Concepts: The Future of Generic Programming

or

How to design good concepts and use them well

Bjarne Stroustrup

Morgan Stanley and Columbia University

www.stroustrup.com

Abstract

I briefly describe concepts (requirements of a template on its template arguments) as defined by the ISO Technical Specification [C++15] and shipped as part of GCC [GCC16]. I aim for an understanding of the aims of concepts, their basic design principles, and their basic ways of use:

§1. The background of the concept design
§2. Concepts as a foundation for generic programming
§3. The basic use of concepts as requirements on template arguments
§4. The definition of concepts as Boolean values (predicates)
§5. Designing with concepts
§6. The use of concepts to resolve overloads
§7. The short-form notations
§8. Language design questions

The use of concepts implies no run-time costs compared to traditional unconstrained templates. They are purely a selection mechanism and after selection, the code generated is identical to traditional template code.

§5 is the heart of this paper. You can find technical details, language design discussions, and more exhaustive tutorial material in the references.

1. A bit of background

In about 1987, I tried to design templates with proper interfaces [Str94]. I failed. I wanted three properties for templates:

- Full generality/expressiveness
- Zero overhead compared to hand coding
- Well-specified interfaces

Then, nobody could figure out how to get all three, so we got

- Turing completeness
- Better than hand-coding performance
- Lousy interfaces (basically compile-time duck typing)

The lack of well-specified interfaces led to the spectacularly bad error messages we saw over the years. The other two properties made templates a run-away success.

The lack of well-specified interfaces bothered me and many others over the years. It bothered me a lot because templates failed to meet the fundamental design criteria of C++ [Str94]. Many tried to find a solution, notably members of the C++ standards committee aiming for C++0x [C++09, Str09], but until recently nobody came up with something that met all three original aims, fitted into C++, and compiled reasonably fast.

Note that the design aims for templates is an example of the general C++ design aims [BS94]:

- Generality
- Zero overhead
- Well-defined interfaces

The solution to the interface specification problem was named "concepts" by Alex Stepanov (§8.8). A concept is a set of requirements on a set of template arguments. The question is how to craft a set of language features to support that idea.

Together with Gabriel Dos Reis and Andrew Sutton, I started to design concepts from scratch in 2009 [Sut11]. In 2011, Alex Stepanov called a meeting in Palo Alto, where a largish group, including Sean Parent and Andrew Lumsdaine, attacked the problem from the user's perspective: What would a properly constrained STL ideally look like? Then, we went home to invent language mechanisms to approximate that ideal [Str12]. That re-booted the standards effort based on a new, fundamentally different, and better approach than the C++0x effort. We now have a ISO TS ("Technical Specification") for concepts [C++15]. Andrew Sutton's implementation has been used for over three years now and is shipping as part of GCC [GCC16].

2. Generic Programming

We need to simplify generic programming in C++. The way we write generic code today is simply too different from the way we write other code. Consider

```
// Traditional code:
double sqrt(double d);  // C++84: accept any d that is a double

double d = 7;
double d2 = sqrt(d);      // fine: d is a double

vector<string> vs = { "Good", "old", "templates" };
double d3 = sqrt(vs);     // error: vs is not a double
```

This is the kind of code we became acquainted with on the first day or so of learning to program. We have a function **sqrt** specified to require a **double**. If we give it a **double** (as in **sqrt(d)**) all is well, and if we give it something that is not a **double** (as in **sqrt(vs)**) we promptly get a helpful error message, such as "a **vector<string>** is not a **double**."

In contrast:

```
// 1990s style generic code:
template<class T> void sort(T& c)    // C++98: accept a c of any type T
{
    // code for sorting (depending on various properties of T,
    // such as having [] and a value type with <
}

vector<string> vs = { "Good", "old", "templates" };
sort(vs);            // fine: vs happens to have all the syntactic properties required by sort

double d = 7;
sort(d);             // error: d doesn't have a [] operator
```

We have problems:

- As you probably know, the error message we get from **sort(d)** is verbose and nowhere near as precise and helpful as my comment might suggest.
- To use **sort**, we need to provide its definition, rather than just its declaration, this differs from ordinary code and changes the model of how we organize code.
- The requirements of **sort** on its argument type are implicit ("hidden") in its function body.
- The error message for **sort(d)** will appear only when the template is instantiated, and that may be long after the point of call.
- The **template<typename T>** notation is unique, verbose, repetitive, and widely disliked.

Using a concept, we can get to the root of the problem by properly specifying a template's requirements on its arguments:

```
// Generic code using a concept (Sortable):
void sort(Sortable& c);      // Concepts: accept any c that is Sortable

vector<string> vs = { "Hello", "new", "World" };
sort(vs);            // fine: vs is a Sortable container

double d = 7;
sort(d);             // error: d is not Sortable (double does not provide [], etc.)
```

This code is analogous to the **sqrt** example. The only real difference is that

- for **double**, a language designer (Dennis Ritchie) built it into the compiler as a specific type with its meaning specified in documentation.
- for **Sortable**, a user specified what it means in code. Briefly, a type is **Sortable** if it has **begin()** and **end()** providing random access to a sequence with elements that can be compared using **<**; §5.

Now we get an error message much as indicated in the comment. That message is generated immediately at the point where the compiler sees the erroneous call (**sort(d)**).

My aim is to make

- simple generic code as simple as non-generic code
- more advanced generic code as easy to use and not that much more difficult to write.

Concepts by themselves do not address the code organization difference; we still need to put templates into headers. However, that is addressed by modules [Rei16]. In a module, a template is represented as a typed abstract graph and to check a call of a template function using concepts only its interface (declaration) as provide by the module is needed.

3. Using Concepts

A concept is a compile-time predicate (that is, something that yields a Boolean value). For example, a template type argument, **T**, could be required to be

- an iterator: **Iterator<T>**
- a random access iterator: **Random_access_iterator<T>**
- a number: **Number<T>**

The notation **C<T>** where **C** is a concept and **T** is a type is an expression meaning "**true** if **T** meets all the requirements of **C** and **false** otherwise."

Similarly, we can specify that a set of template arguments must meet a predicate, for example **Mergeable<In1, In2, Out>**. Such multi-type predicates are necessary to describe the STL and most other application domains. They are very expressive and nicely cheap to compile (cheaper than template metaprogramming workarounds). Students can use this after a lecture or two. You can, of course, define your own concepts (§5) and we can have libraries of concepts. Concepts enable overloading (§6) and eliminate the need for a lot of ad-hoc metaprogramming and much metaprogramming scaffolding code, thus significantly simplifying metaprogramming as well as generic programming.

3.1 Specifying template interfaces

Let's first see how we can use concepts to specify algorithms. Consider a variant of **std::find()** that takes a sequence instead of a pair of iterators.

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

Let's look at it line for line:

- This is a template that takes two template type arguments (nothing new here).
- The first template argument must be a sequence (**Sequence<S>**) and we have to be able to compare elements of the sequence to **value** using the **==** operator (**Equality_comparable<Value_type<S>, T>**).
- This **find()** takes its sequence by reference and the value to be found as a **const** reference. It returns an iterator (nothing new here).

A sequence is something with a **begin()** and **end()** (§5), but to understand the declaration of **find()** that's immaterial.

I have used alias templates to be able to say **Value_type<S>** and **Iterator_of<S>**. The simplest definitions would be:

```
template<typename X> using Value_type<X> = X::value_type;
template<typename X> using Iterator_of<X> = X::iterator;
```

Alias templates have nothing particularly to do with concepts. They are just useful to express generic code. Expect to find such aliases in libraries.

The concept **Equality_comparable** is proposed as a standard-library concept. It requires that its argument supplies **==** and **!=**. Note that **Equality_comparable** takes two arguments. Many concepts take more than one argument: concepts can describe not just types, but also relationships among types. This is essential for much of the STL (for which those relationships are described in the standard) and most other libraries. A concept is not just "the type of a type."

We can try to use this **find()**:

```
void use(vector<string>& vs, list<double>& lstd)
{
        auto p0 = find(vs,"Waldo");      // OK
        auto p1 = find(vs,0.5772);       // error: can't compare a string and a double
        auto p2 = find(lstd,0.5772);     // OK
        auto p3 = find(lstd, "Waldo");   // error: can't compare a double and a string

        If (p0!=vs.end()) { /* found Waldo */ }
        // …
}
```

This is just one example, and quite a simple one, but using only the techniques from that example, we described all the STL algorithms in what is known as "The Palo Alto TM" [Str12]. You can find many more examples of the use of concepts in the Ranges TS [Nie15] (that we expect to evolve into STL2) and [Sut15, Sut16, Sut17]. The Palo Alto TM was just a design document, but the Ranges TS is compiled and tested code.

3.2 A shorthand notation

When requiring a template argument to be a sequence, we can say

```
template<typename Seq>
        requires Sequence<Seq>
void algo(Seq& s);
```

That is, we need an argument of type **Seq** that must be a **Sequence**. I did that by saying "The template takes a type argument; that type argument must be a **Sequence**." That's a bit verbose.

That's not what we actually say when we talk about such code. We say "The template takes a **Sequence** argument" and we can actually write that:

```
template<Sequence Seq>
void algo(Seq& s);
```

That means exactly the same as the longer version above, but it's shorter and matches our thinking better. Similarly, we don't say "There is an animal and it's a chicken" we say "There is a chicken." The rewrite rule is simple and general, for a concept **C**

```
template<C T>
```

means

```
template<typename T>
        requires C<T>
```

We use this simple shorthand for concepts of a single argument. That is, when we are requiring something of a single type. For example, we can simplify

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

to

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

I consider this shorter form a significant improvement in clarity over the longer version. I use explicit **requires**-clauses primarily for multi-type concepts (e.g. **Equality_comparable**) and where a template argument type needs to be referred to repeatedly (e.g., **find**'s **S**). This addresses the frequent and persistent user complaints that the C++ template syntax is verbose and ugly. I agree with those criticisms (§8.6). Making the verbose syntax redundant for simple and frequent examples follows the general design aim of making simple things simple.

4. Defining concepts

Often, you'll find useful concepts, such as **Equality_comparable** in libraries (e.g., the Ranges TS [Nie15]) and we hope to see a set of standard-library concepts, but to see how concepts can be defined consider:

```
template<typename T>
concept bool Equality_comparable =
        requires (T a, T b) {
                { a == b } -> bool;          // compare Ts with ==
```

```
        { a != b } -> bool;          // compare Ts with !=
};
```

The **Equality_comparable** concept is defined as a variable template. To be **Equality_comparable**, a type **T** must provide **==** and **!=** operations that each must return a **bool** (technically, "something convertible to **bool**"). The **requires** expression allows us to directly express how a type can be used:

- **{ a == b }** says that two **T**s should be comparable using **==**.
- **{ a == b } -> bool** says that the result of such a comparison must be a **bool** (technically, "something convertible to **bool**").

A **requires** expression is never actually executed. Instead, the compiler looks at the requirements listed and returns **true** if they would compile and **false** if not. This is obviously a very powerful facility. To learn the details, I recommend Andrew Sutton's paper [Sut16]. Here, I'll just show examples:

```
template<typename T>
concept bool Sequence =
    requires(T t) {
        typename Value_type<T>;      // must have a value type
        typename Iterator_of<T>;     // must have an iterator type

        { begin(t) } -> Iterator_of<T>;   // must have begin() and end()
        { end(t) } -> Iterator_of<T>;

        requires Input_iterator<Iterator_of<T>>;
        requires Same_type<Value_type<T>,Value_type<Iterator_of<T>>>;
    };
```

That is, to be a **Sequence**

- a type **T** must have two associated types **Value_type<T>** and **Iterator_of<T>**. **Value_type<T>** and **Iterator_of<T>** are just ordinary alias templates. Listing those types in the **requires** expression indicates that a type **T** must have them to be a **Sequence**.
- a type **T** must have **begin()** and **end()** operations that each return an appropriate iterator.
- by "appropriate iterator" we mean that **T**'s iterator type must be an **Input_iterator** and **T**'s value type must be the same as its iterator's value type. **Input_iterator** and **Same_type** are concepts from a library, but you could easily write them yourself.

Now, finally, we can do **Sortable** from §2. To be sortable, a type must be a sequence offering random access and with a value type that supports **<** comparisons:

```
template<typename T>
concept bool Sortable =
    Sequence<T> &&
```

```
Random_access_iterator<Iterator_of<T>> &&
Less_than_comparable<Value_type<T>>;
```

Random_access_iterator and **Less_than_comparable** are defined analogously to **Equality_comparable**, so I leave those for the reader write or look up in a library, say, the Range library [Nie15].

Often, we want to make requirements on the relationship among concepts. For example, I defined the Equality_comparable concept to require a single type, but it is usually defined to handle two types:

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

This allows comparing **int**s to **double**s and **string**s to **char***s, but not **int**s to **string**s.

5. Designing with concepts

What makes a good concept? Ideally, a concept represents a fundamental concept in some domain, hence the name "concept." A concept has semantics; it means something; it is not just a set of unrelated operations and types. Without an idea of what operations mean and how they relate to each other we cannot write generic code that works for all appropriate types.

Unfortunately, we cannot (yet) state the semantics of a concept in code (see the Palo Alto TM [Str12] for some ideas). It follows that a guarantee that *all* types accepted by concept checking will work correctly is impossible: they may have exactly the syntactic properties required, but have the wrong semantics. This is nothing new: Similarly, a function taking a double can interpret it differently from what the caller expects. Consider **set_speed(4.5)**. What does this mean? Is **4.5** supposed to be in m/s or maybe miles/hour? Is **4.5** an absolute value, a delta to the current speed, or possibly a factor of change?

I suspect that *perfect* checking for all code will forever elude us; as we get better tools, developers will create more subtle bugs, but there are techniques to make bugs less likely to escape our notice.

5.1 Type/concept accidental match

First, let me make a common design mistake. It will make it easier to illustrate good design. I recently saw a concept version of an old OO problem:

```
template<typename T>
concept bool Drawable = requires(T t) { t.draw(); } ;
```

```
class Shape {
        // …
        void draw();     // light up selected pixels on the screen
};

class Cowboy {
        // …
        void draw();     // pull deadly weapon from holster
};

template<Drawable D>
void draw_all(vector<D*>& v)   // ye olde draw all shapes example
{
        for (auto x : v) v->draw();
}
```

This **draw_all** would, like its OO counterpart, accept a **vector<Cowboy*>** with surprising and potential damaging effects. This problem of "accidental match" (in overloading and class hierarchies) is widely feared, rare in real-world code, and easily avoided for concepts.

Ask yourself: What fundamental concept does "has a draw member function taking no argument" represent? There is no good answer. A cowboy might make a good concept in a games context and a drawable shape a good concept in a graphics context, but we would never confuse them. A shape has several more essential properties than just "can be drawn" (e.g. "has location", "can be moved" and "can be hidden") and so has a cowboy (e.g., "can ride a horse", "likes booze", and "can die"). A concept that requires the full set of carefully specified essential properties is unlikely to be mistaken for another.

My rule of thumb is to avoid "single property concepts." For that reason, **Drawable** is instantly suspicious. It is a good example of something that should not be exposed to application builders. To be more realistic, people sometimes get into trouble defining something like this:

```
template<typename T>
concept bool Addable = requires(T a, T b) { { a+b } -> T; } ;
```

They are then often surprised to find that **std::string** is **Addable** (**std::string** provides a **+**, but that **+** concatenates, rather than adding). **Addable** is not a suitable concept for general use, it does not represent a fundamental user-level concept. If **Addable**, why not **Subtractable**? (**std::string** is not **Subtractable**, but **int*** is). Surprises are common for "simple, single property concepts." Instead, define something like **Number**:

```
template<typename T>
concept bool Number = requires(T a, T b) {
        { a+b } -> T;
        { a-b } -> T;
        { a*b } -> T;
```

```
        { a/b } -> T;
        { -a } -> T;

        { a+=b } -> T&;
        { a-=b } -> T&;
        { a*=b } -> T&;
        { a/=b } -> T&;

        { T{0} };          // can construct a T from a zero

        // …
};
```

This is extremely unlikely to be matched unintentionally.

A good useful concept supports a fundamental concept (pun intended) by supplying the set of properties – such as operations and member types – that a domain expert would expect. The mistake made for **Drawable** and **Addable** was to use the language features naively without regard to design principles.

Please note that **Number** is just an illustrative example. If I had to describe C++ arithmetic types, I'd need to address signed/unsigned issues and how to handle mixed-mode arithmetic. If I wanted to describe computer algebra I'd probably start with group theory: monoid, semi-group, group, etc.

5.2 Semantics

How do we find such a useful set of properties to design a useful concept? Most application areas already have them. Examples are

- C/C++ built-in type concepts: arithmetic, integral, and floating (yes, C has concepts!)
- STL concepts like iterators and containers
- Mathematical concepts like monoid, group, ring, and field
- Graph concepts like edges and vertices; graph, DAG, etc.

Note that these pre-existing concepts all have semantics associated with them (Alex Stepanov once said "concepts are all about semantics"). We have found that trying to specify semantics for a new concept is an invaluable help in designing useful concepts and stable interfaces [Sut11]. Often asking "can we state an axiom?" leads to significant improvements of a draft concept.

The first step to design a good concept is to consider what is a complete (necessary and sufficient) set of properties (operations, types, etc.) to match the domain concept, taking into account the semantics of that domain concept.

5.3 Ideals for concept design

What makes one concept better than another? Fundamentally, concepts are there to allow us to state fairly abstract ideas in code, so that we can write better generic code. One way to look at this is that concepts help us to make algorithms and types "plug compatible":

- we want to write algorithms that can be used for a wide variety of types, and
- we want to define types that can be used with a wide variety of algorithms.

For example: we want to define number types that can be used for our numeric algorithms and algorithms that can be used with all our containers.

This has an important implication on what has become a standard generic-programming technique: We often try to specify the requirements of an algorithm to be the absolute minimum. This is *not* what we do to design the most useful concepts. As an example, consider a simplified version of **std::accumulate**:

```
template<typename Iter, typename Val>
Val sum(Iter first, Iter last, Val acc)
{
        while (first!=last)  {
                acc += *first;
                ++first;
        }
        return acc;
}
```

"Classical GP design" would tempt us to constrain **sum** minimally like this

```
template<Forward_iterator Iter, typename Val>
        requires Incrementable<Val,Value_type<Iter>>
Val sum(Iter first, Iter last, Val acc)
{
        while (first!=last)  {
                acc += *first;
                ++first;
        }
        return acc;
}
```

Incrementable would be a concept that simply required the **+=** operator to be present. This would (apparently) minimize the work needed by someone designing types that might be used as **sum** arguments and maximize the usefulness of the **sum** algorithm. However,

- We "forgot" to say that **Val** had to be copyable and/or movable
- We cannot use this **sum** for a **Val** that provides **+** and **=**, but no **+=**

- We cannot modify this **sum** to use **+** and **=** instead of **+=** without changing the requirements (part of the functions interface)

This is not very "plug compatible" and rather ad hoc. That kind of design leads to programs where

- Every algorithm has its own requirements (a variety that we cannot easily remember).
- Every type must be designed to match an unspecified and changing set of requirements.
- When we improve the implementation of an algorithm, we must change its requirements (part of its interface), potentially breaking code.

In this direction lies chaos. Thus, the ideal is not "minimal requirements" but "requirements expressed in terms of fundamental and complete concepts." This puts a burden on the designers of types (to match concepts), but that leads to better types and to more flexible code. For example, a better sum would be

```
template<Forward_iterator Iter, Number<Value_type<Iter>> Val>
Val sum(Iter first, Iter last, Val acc)
{
        while (first!=last)  {
                acc += *first;
                ++first;
        }
        return acc;
}
```

Note that by requiring a **Number** we gained flexibility. We also "lost" the accidental ability to use **sum** to concatenate **std::string**s and to sum a **vector<int>** into a **char***:

```
void poor_use(vector<string>& vs, vector<int>& vi)
{
        std::string s;
        s = sum(vs.begin(),vs.end(),s);   // error: a string is not a number
        char* p = nullptr;
        p = sum(vi.begin(),vi.end(),p);   // error: a pointer is not a number
        // …
}
```

Good! Strings and pointers are not numbers. If we really wanted that functionality, we could easily write it deliberately.

To design good concepts and to use concepts well, we must remember that an implementation isn't a specification – someday, someone is likely to want to improve the implementation and ideally that is done without affecting the interface. Often, we cannot change an interface because doing so would break user code. To write maintainable and widely usable code we aim for semantic coherence, rather than minimalism for each concept and algorithm in isolation.

5.4 Constraints

The view of concepts described here is somewhat idealistic and aimed at producing "final" concepts to be used by application builders in mature application domains. However, incomplete concepts can be very useful, especially during earlier stages of development in a new application domain. For example, the **Number** concept above is incomplete because I "forgot" to require **Number**s to be copyable and/or movable. Mature libraries provide precedence and supporting concepts to avoid such incompleteness.

Even as it is, the use of **Number** saves us from many errors. It catches all errors related to missing arithmetic operations. But the specification of **sum** using **Number** does not save us from an error if a user calls **sum** with a type that provides the requires arithmetic operations, but cannot be copied or moved. However, we still get an error: we just get one of the traditional late and messy error messages we have been used to for decades. The system is still type safe. I see "incomplete concepts" as an important aid to development and to gradual introduction of concepts (§8.2).

Concepts that are too simple for general use and/or lack a clear semantics can also be used as building blocks for more complete concepts.

Sometimes, we call such overly simple or incomplete concepts "constraints" to distinguish them from the "real concepts" [Sut11].

5.5 Matching types to concepts

How can a writer of a new type be sure it matches a concept? That's (surprisingly?) easy: We simply **static_assert** the desired concept matches. For example:

```
class My_number { /* … */ };
static_assert(Number<My_number>);
static_assert(Group<My_number>);
static_assert(Someone_elses_number<My_number>);

class My_container { /* … */ };
static_assert(Random_access_iterator<My_container::iterator>);
```

After all, concepts are simply predicates, so we can test them. They are *compile-time* predicates, so we can test them at compile time. Note that we do not have to build the set of concepts to be matched into the definition of a type. This is not some kind of hierarchy design that requires perfect foresight or refactoring each time a new use is discovered. This is critical to preserve the compositional benefits of generic code as compared to object-oriented hierarchies. The **static_assert**s don't even have to be in the type designer's code. A user might like to add such tests to catch mismatches early and in specific places in the code. If done, doing so without modifying library code is essential.

5.6 Gradual introduction of concepts

How can we start using concepts? In real-world systems, we essentially never have the luxury of starting from scratch. We depend on other people's code (e.g., libraries) and if we are updating

an existing code base, other people's code relies on our code. In particular, we typically rely on libraries (e.g., a vendor's standard library or a popular networking library) and for years such libraries may not be using concepts. So, we find our code calling templates that do not use concepts. For example:

```
template<Sortable S>
void sort(S& s)
{
        std::sort(s.begin(),s.end());
}
```

Our **sort** requires that **s** is sortable, but what does **std::sort** require? In this particular case, we can look in the standard, but in general, we the details of what an implementation uses is not precisely specified. Furthermore, an implementation may contain "scaffolding code" for statistics gathering, logging, debug aids, assertions, call to platform-specific libraries and facilities. In other words, at the point of call, we cannot be sure that the implementation does not use facilities that we have not required of our callers. Furthermore, the implementation of such templates change over time. The implementation can even differ based on build options.

However, that doesn't matter much because we get complete type checking as ever, just late (instantiation time) and with ghastly error messages. The important point is that we can update our code to use concepts without doing a (typically impossible) complete upgrade of all the code we rely on. As our suppliers upgrade their template interfaces, our error checking moves forward to the point of call and the quality of error messages improves.

If you need to be able to compile with compilers the support concepts and compilers that do not, some workaround are needed. One obvious technique is to use macros. Requires clauses can be handled by commenting them out in older compilers:

```
#ifdef GOOD_COMPILER
#define REQUIRES requires
#elseif
#define REQUIRES //
#endif
```

If the short-hand notation is used, the concepts need to be listed:

```
#ifdef GOOD_COMPILER
#define SORTABLE Sortable
#define ITERATOR Iterator
#elseif
#define Sortable auto
#define ITERATOR auto
#endif
```

To get a rough equivalent of concept-based overloading (§6), **enable_if** can be used. This works, but the reports I hear is that it is quite painful to maintain for real-world code (e.g., the Range library [Nie15]). In particular, remember to use both the positive and negative checks.

6. Concept overloading

Generic programming relies on using the same name for operations that can be use equivalently for different types. Thus, overloading is essential. Where we cannot overload, we need to use workarounds (e.g., traits, **enable_if**, or helper functions). Concepts allows us to select among functions based on properties of the arguments given. Consider a simplified version of the standard-library **advance** algorithm:

```
template<typename Iter> void advance(Iter p, int n);
```

We need different versions of **advance**, including

- A simple one for forward iterators, stepping through the sequence one step at a time.
- A fast one for random-access iterators to take advantage of the ability to advance the iterator to an arbitrary position in the sequence in one operation.

Such compile-time selection is essential for performance of generic code. Traditionally, we have implemented that using helper functions and tag dispatch [Str94], but with concepts the solution is simple and obvious:

```
void advance(Forward_iterator p, int n) { while(n--) ++p; }

void advance(Random_access_iterator p, int n) { p+=n; }

void use(vector<string>& vs, list<string>& ls)
{
        auto pvs = find(vs,"foo");
        advance(pvs,2);         // use fast advance

        auto pls = find(ls,"foo");
        advance(pls,2);         // use slow advance
}
```

How does the compiler figure out how to invoke the right **advance**? We didn't tell it directly. There is no defined hierarchy of iterators and we did not define any traits to use for tag dispatch.

Overload resolution based on concepts is fundamentally simple:

- If a function matches the requirements of one concept only, call it
- If a function matches the requirements of no concept, the call is an error
- If the function matches the requirements of two concepts, see if the requirements of one of those concepts is a subset of the requirements of the other.
    - If so, call the function with the most requirements (the strictest requirements).

- If not, the call is an error (ambiguous).

In the **use** example, a **Random_access_iterator** has more requirements than **Forward_iterator** ("**Random_access_iterator** is stricter than **Forward_iterator**") so we pick the fast **advance** for **vector**'s iterator. For **list**'s iterator, only **Forward_iterator** matches, so we use the slow **advance**.

Random_access_iterator is stricter than **Forward_iterator** because it requires everything **Forward_iterator** does plus additional operators such as **[]** and **+**.

There are a few technicalities related to the exact comparison of concepts for strictness, but we don't need to go into those to use concept overloading. What is important is that we don't have to explicitly specify an "inheritance hierarchy" among concepts, define traits classes, or add tag dispatch helper functions. The compiler computes the real hierarchies for us. This is far simpler, more flexible, and less error-prone.

Concept-based overloading eliminates a significant amount of boiler-plate from generic code and metaprogramming code (most uses of **enable_if**). The general principle here is that we should not force a programmer to do what the compiler can do better. Concept-based overloading ensures that the code follows general and widely-used resolution rules, rather than differing and potentially subtle implementation details (such as remembering to use both the positive and negative forms of **enable_if** when expressing overloading based on a property of a type).

One obvious question: How do we distinguish types that are syntactically identical, but differ in their semantics? The standard example of that is **Input_iterator** and **Forward_iterator** that differ only in that repeated traversal is allowed for **Forward_iterator**. The simplest answer is "don't do that; add an operation to one of the types to make them distinguishable." A more conventional and complicated answer is "use a traits class." The latter is what we do when we can't modify either type. In the **Input_iterator** and **Forward_iterator** case, we could actually distinguish because an **Input_iterator** is only moveable and not copyable (use the **is_copy_constructible<T>** trait), but that's subtle.

7. The short-form notations

One of my design aims for C++ is to make simple things simple. Thus, we have the shorthand notation (§3.1) to avoid annoying repetition and more succinctly state our requirements. Consider:

```
template<typename Seq>
        requires Sortable<Seq>
void sort(Seq& s);
```

We can shorten that to:

```
template<Sortable Seq>
void sort(Seq& s);
```

However, that still doesn't get us to the ideal equivalence to "ordinary non-generic" code as articulated in §2:

```
void sort(Sortable& s);
```

To get there, we have a further "rewrite rule." The short form and the shorthand forms are simply equivalent to the long, very explicit form above. We use the shortest form for the simplest cases, and the other two forms – often in combination – when we need to express more complicated requirements, especially for requirements involving more than one template argument. For example:

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

Why bother? The long form is unpleasantly verbose for most code and ever since the introduction of templates the verbosity ("heaviness") of the template syntax has been a source of constant complaints from users. However, we need the long form expressing complex requirements: to keep it short, the shortest form is deliberately not perfectly general.

This design follows "the onion principle." The default is short and simple. Whenever you need to do something that cannot be expressed that simply, you peel one layer off the onion. Each layer gives you more flexibility, and make you cry more (because of the added work and the added opportunities for mistakes). The presence of both old-style (perfectly general) **for** loops and (simpler and less error-prone) range-**for** loops is another example of that principle.

The three forms match the way we speak about functions:

```
        // the template argument must be a type, and
        // that type must be a sequence, and
        // s must be a reference to that type:
template<typename Seq>
        requires Sequence<Seq>
void algo(Seq& s);

        // the template argument must be a sequence, and
        //  s must be a reference to that sequence:
template<Sequence Seq>
void algo(Seq& s);

        // s must be a reference to a sequence:
void algo(Sequence& s);
```

We use the shorter forms unless we have reason not to.

7.1 auto arguments

The short form also offers an unconstrained variant:

```
void f(auto x);          // take argument of any type
```

That is, **auto** is the least constrained concept. I first proposed **auto** for arguments and return types in 2001 [Jar02] and C++14 supports it for lambdas.

We could also define something like that using a trivial constraint

```
concept bool Any = true;      // every type is an Any
void g(Any x);                // take argument of any type
```

These two ways of specifying unconstrained arguments differ in one small useful way:

```
void ff(auto x, auto y);  // x and y can be of different type
void gg(Any x, Any y);    // x and y must take the same type
```

That is, **gg** represents the STL iterator pair style and many other "sets of arguments of the same type" styles, whereas **ff** represents completely unrelated template arguments. Both styles are useful and common. For example:

```
void user(vector<string>& vs, list<double>ld)
{
        ff(&vs,&ld);                // keep a list of containers (of arbitrary types)
        gg(vs.begin(),vs.end());    // OK: two iterators of the same type
        gg(vs.begin(),ld.end());    // error: two iterators to different types
}
```

7.2 Readability

I had expected people using concepts to praise the expressiveness of concepts and the improved error messages they enable. Those aspects were mentioned (by students and professional developers), but again and again people emphasized the vastly improved readability of code using concepts. I should have expected that because fundamentally concepts enable better interface specification and good interfaces simplify understanding.

Interestingly, such comments came from people who expressed different preferences in notations (and sometimes dislike of other notations): some prefer requires clauses, some prefer concepts instead of **typename**, and some prefer to use the shortest form whenever they can. Different people simply find one or more notations suitable for their needs. I should have expected that also because I understand that different people have different needs.

In both cases, I underestimated the importance of readability. I hear three aspect emphasized:

- Declarations are more precise and informative. Declarations using concepts are simply easier to read and more trustworthy than declarations using descriptive names for fully

generic types plus comments. Also, declarations using concepts are on average shorter than the workarounds (at least where comments are used to document constraints).

- Replacing **auto** with a concept at a call point removes uncertainty about the nature of a result. I see this largely as a response to overuse of **auto**, but without concepts there are few alternatives to widespread use of **auto** in generic code that is not purely functional. For example, **if (auto x = foobar(z))** is far less readable than **if (InputChannel x = foobar(z))**.
- Concepts eliminate unreadable workarounds and complicated boilerplate. Yes, we are not supposed to look into template definitions and macros, but we often do (to understand, to debug, and to improve) and often workarounds bleed into interfaces (e.g., in the form of **enable_if**).

Obviously, these observations are aesthetic judgments individuals, rather than experimentally verified facts, but I think they are significant. The aesthetic judgment of programmers matters and the volume of positive comments on readability surprised me.

I believe that the readability issue contributes to the improvements of design and maintainability attributed to concepts.

8. Language design questions

I added this section because people often ask about language design and sometimes suggest alternatives to or radical modifications of the current concept design. This article is not a language design document, so the discussion here is brief. It tries to

- satisfy natural curiosity about language design decisions
- reassure potential users that obvious alternatives have been considered
- show how the design follows the general design principles for C++

8.1 Do we really need concepts?

From very early in the design and use of templates, I and others realized that some forms of interface checking could be expressed using templates themselves without added language support [Str94]. Boost concept check is an example. Today, with **static_assert**, **constexpr** functions, **constexpr if** [Vut16], a plethora of standard-library type traits, and mature template metaprogramming techniques, I hear many variants of this argument. Typically, what is achieved leads to instantiation-time checking, which is less than ideal. More fundamentally, it lowers the level of programming, making the foundation of generic programming depend on a variety of ad hoc metaprogramming libraries. Claiming that relying on low-level primitives is sufficient is like (correctly) pointing out that we don't absolutely need **for** statements, **while** statements, and range-**for** statements when we have **if** and **goto**. Similarly, given pointers to functions, virtual functions and lambdas are not absolutely necessary. However, C++ is not meant to be merely assembly code for (template) metaprogramming:

- "Concepts" is not some advanced feature for experts bolted on top of compile-time duck typing.
- "Concepts" is not just syntactic sugar for type traits and **enable_if**.
- "Concepts" is a foundational feature that in the ideal world would have been present in the very first version of templates and the basis for all use.

I'm pretty sure that if we had had concepts in 1990, templates and template libraries would have been significantly simpler today.

Note that in template argument declarations, **typename** is simply the least demanding concept: it just requires the template argument to be a type (and not a non-type value). Thus, the old pre-concept templates integrate smoothly with concepts. If constraints are not needed, we just don't use concepts or use concepts providing very minimal constraints. Examples are very general AST-manipulating template metaprogramming and template arguments for which requirements are exclusively expressed as relations to other template arguments (e.g., see **find()** in §3.2).

8.2 Definition checking

Concepts currently do not prevent a template from using operations that are not specified in the requirements. Consider:

```
template<Number N>
void algo(vector<N>& v)
{
        for (auto& x : v) x%=2;
}
```

Our **Number** concept does not require **%=**, so whether a call of **algo** succeeds will depend not just on what is checked by the concept, but on the actual properties of the argument type: does the argument type have **%=**? If not, we get a late (instantiation time) error.

Some consider this a serious error. I don't: not checking template definitions against the template's concepts was a deliberate design choice. We (Gabriel Dos Reis, Andrew Sutton, and I) know how to implement definition checking with the concepts as currently specified. We have done analysis and experiments (e.g., [Rei12]), but we very deliberately decided not to include such a feature in the initial concept design:

- We didn't want to delay and complicate the initial design (that would delay getting essential feedback and delay library building).
- We estimate that something like 90% of the benefits of concepts are in the value of improved specification and point-of-use checking.
- The template implementer can compensate through normal testing techniques.
- As ever, type errors are *always* caught, only uncomfortably late.
- By checking definitions, we would complicate transition from older, unconstrained code to concept-based templates.
- By checking definitions, we would be unable to insert debug aids, logging code, telemetry code, performance counters, and other "scaffolding code" into a template without affecting its interface.

The last two points are crucial:

- A typical template calls other templates in its implementation. Unless a template using concepts can call a template from a library that does not, a library with the concepts cannot use an older library before that library has been modernized. That's a serious problem, especially when the two libraries are developed, maintained, and used by more than one organization. Gradual adoption of concepts is essential in many code bases.
- Scaffolding code (both templates and non-template code) is very common, and it changes during the lifetime of a library. If the interface must change to accommodate, say, logging, we have a maintenance issue of the first order.

Note that a constrained template (a template using concepts) can call an unconstrained template. In that case, errors in the implementation are not found until instantiation time. Similarly, an unconstrained (traditional template) can call a constrained template. In that case, usage errors are not found until instantiation time. The former implies that concepts can be introduced "top down" whereas the latter implies that most benefit arise from doing that.

So, we know how to do "definition checking," but we won't do it until those two problems are solved. There are obvious possible solutions, such as an indicator in a template function body/implementation (not in its declaration/interface) that it will use facilities not guaranteed by the concepts, but any such mechanism would have to be seriously analyzed and tested. It might very well not be worth the effort. It is also worth remembering one of the fundamental rules that guided the C++ design: *"It is more important to allow a useful feature than to prevent every misuse"* [Str94]. That is one of the rules that distinguishes C++ from many other languages. There are ways to prevent bad programming beyond language rules, but gradual adoption and scaffolding code must somehow be enabled by the language.

8.3 Separate compilation of templates

Complete separate compilation of a template presupposes definition checking and the most obvious implementation involves lots of indirect function calls, which would kill performance.

The obvious alternative is a semi-compiled form of templates as part of a module system [Rei16].

8.4 Multiple notations

Concepts provide multiple ways of saying things. I'm fine with that. In fact, I consider it ideal (§7), but there are people devoted to the notion of "only one way." That notion of simplicity, inevitably leads to either limitation of expressiveness (e.g., only the short form of concept use) or systematic verbosity (e.g., only the long form of concept use).

Concepts can be defined as template variables or as template functions. For example:

```
template<typename T>
concept bool Eq1 =
        requires (T a, T b) {
                { a == b } -> bool;        // compare Ts with ==
                { a != b } -> bool;        // compare Ts with !=
        };
```

and

```
template<typename T>
concept bool Eq2() {
        return requires (T a, T b) {
                { a == b } -> bool;        // compare Ts with ==
                { a != b } -> bool;        // compare Ts with !=
        };
}
```

Unfortunately, the syntax for using a variable template and the syntax for calling a template function differ. This has nothing to do with concepts, but it leads to curiosities like this:

```
template<typename T> requires Eq1<T> void f(T&);
template<typename T> requires Eq2<T>() void g(T&);
```

Having to remember whether to add **()** or not is a nuisance. However, calling a function in C++ requires **()** and getting the value of a variable does not. The reason for allowing both variables and functions is generality. It follows from the way expressions are defined in C++.

Note that C++ supports overloading for functions, so if you need two concepts with the same name, you have to use the functional form. So far, that has been rare.

Why do we require programmers to write **concept bool** instead of plain **concept** (implying **bool**)? We wanted **concept** to be followed by an ordinary definition, rather than inventing something new, and definitions start with a type. There is one exception to that rule in the C++ grammar: we don't specify a type for constructors, destructors, and conversion operators. This caused a bit of confusion and complexity and a few complaints, so we decided not to follow that precedent.

The reasons for the notational alternatives when using concepts in template declarations are outline in §7.

8.5 Opt in

The concept design follows the principle that we should not force the programmer to say things that the compiler already knows (and often knows better than the programmer). This leads to shorter, cleaner code, and fewer errors.

- You don't opt into using concept-based overloading, just as you don't opt into using ordinary overloading. This is consistent, not verbose, and in the spirit of C++. Some people have become used to opting into facilities by adding to traits. That has its charms, but it is basically a workaround and a spurious difference between generic programming and "ordinary programming." It's added work for the applications programmer and an opportunity to make mistakes.
- You don't opt into or explicitly define a hierarchy of relations among concepts. The compiler computes the proper relations among concepts and applies the very simple resolution mechanisms (§6) to uses. Requiring the specification of an explicit concept

hierarchy would limit flexibility, require it to be feasible for a concept to be part of several hierarchies, and/or require more foresight from library designers.

- You don't opt into having a type match a concept: If a type has the syntactic properties required by the concept, it matches. That is, you don't have to litter your code with "modeling declarations" such as "**vector<T>** models **Container**" or "**List<T>::iterator** isa **Bidirectional_iterator**". Requiring that would complicate library use, (unless checked by the compiler) could be a source of errors, and/or would require more foresight from library designers.

In the rare case where two concepts are identical syntactically but differ semantically (e.g., **Input_iterator** and **Forward_Iterator** are almost indistinguishable), we need to do something to disambiguate: we either disambiguate them by adding an operation or a member type or use a traits class in the implementation of operations on them (§6).

If you want to guarantee that a type meets a concept, use a **static_assert** (§5.5).

8.6 Can't spot the templates

Consider

```
void sort(Sortable&);
```

Some experienced C++ programmers worry that there is no syntactic clue that this is a template. Others, like me, consider it ideal: finally generic functions can be dealt with just like other functions! Note that we have had that property for operators "forever." I have taught concepts to dozens of students. Students don't worry. They either say "cool!" or (more often) just take it for granted. They don't get confused or write particularly bad code using this notation. The big problems and confusions are elsewhere – in areas that have been established C++ for decades. The newer features, such as concepts, are simpler to use than their more established workarounds.

Initially, I too worried about the potential for confusion, and wondered if we needed a naming convention or a syntactic clue. I tried **Sortable_c**, similar to the way some (C style) code use **foo_t** to distinguish a type **foo** from a variable **foo**, and **cSortable**, but after a while that became tedious and it was always ugly – a kind of reverse Hungarian notation. It didn't make the code easier to deal with.

Furthermore, we already introduced the equivalent "erasure" of the notational distinction between type names and template names when we (finally!) decided to allow template arguments to be deduced from constructor arguments [Spe16]. We can now say

```
pair p { 9.2,4};
```

Rather than

```
pair<double,int> p {9.2,4};
```

Interestingly, I did not hear suggestions to rename **pair** (and most other templates) to something like **pair_tmpl** to increase readability. Nor did I hear suggestions that **pair** needed a prefix, e.g., **template**, to tell the user that it is a template.

Note that in C++, we say

```
void f(Node*);
```

rather than

```
void f(struct Node*);
```

This rarely causes confusion and I have not heard complaints about that since the earliest days of C++ (say 1982) when a few people considered the added **struct** helpful, as it was familiar from C. When dealing with real code, people know whether a name refers to a variable, a type, a template, or a concept. So do compilers.

I suspect that text coloring will offer further help to human readers to distinguish types, templates, and concepts, but basically, this is a non-problem. I ascribe the worries about notation and about possible "confusion between types and templates" to the well know phenomenon of people imagining problems with new language features and wondering if heavy-handed syntax or notational conventions are needed to address the imagined problems. As time goes by and the novel facilities become familiar, concise notation invariably becomes preferred. My initial template design did not have the prefix **template<typename T>** syntax. It was introduced primarily to assure "worriers." It is now widely disliked and often ridiculed as an example of verbosity and poor design.

8.7 Should concepts be a kind of classes?

Based on experience with other languages and experimentation with C++0x concepts, some people became convinced that concepts should be defined similar to classes. That is, as a list of declarations. Obviously, the designers of concepts disagree. The use patterns we use are more flexible, shorter to write for real cases, and handle overloading and implicit conversions better [Rei06]. However, since use patterns are more general than concepts based on declarations, we can actually define concepts in terms of signatures. We don't recommend this technique, but here is one way you can define a concept of a type required to be derived from **Base**, have a member function **int f(double)**, and a member **int m**:

```
template<class T>
concept bool Hack =
 requires(T t, Base* p, int(T::*pp)(double), int* ppp) {
    { &t==std:addressof(t) };      // make sure operator & isn't overloaded in a nasty way
    { p = &t };                    // T is derived from Base
    { pp = &T::f };                // T has a member int f(double)
    { ppp = &t.m};                 // T has a member int m
 };
```

This does not address overloaded functions, but many languages based on signatures don't handle overloading well, and, after all, this is not a technique we recommend for general use.

It is not uncommon to approach a new language or a new language feature by trying to use it in a style familiar from another language. In fact, it seems almost inevitable: "You can write Fortran" in any language." Similarly, you can write "Java", "C", "C#", "Haskel", etc. in C++, but I encourage people to try not to fall into that trap when trying out concepts (e.g., basing constraints on explicit hierarchies and base classes).

A long time ago Andrew Koenig did an interesting experiment (which, unfortunately, I don't think he wrote up). He took some simple C++ programs and transcribed them to ML and some simple ML programs and transcribed them to C++. The transcriptions were uniformly ugly, verbose, and slow. Then, he looked at the problems that those small programs were written to solve, and solved them in a native style in "the other language." These programs were in all cases, reasonably nice and fast. Try to use concepts as they were designed to be used, at least initially. I expect that they are general enough to find uses beyond my imagination, and that would be good. However, do not judge them based on a simple transcription on something from another language.

8.8 Concepts are not type classes

Comments on earlier drafts of this paper have convinced me that a historical note is needed to address confusions and misconceptions.

As mentioned in §1 and [Str94], when I designed templates in 1987-8, I would have liked to constrain template arguments. This is an idea that goes back at least as far as Clu [Lis77] and probably further. It seemed an obvious idea at the time. The reason I backed out was a nasty combination of implementation problems and notational problems. In particular, I did not think that any type system that implied the use of indirect function calls was acceptable in the context of systems programming. I still don't. So, like templates, concepts do not rely on indirect function calls. Any run-time overhead would lead to disuse in critical areas. Concepts are not meant to be abstract classes: If you want a vtable, you know where to find it.

Also, implicit conversions, general overloading, and specialization don't mesh well with concepts designed conceptually to be tables (sets) of functions (as tried for C++0x). Concepts are not classes in any common Computer Science sense of the word "class."

A C++ concept is a compile-time predicate on zero or more template argument type argument or value arguments:

- Operational requirements for concepts are specified in terms of usage patterns ("valid expressions"), rather than function signatures §4.
- Concepts are specified as general predicates (Boolean expressions), including functions returning **bool**.
- A type does not need to be explicitly defined to match a concept; the match is deduced.
- Concepts can take value arguments (rather than just type arguments).
- Concepts can take many arguments.

- Concept functions can be overloaded.
- Algorithms can be overloaded on concepts.
- Concepts do not by default constrain implementations; §8.2.
- Concepts are not defined as members of hierarchies; relations among concepts are deduced.
- Concepts can constrain template template argments.

This differs from type classes (e.g., Haskell type classes). We know that multi-parameter type classes have been discussed and used in dialects of Haskell for a long time, but since they are not officially part of Haskell and controversial in places, I list them here as a difference. Multi-argument concepts have always been an integral part of the C++ notion of concepts. The STL – designed and implemented before 1984 by Alex Stepanov – was designed with an eye on concepts [Kap81] and cannot be properly constrained without multi-type concepts.

Please note that two designs aiming to solve similar problems can differ in significant ways for good reasons. I believe that to be the case for type classes and concepts. Haskell type classes would not serve well in the context of C++ and I doubt that C++ concepts would be ideal for Haskell. Conversely, referring to two different solutions to a problem by the same name can cause confusion. If I had thought that some form of type classes would have solved the problem of elegantly and efficiently constraining C++ templates, I would have proposed them. The C++0x design tried that [Gre06, Str09].

Concepts, as defined for C++, goes back to Alex Stepanov's work on generic programming staring in the late 1970s and first documented under the name "Algebraic structures" in [Kap81]. Note that's almost a decade before the design of Haskell. Alex used the name "concept" in lectures in the late 1990s and documented it in [Den98]. The current use of predicates relying of usage patterns to describe operations has its origins in the work of Stroustrup and Dos Reis in the 1990s and early 2000s and documented in [Rei06]. The approach is even mentioned in D&E [Str94], but I don't remember when I first experimented with it. The reason for using usage patterns is to handle implicit conversions and overloading. Though we knew of Haskell and ML, they were not significant influences on the current C++ design.

Conclusions

Concepts complete C++ templates as originally envisioned. I don't see them as an extension but as a completion.

Concepts are quite simple to use and define. They are surprisingly helpful in improving the quality of generic code, but their principles – and not just their language-technical details – need to be understood for effective use. In that, concepts are similar to other fundamental constructs, such as functions, classes, and templates. Compared to unconstrained templates, there are no run-time overheads incurred by using concepts.

Concepts are carefully designed to fit into C++ and to follow C++'s design principles:

- Provide good interfaces

- Look for semantic coherence
- Don't force the user to do what a machine does better
- Keep simple things simple
- Zero-overhead

Don't confuse familiarity and simplicity. Don't confuse verbosity with "easy to understand." Try concepts! They will dramatically improve your generic programming and make the current workarounds (e.g., traits classes) and low-level techniques (e.g., **enable_if** – based overloading) feel like error-prone and tedious assembly programming.

Acknowledgements

Similarities to the writings of Andrew Sutton are not accidental. We have worked together on concepts for many years and share some favorite examples that we have used in the design of concepts and to explain concepts. Thanks Andrew!

Also thanks to Mircea Baja, Gleb Dolgich, Howard Hinnant, Peter Juhl, Paul McJones, Herb Sutter, Andrew Sutton, Benedek Thaler, J.C. van Winkel, and Sergey Zubkov for constructive comments on drafts of this paper.

References

- [C++09] C++0x working paper containing C++0x concepts. June 2009.
- [C++15] A final draft of the Concepts TS. WG21-N4549. ISO/IEC TS 19217. 2015.
- [GCC16] GCC 6.0 supports concepts.
- [Deh98] J. C. Dehnert and A. Stepanov: Fundamentals of Generic Programming Dagstuhl Seminar on Generic Programming.1998. Springer LNCS.
- [Gre06] D. Gregor, J. Jarvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. OOPSLA'06.
- [Jar02] J. Jarvi, B. Stroustrup, D. Gregor, J. Siek: Decltype and auto. N1478/03-0061.
- [Kap81] D. Kapur, D.R. Musser, and A.A. Stepanov: Operators and Algebraic Structures Proc. the 1981 conference on Functional programming languages and computer architecture.
- [Lis77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert: Abstraction mechanisms in CLU . CACM, 20(8):564–576, 1977.
- [Nie15] E. Niebler: Ranges TS. WG21 N4569. 2015.
- [Rei06] G. Dos Reis and Bjarne Stroustrup: Specifying C++ Concepts. POPL'06.
- [Rei12] G. Dos Reis. A System for Axiomatic Programming. ICM'12.
- [Rei16] G. Dos Reis: A Module System for C++ (Revision 4) . WG21 P0142R0. 2016.
- [Spe16] M. Spertus, F. Vali, R. Smith: Template argument deduction for class templates (Rev. 4). WG21 P0091R1. 2016.
- [Str94] B. Stroustrup: The Design and Evolution of C++. Addison Wesley, ISBN 0-201-54330-3. 1994.

- [Str09] B. Stroustrup: The C++0x "Remove Concepts" Decision. Dr.Dobb's Journal. July 2009.
- [Str12] B. Stroustrup and A. Sutton (editors): A Concept Design for the STL. WG21 N3351. January 2012.
- [Sut11] A. Sutton and B. Stroustrup: Design of Concept Libraries for C++. Proc. SLE'11.
- [Sut15] A. Sutton: Introducing concepts. ACCU Overload 2015.
- [Sut16] A. Sutton: Defining concepts. ACCU Overload 2016.
- [Sut16a] A. Sutton: Overloading with Concepts. ACCU Overload, December 2016.
- [Vou16] V. Voutilainen and D. Vandevoorde: constexpr if. WG21 P0128R1. 2016.