Concepts Lite: Constraining Templates with Predicates

Andrew Sutton, Bjarne Stroustrup

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

1 Introduction

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

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

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

This paper is organized like this:

- Tutorial: introduces the basic notions of constraints, shows examples of their use, and gives examples of how to define constraints.
- Discussion: explains what constrains are not. In particular, we try to outline constraints's relation to concepts and to dispel some common misconceptions about concepts.

- User's guide: provides many more tutorial examples and demonstrate the completeness of the constraints mechanism.
- Implementation: gives an overview of our GCC compiler support for constraints.
- Extensions: we discuss how constraints might be extended to interact with other proposed features.
- Language definition: presents a semi-formal definition of constraints

2 Constraints Tutorial

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

2.1 Introducing Constraints

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
template<Object T, Allocator A>
```

```
class vector
{
  requires Equality_comparable<T>()
    bool operator==(const vector& x) const;

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

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

2.1.1 Multi-type Constraints

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

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

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

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

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

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

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

2.1.2 Overloading

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Calling `advance` works as expected. For example:

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

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

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

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

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

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

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

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

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

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

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

2.2 Defining Constraints

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

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

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

The `Equality_comparable` constraint might be defined like this:

```
template<typename T>
constexpr bool Equality_comparable()
{
  return has_eq<T>::value && Convertible<eq_result<T>, bool>()
      && has_neq<T>::value && Convertible<neq_result<T>, bool>();
}
```

The body of the constraint is a conjunction of type traits and calls to other constraints. Constraints build directly on type traits, they do not replace them. For example, `Convertible` has this definition:

```
template<typename T typename U>
constexpr bool Convertible()
{
  return is_convertible<T, U>::value;
}
```

The template constraint features do not aim to define a trait definition language. Our primary interest with this proposal is improving the declaration and use of generic algorithms and data structures. Because traits can be reasonably implemented in C++11, we have not proposed any new language features that will simplify their writing. However, some support can be provided in the form of compiler intrinsics. These compiler features help reduce the burden of implementing constraints, speed up compilation (compared to C++11 code using `enable_if`), and help avoid some of the substitution failure problems typically associated with low-level type trait implementation. We discuss these issues at length in Section 5.1.

There is no `concept` keyword. For this simpleconstraints system `costexpr` suffices. However, for proper integrations with generic lambdas, for better messages from `static_assert` and for the eventual inclusion of semantic properties

in a full-blown concepts design, some `constexpr` functions need to be labelled `concept`; see Section 6.2;

3 Constraints and Concepts

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

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

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

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

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

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

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

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

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

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

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

4 User's Guide

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

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

```
Equality_comparable<T>()
!is_void<T>::value
is_lvalue_reference<T>::value && is_const<T>::value
is_integral<T>::value || is_floating_point<T>::value
N == 2
X < 8
```

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

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

4.1 Anatomy of a Constraint

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

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

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

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

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

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

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

4.1.1 Constraint Predicates

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

- A function template
- Has no function parameters (is nullary)
- Returns `bool`
- Is `constexpr`

Recall that the definitions of `Equality_comparable` and `Convertible` from Section 2.

```
template<typename T> constexpr bool
Equality_comparable()
{
  return has_eq<T>::value && Convertible<eq_result<T>, bool>()
      && has_neq<T>::value && Convertible<neq_result<T>, bool>();
}

template<typename T, typename U>
constexpr bool Convertible()
{
  return is_convertible<T, U>::value;
}
```

Both are constraint predicates. Inside a `requires` clause, a call to a constraint predicate is called a *constraint check*. Check expressions are recursively expanded, inlining the definition of the predicate into the expression. For example, suppose we have this:

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

The shorthand `Equality_comparable` requirement is first transformed into a constraint expression: `Equality_comparable<T>()`. Because `Equality_comparable` is a constraint predicate, it is recursively expanded (as is the nested check of

`Convertible`). Ultimately, the previous declaration is equivalent to having written:

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

We prefer the more concise expression of requirements. Constraint predicates are the basic building block of conceptual abstractions. Concepts like `Input_iterator`, `Range`, and `Relation` are defined through the composition of constraint predicates.

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

4.1.2 Connectives

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

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

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

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

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

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

```
template<typename T>
constexpr bool Container()
{
  return value_semantic<T>::value
      && Equality_comparable<T>()
      && has_begin<T>::value
      && has_end<T>::value
```

```
      && has_size<T>::value; // Probably more..
}

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

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

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

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

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

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

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

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

```
has_end<T>::value
```

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

4.1.3 Relations on Constraints

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

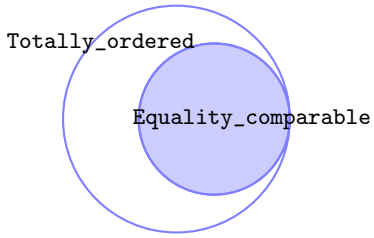
Two constraints are equivalent when the contain the same propositions. Because equivalence is based on the decomposed sets of propositions, two `requires` clauses may have different spellings, but may require the same things.

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

For example, we define `Totally_ordered` like this:

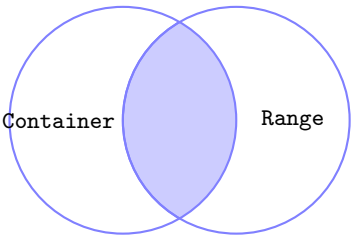
```
template<typename T>
constexpr bool Totally_ordered()
{
  return Weakly_ordered<T>() && Equality_comparable<T>();
}
```

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



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

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



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

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

4.2 Declarations, Redeclarations, and Overloading

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

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

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

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

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

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

This holds for functions as well:

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

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

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

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

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

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

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

When `distance` is called, the compiler determines the best overload. The process of overload resolution is described in 4.3 and an more specifically in **??**. 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 there is no other relation between the constraints of partial specializations. It is not strictly required, for example, that the specializations match or refine the constraints of the primary template. However, we have found that it is generally a good idea to do so.

4.3 Overloading and Specialization

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

1. Construct a set of candidate functions, instantiating templates if needed
2. Determine which of those candidates is viable
3. Select the best of the viable candidates.

Constructing the candidate set entails the instantiation of function templates. 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 that substitution fails, or if the constraint evaluates to `false`, then that declaration is not a viable candidate.

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

Consider the following:

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

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

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

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

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

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

- Identify viable specializations
- Select the best viable specialization

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. If template argument deduction fails, the constraints cannot be instantiated, or if they evaluate to `false`, the specialization is not viable.

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 : integral_constant<bool, (T(-1) < T(0))> { };

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

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

4.4 Non-Type Constraints

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

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

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

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

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

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

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

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

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

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

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

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

The hash table's can declared like this:

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

or equivalently:

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

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

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

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

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

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

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

4.5 Template Template Parameters

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

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

Any argument substituted for the `Cont` must have a conforming template "signature" (same number and kinds of parameters) and also be at least as constrained than 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<Checking>()
class smart_pointer;
```

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

4.5.1 Variadic Constraints

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

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

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

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

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

`Convertible<`$\texttt{Arg}_1$`, size_t>() && Convertible<`$\texttt{Arg}_2$`, size_t>() && ...`

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

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

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

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

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

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

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

Or equivalently:

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

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

4.6 Constrained Members

We conclude this guide with some notes about class templates and their members. Member functions, constructors, and operators can be constrained and overloaded just like regular function templates. For example, the constructors of the `vector` class are declared like this:

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

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

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

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

Definitions could be written inline or outside the declaration in the usual ways. For example:

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

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

The template requirements must be repeated for each declaration because the constraints are needed to match the definitions against their original declarations.

5 Implementation

We have implemented the proposed features as a branch of GCC 4.8. A few features are currently still incomplete or being refined: We are considering how constraints interact with variadic templates, working on improving diagnostics, and constraining the facilities provided in the Standard Library. The constraints used in the Standard Library are essentially the same as those presented in "A Concept Design for the STL" [1].

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

5.1 Compiler Support for Type Traits

One of the goals of this implementation is to decrease compile times by providing facilities to help eliminate complexity in the definition of type traits and constraints. The support provided by the compiler will also help us understand the kinds of constraints that need to be written and guide the design of a comprehensive constraint language for concepts.

Compiler support comes in the form of several intrinsics: `__is_same`, `__is_valid_expr` and `__is_valid_type`. These extend the set of built-in trait expressions already provided by GCC.

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

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

The declaration avoids the comparison of specializations by expressing the conditions as a disjunction, and the use of the intrinsic avoids numerous instantiations of the `std::is_same` type trait.

The `__is_valid_expr` and `__is_valid_type` intrinsic provides facilities for writing queries on valid expressions and associated type name. With this facility, we can reduce library complexity and hopefully decrease compile times by reducing the number of template instantiations required to implement such queries in C++11. For example, a possible definition of the `has_eq` and `eq_result` traits referenced earlier might be:

```
template<typename T>
struct eq_result_impl
{
  template<typename X>
```

```
  static auto check(const X& x) -> decltype(x == x);
  static subst_failure check(...);

  using type = decltype(check(declval<T>()));
};

template<typename T>
using eq_result = typename eq_result_impl<T>::type;

template<typename T>
struct has_eq
  : integral_constant<bool, !is_same<eq_result<T>, subst_failure>::value>
{ };
```

While such implementations are easily implemented, they incur significant overhead. Following this pattern, we could requires three templates for each syntactic query, causing the size of a generic library to grow considerably. Checking these constraints requires the instantiation of all these templates also. This can add significant overhead to your compiler times.

The `__is_valid_expr` and `__is_valid_type` features provide mechanisms for eliminating this overhead. The `__is_valid_expr` intrinsic takes:

- A type name, and evaluates to `true` if that name is valid
- A use pattern (expression), evaluating to `true` if the expression can be type checked.
- An interface requirement, specifying a use pattern and constraints on the expression's result type, and evaluating to `true` if the expression can be type checked and the result type satisfies the associated constraint.

The syntax for writing interface rquirements is the same as that used to write requirements in "A Concept Design for the STL" [1]. It follows the intialization syntax. For example:

```
T (e)     // T can be constructed with decltype(e)
T = {e}   // decltype(e) is convertible to T
T == {e}  // decltype(e) is the same as T
```

This feature lets us write SFINAE-friendly traits in a reasonably concise way:

```
template<typename T>
constexpr bool Equality_comparable()
{
  return __is_valid_expr(bool = {declval<T>() == declval<T>()})
      && __is_valid_expr(bool = {declval<T>() != declval<T>()})
}
```

If the use pattern cannot be type checked or the result type does not satisfy the conversion or same-type requirements, the entire intrinsic evaluates to `false`.

The use of `declval` causes the traits to be more verbose than we would prefer, but benefit of this specification is that it causes the instantation of only one template (`declval`).

The `__is_valid_type` feature allows programmers to test whether an associated type name is valid:

```
template<typename I>
constexpr bool Iterator()
{
  return __is_valid_type(Iterator_category<I>);
}
```

The `Iterator` constraint evaluates to `true` whenever the alias `Iterator_category` names a valid type name.

5.2 Declval

The diagnostiscs reported by the compiler are pretty-printed expressions required in a constraint. This means that the readability of a particular requirement will affect the readability of error messages. Unfortunately, the only the way to write syntactic requirements is the generous use of the `declval` function to create expressions of a particular type. These requirements become unacceptably verbose. For example, the subscript operator for random access iterators can be written like this:

```
__is_valid_expr(decltype(*i) == declval<I>()[declval<Difference_type<I>>()]);
```

Between all of the enclosing parentheses, brackets, and angles, it is very difficult to see what is actually required.

In experimenting with ways to increase readability, we created a new declaration specifier, `__declval`, that could be used to introduce local, unevaluated variables into a constraint function. These declarations can be used in place of the `declval` function. This the readability of constraints significantly.

```
template<typename I>
constexpr bool Random_access_iterator()
{
  __declval I i;
  __declval Difference_type<I> n;
  return Bidirectional_iterator<I>()
      && __is_valid_expr(I& == {i += n})
      && __is_valid_expr(I == {i + n})
      && __is_valid_expr(I == {n + i})
      && __is_valid_expr(I& == {i -= n})
      && __is_valid_expr(I == {i - n})
      && __is_valid_expr(Difference_type<I> == {i - i})
      && __is_valid_expr(decltype(*i) == {i[n]});
}
```

However, this might be done more elegantly in a different way. In particular, the `requires` notation used in [1] allows the introduction of local variables, which can then be used in constraints.

5.3 The Library

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

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

5.3.1 Type Traits

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

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

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

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

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

- `Equality_comparable`
- `Totally_ordered`
- `Movable`
- `Copyable`

- `Semiregular`
- `Regular`
- `Function`
- `Regular_function`
- `Predicate`
- `Relation`

Their definitions follow from those given in [1].

5.3.2 Iterator

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

- `Iterator_category`
- `Value_type`
- `Difference_type`

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

For reference, the implementation of `Value_type` is:

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

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

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

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

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

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

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

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

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

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

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

The standard iterator hierarchy is unchanged.

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

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

5.3.3 Algorithm

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

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

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

6 Extensions

The following sections describe features that we haveconsidered, but are not yet ready for implementation.

6.1 Concepts

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

6.2 Constrained Lambdas

Constraining templates, but not lambdas would create a major irregularity in the language. Fortunately, constraining lambdas is not too hard. The main problem is to find a suitably convenient syntax. Consider:

```
template<Ordered T>
struct Greater {
  const T val;
  Greater(const T& v) :val{v} {}
  bool operator()(const T& x) const { return x>val;}
};
```

For `find_if()`, we can use `Greater` or a lambda:

```
template<typename T>  // deliberately not constrained
void test(vector<string>& v)
{
  auto p = find_if(v.begin(),v.end(),Greater{"Bristol"});
  auto q = find_if(v.begin(),v.end(),
                   [](const auto& x) { return x>"Chicago"; });
};
```

This is of course a trivial example, but if we passed a `vector<complex<double>>` the error would be diagnosed at the point of call of `greater` but only at instantiation time for the lambda. With the increasing popularity of lambdas, this would be a very bad "oversight."

Also, we cannot do overload resolution based on properties of a generic lambda.

How do we say that a lambda is constrained. The obvious first solution is to use a `requires`. However, the lambda has no named template argument type use mention so we would have to write something like

```
template<typename T>  // deliberately not constrained
void test(vector<string>& v)
{
  auto q = find_if(v.begin(),v.end(),
                   [](const auto& x) requires Ordered<decltype(x)>()
                   { return x>"Chicago"; });
};
```

Unimpressive!

We suspect that would work, but recommend exploring the (old) notion of `auto` being the least constraining type (just like `typename` in a template argument list), and then allowing the name of a concept as a more constraining alternative:

```
template<typename T>   // deliberately not constrained
void test(vector<string>& v)
{
  auto q = find_if(v.begin(),v.end(),
                   [](const Orderded& x) { return x>"Chicago"); }
};
```

That is, a constraint used in a lambda argument type would mean, "the type of the argument must be one that satisfy the constraint."

Similarly, we might write:

```
void sort(Sequence& c)   // takes sequences
{
  Random_access_iterator q = c.begin(); // we need a random access iterator
  // ...
};
```

For this construct to parse and be unambiguous, the compiler must be able to know that `Ordered`, `Sequence`, and `Radom_access_iterator` are constraints. Just knowing that they are `constexpr` functions is not sufficient.

We suggest that the keyword `concept` is used to designate a `constexpr` functions that we deem to be a concept so that it can be used as the base type. This would also give us a syntactic handle on which to eventually attach semantic requirements.

7 Standard Wording

7.1 Language

TBD.

7.2 Library

TBD.

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 N3351=09-0104, ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++, Jun 2012.

[3] Gabriel Dos Reis, Bjarne Stroustrup, and Alisdair Merideth, *Axioms: aemantic Aspects of C++ Concepts* Technical Report N3351=09-0077, ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++, Jun, 2012.

[4] 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.

[5] Gabriel Dos Reis and Bjarne Stroustrup, "Specifying C++ concepts", *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.

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