clause:*

```
constant-expression
```

Add new paragraphs:

A template declaration with a *requires-clause* is a *constrained template*. A *requires-clause* introduces template constraints in the form of a *constant-expression* whose result type is **bool**.

A *declaration* introduced by a *concept-introduction* (14.9.5) is a *constrained template*.

### 14.1 Template parameters

[temp.param]

The syntax for *template-parameters* is:

*template-parameter:*

```
type-parameter  
parameter-declaration  
constrained-parameter
```

*constrained-parameter:*

```
constraint-id ...opt identifier  
constraint-id ...opt identifier = constrained-default-argument
```

*constraint-id:*

```
concept-name partial-concept-id
```

*constrained-default-argument:*

```
type-id  
template-name  
expression
```

Add new paragraphs:

A *constrained-parameter* is introduced by a *constraint-id*, which is either a *concept-name* or a *partial-concept-id*. The concept definition referred to by the *constraint-id* determines the kind of template parameter and the constraints applied to that argument.

The template parameter introduced by the *constraint-id* has the same kind and constraints as the first template parameter of the concept definition. If that template parameter is a parameter pack, then the constrained parameter shall also be declared as a parameter pack. [Example:

```
template<typename... Ts>
  concept bool Same_types() { ... }

template<Same_types Args> // error: Must be Same_types...
  void f(Args... args);
```

— end example]

The *constraint-id* is used to form a constraint on the declared template parameter by applying the *concept-name* to that parameter. If the *constraint-id* is a *partial-concept-id*, then the supplied *template-arguments* follow the declared parameter in the application. [Example:

```
template<Input_iterator I, Equality_comparable<Value_type<I>> T>
  I find(I first, I last, const T& value);
```

The constraints formed from these constrained template parameters are equivalent to the following declaration:

```
template<typename I, typename T>
  requires Input_iterator<I>() && Equality_comparable<T, Value_type<I>>()
  I find(I first, I last, const T& value);
```

— end example]

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

A *template-declaration* having *constrained-parameters* in its *template-parameter-list* is a *constrained template*.

## 14.2 Template names

[tmp.names]

Modify paragraph 6.

A *simple-template-id* that names a class template specialization is a *class-name* provided that the *template-arguments* satisfy the constraints (if any) of the referenced primary template. Otherwise the program is ill-formed. [Example:

```
template<Object T> // T must be an object type
  class optional;

optional<int&>* p; // error: int& is not an object type
```

— end example] [Note: This ensures that a partial specialization cannot be less specialized than a primary template.]

### 14.3.3 Template template arguments [tmp.arg.template]

Modify paragraph 3.

A *template-argument* matches a template *template-parameter* (call it P) when each of the template parameters in the *template-parameter-list* of the *template-argument*'s corresponding class template or alias template (call it A) matches the corresponding template parameter in the *template-parameter-list* of P, and when P is not less constrained (14.9.3) than A. [Example

*// ... from standard*

```
template<template<Copyable>> class C>
    class stack { ... };
```

```
template<Regular T> class list1;
template<Object T> class list1;
```

```
stack<list1> s1; // OK: Regular is more constrained than Copyable
stack<list2> s2; // error: Object is not more constrained than Copyable
```

— *end example*]

### 14.5.1.5 Constrained members of class templates [temp.mem.func]

A member function of a class template can be constrained by writing a *requires* clause after the member declarator. [Example:

```
template<typename T>
    class S {
        void f() requires Integral<T><();
    };
```

— *end example*]

The member function's constraints are not evaluated during class template instantiation. [Note: Member function constraints are checked during overload resolution].

### 14.5.5.1 Matching of class template partial specializations [temp.class.spec.match]

Modify paragraph 2.

A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (14.8.2), and the deduced template arguments satisfy the constraints of partial specialization (if any) (14.9.2).

### 14.5.5.1 Partial ordering of class template specializations [\[temp.class.spec.match\]](#)

Modify paragraph 1.

For two class template partial specializations, the first is at least as specialized as the second if, given the following rewrite to two function templates, the first function template is at least as specialized as the second according to the ordering rules for function templates (14.5.6.2):

- the first function template has the same template parameters **and constraints** as the first partial specialization and has a single function parameter whose type is a class template specialization with the template arguments of the first partial specialization, and
- the second function template has the same template parameters **and constraints** as the second partial specialization and has a single function parameter whose type is a class template specialization with the template arguments of the second partial specialization.

Add paragraph 3.

[*Example:*

```
template<typename T> class S { };
template<Integer T> class S<T> { }; // #1
template<Unsigned_integer T> class S<T> { }; // #2

template<Integer T> void f(S<T>); // A
template<Unsigned_integer T> void f(S<T>); // B
```

The partial specialization #2 will be more specialized than #1 for template arguments that satisfy both constraints because **A** will be more specialized than **B**. — *end example*]

### 14.5.6.1 Function template overloading [\[tmp.over.link\]](#)

Modify paragraph 6.

Two function templates are *equivalent* if they are declared in the same scope, have the same name, have identical template parameter lists, **and** have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters, **and have equivalent constraints** (14.9.2).

### 14.5.6.2 Partial ordering of function templates [\[tmp.func\]](#)

Modify paragraph 2.

Partial ordering selects which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function type. The deduction process determines whether one of the templates is more specialized than the other. If so, the more specialized template is the one chosen by the partial ordering process. **If both deductions succeed, the the more specialized template is the one that is more constrained (14.9.3).**

## 14.9 Constraints

[temp.con]

A *template-declaration* declared with *constrained-parameters*, a *requires* clause, or introduced by a *concept-introduction* is a *constrained template*. Otherwise, the declarations is an *unconstrained template*.

The *actual constraints* of a constrained template are taken to be the conjunction of constraints formed from constrained template parameters, a *requires* clause, and those derived from an *concept-introduction*, inclusively. The actual constraints can be represented as a *logical-and-expression* that is also a *constant-expression*.

For constrained templates, these constraints determine if the template declaration can be instantiated (14.7). For constrained member functions of class templates, the constraints determine if the instantiated member function is a viable candidate during overload resolution (14.5.1.1).

The actual constraints of a constrained template are represented as an logical expression comprised of *atomic propositions*, *concept checks*, and the logical connectives, `&&` and `||`.

An *atomic proposition* is any C++ expression except a *concept-check*, *logic-or-expression*, or *logical-and-expression*. [Example: The following are examples of atomic propositions:

```
true
is_integral<T>::value
!(a == b)
max(a, b) == a
(N == 0)
(M < 3)
```

Assuming that `a`, `b`, `max(a, b)`, `M`, and `codeN`, and are constant expressions. —  
*end example*]

A *concept check* is a call to concept definition.

### 14.9.1 Constraint reduction

[temp.con.reduce]

A constrained template's actual constraints are *reduced* by inlining concept checks, and producing an expression comprised of only atomic propositions and

*logical-and-expressions* and *logical-or-expressions*.

### 14.9.2 Constraint Satisfaction [temp.con.sat]

A template's constraints are satisfied if the `constexpr` evaluation of the reduced constraints results in *true*.

### 14.9.3 Partial ordering of constraints [temp.con.ord]

Partial ordering of constraints is used to choose among ambiguous specializations during the partial ordering of function templates, the partial ordering of class templates, and the use of template template arguments. The partial ordering of two reduced constraints  $P$  and  $Q$  determines if  $P$  *subsumes*  $Q$ .

Determining the partial ordering of reduced constraints requires their *decomposition* into a list of lists of atomic propositions. Decomposition of a reduced constraint  $P$  begins with a single list (the *current list*) containing the reduced constraint  $P$  (the *current term*). Decomposition proceeds by analyzing the current term.

- If  $P$  is of the form  $a \ \&\& \ b$ , then replace  $P$  in the current list with the operands  $a$  and  $b$  so that  $a$  is the current term and  $b$  will be the next term.
- If  $P$  is of the form  $a \ \|\| \ b$ , then create a copy of the current list, replacing  $P$  with  $a$  in the original and replacing  $P$  with  $b$  in the copy so that  $a$  is the current term in the original and  $b$  is the current term in the copy.
- Otherwise, advance to the next term. If there are no remaining terms, advance to the next list. If there are no remaining lists, decomposition terminates.

Given two reduced constraints  $P$  and  $Q$ , then  $P$  subsumes  $Q$  only if all of  $Q$ 's atomic propositions can be found in the  $P$ . Denote this comparison as  $P \vdash Q$ .

This is computed by first decomposing  $P$  into a list of lists of atomic propositions,  $L$ , and then comparing the expression  $Q$  against each list  $L_i$  in  $L$ , which is equivalent to determining if  $L_i \vdash Q$  according to the following rules:

- If  $Q$  is of the form  $a \ \&\& \ b$ , then  $L_i \vdash Q$  if and only if  $L_i \vdash a$  and  $L_i \vdash b$ .
- If  $Q$  is of the form  $a \ \|\| \ b$ , then  $L_i \vdash Q$  if and only if  $L_i \vdash a$  or  $L_i \vdash b$ .
- Otherwise  $L_i \vdash Q$  if and only if there exist some atomic proposition  $A$  in  $L_i$  that *matches*  $Q$  (14.9.4).

Only when each  $L_i \vdash Q$  does  $P \vdash Q$ .

For two template declarations with equivalent type, the first is at least as constrained as the second if both templates are unconstrained, the first is constrained and the second unconstrained, or the constraints of the first subsume the constraints of the second.

### 14.9.3 Constraint equivalence [temp.con.eq]

Two constraints  $P$  and  $Q$  are equivalent if and only if  $P \vdash Q$  and  $Q \vdash P$ .

Two template declarations are equivalently constrained if they are both unconstrained or have equivalent constraints.

### 14.9.4 Matching propositions [temp.con.match]

Two propositions  $P$  and  $Q$  match ( $P \vdash Q$ ) if they have the same spelling. The names of arguments introduced in a *requirement-parameter-list* are not considered.

### 14.9.5 Concept Introductions [temp.con.intro]

A *concept-introduction* introduces a list of template parameters for trailing declaration (14) or lambda expression. (5.1.2).

```
concept-introduction:
    concept-name { introduced-parameter-list }
introduced-parameter-list:
    identifier introduced-parameter-list , identifier
```

The concept name is matched, based on the number of introduced parameters to a corresponding concept definition. If no such concept can be found, the program is ill-formed.

The kind of each introduced parameter (type, non-type, template), is the same as the corresponding template parameter in the matched concept definition. The concept is applied introduced parameters as a constraint on the trailing declaration. [*Example*:

```
template<typename I1, typename I2, typename O>
    concept bool Mergeable() { ... }

Mergeable{A, B, C}
    O merge(A first1, A last1, B first2, B last2 C out);
```

$A$ ,  $B$ , and  $C$  are introduced as type parameters. The constraint on the algorithm is `Mergeable<A, B, C>()`. The declaration is equivalent to:

```
template<typename A, typename B, typename C>  
    requires Mergeable<A, B, C>()  
O merge(A first1, A last1, B first1, B first2, C out);
```

— *end example*]

## Acknowledgements

We are grateful for the input, comments, and corrections from Jason Merrill, Greg Marr, Chris Jefferson, Daveed Vandevoorde, Matt Austern, Herb Sutter, Tony Van Eerd, and Michael Lopez. The feedback received from participants in the ACCU and C++Now conferences has also been valuable.

This work is funded, in part, by the National Science Foundation through grant ACI-1148461.

## 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.