

# Seminar Program Analysis and Transformation

## Concepts for C++

Mirko Stocker, me@misto.ch

May 28, 2009

*C++ templates are a powerful language construct that enable generic programming techniques without impacting runtime performance. A problem when using templates is the lack of an explicit contract between the user and the definition of a template. This leads to difficult to understand error messages and inhibits separate checking of template code.*

*Concepts are part of the next C++ standard and mitigate these problems. Concepts express explicit requirements to template argument types, thus allowing separate checking. Other benefits include syntactical adaption of existing types, making code more reusable.*

*This paper introduces concepts and shows how existing code can be refactored to use concepts, making the code easier to understand.*

### 1 Introduction

Concepts are a new feature of the C++ programming language and part of the working draft [Bec09] for the new standard<sup>1</sup>. The aim is to make the definition and the usage of a template verifiable for correctness separately from each other. This allows for a clearer separation between the implementation details of a template and its usage, preventing the often seen *leaking* of implementation details

<sup>1</sup>See the website of the C++ Standards Committee at <http://www.open-std.org/jtc1/sc22/wg21>.

through error messages to the user of a template. All of this can be achieved without affecting the runtime performance of the code, a necessary precondition if concepts are to become widely adapted and used in the standard library.

This paper is structured as follows. The remainder of this section features an introduction to the current situation in generic C++ programming, its problems and areas for improvement. Section 2 starting from page 3 contains a short repetition of templates terminology as well as a comparison to other languages. The main part of this paper is contained in Section 3 on page 5 where concepts are introduced and explained in detail with accompanying examples. Section 4 on page 14 shows how existing code can be refactored to use concepts as well as how concepts improve existing code.

#### 1.1 Current Situation

C++ templates have become useful building blocks for developers and are “the preferred implementation style for a vast array of reusable, efficient C++ libraries” [GJS<sup>+</sup>06]—for example the C++ standard template library [MSL94] and the libraries of the boost project [Com09]. While making code more universal to use, templates do not impact the performance of the code. In fact, the gener-

ated code is “optimal in both time and space” [GJS<sup>+</sup>06].

Templates do not introduce a level of indirection between the user and the implementation of the template one would inevitably have if virtual member functions and inheritance would be used to achieve polymorphism.

The code for templates is only generated if actually needed, during compilation. When generating code for template instantiations, the compiler has all the information from the caller and the callee available, allowing for various optimizations (e.g. inlining). Another benefit of not having to use interfaces is the enhanced flexibility—and more universal applicability—of templated code for both library writers and users of them.

This weak separation between templates and its usages is also a source of many troubles, as we shall see in the next section.

## 1.2 Problems With Templates

The lack of an *explicit* interface that forms a contract between the template and its user is problematic because it does not allow either side to be verified for correctness independently. Also, type-checking of templates happens late in the compilation process, when the definition and the usage of a template have already been combined [GJS<sup>+</sup>06]. The result can be observed—and is well known to every C++ programmer—by dreaded long-drawn-out error messages from the compiler. Implementation details leak through from libraries to the programmer, who should only be concerned with the library’s interface, not its internals. A well known example can be observed when trying to sort a list:

---

```
int main () {
    std::list<int> l;
    std::sort(l.begin(), l.end());
}
```

---

This results in 18 lines of error messages (from a recent GNU compiler), complaining

about the missing operator+ and operator- in, for example, `__i + 1` at line 1759 in `stl_algo.h`.

While a more senior developer is not scared by such messages and knows how to pick out the relevant details to fix the error, such complex error messages are often of no use to the novice C++ programmer and, as observed by the author during the education of—Java accustomed—students, can be a source of hostility towards the C++ language, which is unfortunate and a hindrance for more elaborate template use.

As previously noted, just working with abstract base classes and virtual member functions is not an acceptable solution, so we need another way of specifying the requirements a template definition has towards its parameters. This is what a new language feature called *concept* amends.

## 1.3 Introducing Concepts

Concepts are a means to formulate contracts between template definitions and their users. By naming or introducing a concept, a template author has to exactly specify all requirements template parameters have to satisfy: method signatures or other aspects—for example the availability of a copy constructor.

Concepts are already in use in the STL, although not as something that can be enforced and checked by the compiler, but simply as documentation in the form of naming conventions and terminology—like Bidirectional Iterator—telling the reader what to expect.

But there is more that can be achieved by introducing concepts as a language feature. Suppose you have a template that implements the functionality you want, but does not use the same function names or operations your classes expect. Traditionally, you would apply the adapter design pattern [GHJV95] to combine the two. *Concept maps* remove this burden and form a minimal solution by generating this adapter for you, of course also without introducing any runtime overhead.

Concepts are in work for the new C++ standard, expected to be ratified in late 2009 and published in late 2010 or early 2011 [WG208].

## 1.4 Refactoring

Refactoring is the technique to improve the structure, readability and maintainability of source code, or as stated by Martin Fowler [Fow99]: “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.”

We have already learned that the main benefit of concepts is improved checking of templates, resulting in better error messages for the user as well as making templates more robust and universal to use. Adding concepts to an existing code base is certainly a refactoring activity, resulting in more readable, expressive and better maintainable code.

Regarding concept maps, the removal of non-essential code without loss of functionality and the enhancement of the readability also speak in favor of refactoring.

Templates are a prerequisite for using concepts; an overview and recapitulation on templates follows in the next section.

## 2 Template Usage

This section gives a short overview on template notation and its different uses, as well as a comparison to other popular mainstream languages.

### 2.1 Generic Programming

Generic programming was introduced by Ada and designates the “decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces” [DS00].

In C++, generic programming is achieved via templates. So-called *template parameters* serve as a placeholder for an arbitrary type

(or concrete value of a type—for example an `int`—if the argument is not a type). For each type a template is parametrized with, it is instantiated once; the compiler virtually generates code from generic templates (at least in a mental model). A huge benefit of this approach is that we do not force any interfaces on the users of our templates, thus templates can work with arbitrary object hierarchies or primitive types as well, without the need to encapsulate (or “box”) them in an object as Java does. Or to put it in other words, templates enable compile time duck typing<sup>2</sup>.

But where are these *well-defined interfaces*? In templated C++, they are not described as such but implicit in the definition of the template.

So although C++ is statically typed, templates offer a lot of freedom in writing code that can work with various kinds of types. We will now take a closer look at templates and work out some terminology that is used in this paper.

### 2.2 Templates Revisited

A *template* can be a function or a class that is parametrized with one or multiple *template parameters*. The concrete types with which a template is used are called the *template arguments*. When the compiler encounters a template usage, it creates a *template instantiation* with the supplied or inferred template arguments (template type parameters can often be inferred from arguments using type inference [Str89]).

In the following listing, we can see the template function `max` whose template parameter is named `T`. The function is used with `int` as its template argument (note that the type `int` for `T` is inferred by the compiler).

---

<sup>2</sup>The term *duck typing* is often used in dynamically typed languages. If something *quacks like a duck* and *walks like a duck*, then treat it as a *duck*, without checking if it really is of the type *duck*.

---

```

template <typename T>
T max (const T& a, const T& b) {
    return a > b ? a : b;
}

int main () {
    int i = 1, j = 2, k;
    k = max(i, j);
}

```

---

The template applies the operator `>` on its parameters. In this case, operator `>` is a *dependent name* of the template; it is inferred from the use of the template parameters in the template function.

An *associated type* in a template is a type that is related to the template parameter type—for example an iterator’s `value_type`.

Class templates can also be *specialized* by providing an additional definition with a concrete type supplied for a template parameter (template functions are simply overloaded). This allows the programmer to specify a different implementation for the specified type, which is important to build highly efficient algorithms. An example from [GJS<sup>+</sup>06] is the STL `advance` algorithm, which moves an iterator a specified distance  $n$ . The general version simply increments the iterator  $n$  times, but if we are given an iterator that directly supports moving arbitrary distances—presumably in constant time—the compiler can dispatch to this specialized implementation.

### 2.3 Template Metaprogramming

As template parameters need not be types but can also be concrete values, together with template specialization, this enables template metaprogramming<sup>3</sup>.

Probably the most enlightening example is the compile time calculation of a factorial, as seen in the following listing.

---

```

template<int n>
struct factorial {
    static const int value =
        n * factorial<n-1>::value;
};

template<>
struct factorial<1> {
    static const int value = 1
};

int main() {
    int f = factorial<10>::value;
}

```

---

The compiler repeatedly instantiates the first template until the value has been reduced to 1, where the specialized version matches and the computation halts. Templates have not been designed to do metaprogramming, so the code can become quite confusing and not for the faint of heart.

## 2.4 Other Languages

Of course, C++ is not the only language supporting generic programming. In this section, we briefly relate C++ templates to Java and C# *generics* and explain how concepts relate to Haskell type classes.

### 2.4.1 Java

Java generics are based on the Pizza project’s Generic Java Compiler from Martin Odersky and Philip Wadler [BOSW98]. To be backwards compatible to older virtual machines and ungenerified Java, generics are erased (so-called *type erasure*) by the compiler and replaced with the `Object` type. All necessary type casts are inserted automatically at usage sites. This design allows to save memory in the virtual machine as there are no new classes present during runtime (single classes cannot be garbage collected in Java, only whole class loaders), but also, as Odersky recently [VS] said, to “a number of fairly ugly things”.

---

<sup>3</sup>In fact, the compiler’s template resolution process is Turing-complete, which was discovered by Erwin Unruh [Unr94]; a proof is sketched in [Vel03].

While this design allows backwards compatibility, it also makes generics much harder to understand and sometimes not very intuitive unless you know what the compiler is actually doing (see Angelika Langer’s Java Generics FAQ [Lan09] for more detailed information and examples on many pitfalls).

Generic’s parametrized types can be expressed in terms of other types to constrain the type with interfaces or a base class, but no finer grained constraints on signatures are possible. Java also does not offer any template metaprogramming facilities.

### 2.4.2 C#

C# generics, compared to C++ templates are, as C# lead architect Anders Hejlsberg says, “really just like classes, except they have a type parameter. C++ templates are really just like macros, except they look like classes. [...] C++ templates [without concepts] are actually untyped, or loosely typed. Whereas C# generics are strongly typed” [VE04]. C++ is a statically typed language, and templates do not change this; but the templates themselves are not type checked separately, only when instantiated.

The code for generics is created during runtime and only when actually needed. Generic methods share their native code.

C# constraints can be used to specify a base class, interfaces and a constraint on the constructor of the specified type. It is not possible to specify single operations, so in C#, constraints are based on types and not on signatures.

### 2.4.3 Haskell

A study on language support for generic programming [GJL<sup>+</sup>03] in 2003 came to the conclusion that Haskell has extraordinary support for generic programming, while C++ was placed only in the middle field. A more recent comparison [BJZ<sup>+</sup>08] between C++ with concepts and Haskell type classes shows that both languages are mostly equally

powerful with regards to generic programming.

Type classes are Haskell’s way of doing overloading of functions. For example, the Eq class is defined as the class having an == operator taking two arguments of the same type and returning a Bool<sup>4</sup>.

---

```
class Eq a where
  (==) :: a -> a -> Bool
```

---

To declare that a type is an instance of a type class, an instance declaration is used<sup>5</sup>:

---

```
instance Eq Integer where
  x == y = integerEq x y
```

---

As we shall see later, this is very similar to concepts and concept maps.

Probably the most fundamental difference between Haskell and C++ is that in Haskell, all types are inferred by the compiler. In a generic algorithm, Haskell deduces the constraints from the used entities, whereas in C++, the compiler limits the acceptable types according to the specified concepts.

## 3 Concepts

We saw in the previous section that C++ templates are much more powerful than their counterparts in other mainstream languages. But this greater flexibility requires more thought from the programmer, considering the knowledge (understanding the inner workings of libraries) and experience (reading and interpreting error messages) he needs to have.

What C++ needs is a means to make templates easier to write and to use, without the need to rely on external documentation the developer has to consult. This can be

---

<sup>4</sup>Can be read as: A type *a* is of class Eq if there exists an operator (==) whose type signature is defined by taking two arguments (the *as*) of the same type and returning an instance of Bool—either True or False.

<sup>5</sup>We say that Integer is of type Eq, and that the required operator (==) with the two arguments *x* and *y* is defined in terms of the primitive function integerEq.

achieved by introducing another element between the template definition and its users that formulates a contract both sides have to comply with: the template definition can only use the template parameters in the ways specified, and the template user can clearly see the requirements to his template arguments. Apart of these benefits, concepts also must not impair performance or compatibility with the existing C++ standard.

Actually, requirements could not only be of syntactic nature, but might also include semantics or even performance guarantees, but these obviously cannot be checked by a compiler.

The following introduction is loosely based on [DRS06] and [GJS<sup>+</sup>06].

To start, let us take a look at the following template function that sums up values—also known as *accumulate*—starting from a specific initial value, and then make the requirements to the parameters explicit in the form of concepts.

---

```
template <typename T, typename S>
S sum(T first, T last, S initial) {
    for(; first != last; ++first) {
        initial += *first;
    }
    return initial;
}
```

---

There are several prerequisites the writer of this function requires from the template arguments:

- T must be operator!= comparable with another T and return a value convertible to bool.
- T must have a pre-increment operator++.
- S must have a plus-assign operator+= that takes (or converts) T's value\_type<sup>6</sup>.
- S is returned from the function, so it must be copy-constructible.

---

<sup>6</sup>An iterator's value\_type is the type of the element an iterator points to.

Clearly, T is a so-called *input iterator*, and S is related to T's value\_type by the operator+=. With concepts, we can express these assumptions in code (to show the usage of associated types, I deliberately limited S to be T's value\_type):

---

```
template<typename Iter>

requires
    InputIterator<Iter>,
    PlusAssign<Iter::value_type>,
    HasCopyConstructor<Iter::value_type>

Iter::value_type sum(
    Iter first,
    Iter last,
    Iter::value_type initial) {

    for (; first != last; ++first)
        initial += *first;

    return initial;
}
```

---

We even got rid of one template parameter by explicitly involving the iterator's value\_type. We will shortly define all these concepts in the remainder of this section.

Using our sum function is straight forward, but what happens if you provide bogus arguments, for example another collection instead of the integer initial argument.

---

```
vector<int> i;
cout << sum(i.begin(), i.end(), list<int>());
```

---

GCC's g++ in the current version 4.3 spits out over twenty lines of error messages:

```
sum.cpp: In function 'int main()':
sum.cpp:28: error: no match for
'operator<<' in 'std::cout <<
sum(T, T, S) . . .

.. another 20 lines

sum.cpp:28: instantiated from here
sum.cpp:15: error: no match for
'operator+= ' in 'initial += first ...
```

The first error does not really help us much, but the last line gives us a clue that we

cannot += to our list initial. Compare this to the output from ConceptGCC (see Section 3.6 on page 13 for more information on ConceptGCC):

```
sum.cpp: In function 'int main()':
sum.cpp:49: error: cannot convert
'std::list<int, std::allocator<int> >'
to 'int' for argument '3' to ...
```

The reported line 49 of the error is the actual line where we called `sum` with the bogus argument, and the message clearly states that a `list<int>` is not convertible to `int`, the iterator's `value_type`. Contrast this to the first error message where the compiler told us a location from inside the template function; code we might not have written and have no intention to get too familiar with.

I believe these examples speak in strong favor of the usefulness of concepts.

### 3.1 Concept Syntax

According to Dos Reis and Stroustrup [DRS06], a concept is a triple  $\langle P, G, B \rangle$ , where  $P$  is a list of *parameters* to the concept; the same as in a template definition. Also as in templates, concept parameters can be compile-time values, such as an `int`, to express minimal semantic properties.

$G$  is the *guard*, a logical compile-time expression to add constraints to the concept parameters.  $G$  is optional.

$B$  denotes the *body* of the concept, a sequence of expressions and declarations that describe the usage of the template parameters.

Each concept also has a name, which is used to refer to the concept in a template definition or from other concepts.

$\langle P, G, B \rangle$  are combined to a concept using the following syntax<sup>7</sup>:

---

<sup>7</sup>Note that *requires* was formerly named *where*. This was changed in early 2007 by the ISO C++ standards committee.

---

```
auto8 concept ConceptName<P>
  requires G {
    B
  };
```

---

Returning to our example, we will now define the three concepts we used as follows:

---

```
auto concept HasCopyConstructor
  <typename T> {
  T::T(T const &);
}

auto concept PlusAssign
  <typename T, typename U = T> {
  typename result_type;
  result_type operator+=(T&, U const &);
}
```

---

We specify the return value of the operator to be the associated type `result_type`. We do not have to specify this type further as we do not use it in our code.

---

```
auto concept InputIterator
  <typename T> {
  typename value_type = T::value_type;
  bool operator!=(T const &, T const &);
  void operator++(T&);
  value_type operator*(T&);
}
```

---

Our version of an `InputIterator` is slightly more complex, it uses a `value_type` for the iterator and defaults it to the iterator's `value_type`. Although sufficient for our simple algorithm, this `InputIterator` is not very reusable. We could make it more universal by for example providing the `operator==`.

We shall now take a more detailed tour through the various features concepts offer.

#### 3.1.1 Associated Types

Associated types are other types the concept or template relies on, like the iterator's

---

<sup>8</sup>The `auto` keyword defines an implicit concept and is optional. This is explained later in section 3.2.1 and can be ignored for the moment.

value\_type we have already seen or a container's iterator. Associated types can be initialized with a default, often a nested type from the concept parameter, as in the excerpt of the container concept from the Concept-GCC [Cona] standard library header below.

---

```
concept Container<typename X> {
    //...
    ForwardIterator iterator =
        typename X::iterator;
}
```

---

The definition of an associated type can also be provided by a *concept map*, as we shall see later, or simply be left unspecified if unused, as we did with the result\_type.

### 3.1.2 Refinement

Refinement of concepts can be used to define a new concept in terms of existing ones. The typical example comes from iterators, where the forward iterator is a refinement of the input and output iterators (the forward iterator can in addition be default constructed). Bidirectional iterators in turn are a refinement of forward iterators (bidirectional iterators can be decremented).

Refinement of concepts uses the same syntax as C++ class inheritance. As with classes, concepts may refine multiple other concepts. For an example from the STL, InputIterator refines both IteratorBase and EqualityComparable.

---

```
concept InputIterator<typename X> :
    IteratorBase<X>,
    EqualityComparable<X>
{ ... }
```

---

### 3.1.3 Nested Requirements

Nested requirements express requirements on type parameters and on a concept's associated types, but in contrast to a requires clause belonging to *G*, nested requirements are requires clauses inside the body of the concept.

For example, we could extract a single requirement on a function into its own concept and require it from the originating concept:

---

```
auto concept HasPreIncrement
    <typename T> {
    void operator++(T&);
}

auto concept InputIterator
    <typename T> {
    requires HasPreIncrement<T>;
    typename value_type = T::value_type;
    bool operator!=(T const &, T const &);
    value_type operator*(T&);
}
```

---

### 3.1.4 Constraint Propagation

Constraint propagation is useful to minimize the effort of specifying concepts for templates. Constraint propagation means that the constraints of other templates used in a template definition are automatically added (propagated) to the template's requires clause. A special form of constraint propagation is the CopyConstructible concept, that is added to the requires clause whenever template parameters are passed to the template *by value*. You might have noticed that we did not explicitly state that the iterator from our example needs to be copy-constructible, this is thanks to constraint propagation.

### 3.1.5 Default Implementations

Finally, let's take a closer look at the ways we can specify function signatures. We have already seen function and method declarations, but it is also possible to directly specify a default implementation for operators and free functions, but *not* member functions. The EqualityComparable concept is defined as:

---

```

auto concept EqualityComparable
  <typename T, typename U = T> {

  bool operator==(T const & t,
    U const & u);
  bool operator!=(T const & t,
    U const & u) {
    return !(t == u);
  }
}

```

---

The limitation to free functions and operators is due to the lack of syntax to define member functions on primitive types, see the ConceptGCC FAQ [Conb] for more information.

### 3.1.6 Standard Concepts

The new standard header `<concepts>` defines concepts to check for syntactic and semantic properties [Bec09, Section 20.2]:

Concepts whose name is prefixed with `Has` provide detection of a specific syntax (e.g., `HasConstructor`), but do not imply the semantics of the corresponding operation. Concepts whose name has the `able` or `ible` suffix (e.g., `Constructible`) require both a specific syntax and semantics of the associated operations.

For example, there is a `LessThanComparable` concept that refines the `HasLess` concept with default implementations for all ordering operators:

---

```

auto concept HasLess<
  typename T, typename U> {
  bool operator<(T const &, U const &);
}

auto concept LessThanComparable<typename T>
  : HasLess<T, T> {

  bool operator>(T const & a, T const & b) {
    return b < a;
  }
}

```

```

bool operator<=(T const & a, T const & b) {
  return !(b < a);
}

bool operator>=(T const & a, T const & b) {
  return !(a < b);
}

```

---

We now know how to define concepts as well as templates that are constrained by concepts. What is still missing is a way to tell the compiler which type satisfies—also called “models”—a given concept. This is the responsibility of *concept maps*.

## 3.2 Concept Maps

As an example, the `EqualityComparable` concept we defined earlier is satisfied by the type `int` without any further work, we just have to tell this the compiler with the following concept map:

---

```

concept_map EqualityComparable<int> {}

```

---

What if the type that should model a concept does not provide all requirements of the concept? For example, if not all needed operations are available? Or you—for whatever reasons—want to have a different comparison to test for equality? In such a case, the concept map can hold definitions, and it can even override definitions from the type. The next listing shows a `Point` class, an `add` function that adds two variables, and an `Addable` concept that requires a `+` operator. The `Point` class does not have such an operator, so we have to define it in the concept map in order to use the `add` function.

---

```

struct Point {
  Point(int x, int y)
    : x(x), y(y) {}
  int x;
  int y;
};

auto concept Addable<typename T> {
  typename result_type;
}

```

```

    result_type operator+(T const &, T const &);
};

concept_map Addable<Point> {
    typedef Point result_type;

    result_type operator+(
        Point const & p1,
        Point const & p2) {
        return Point(p1.x + p2.x, p1.y + p2.y);
    }
};

template <Addable T>
requires
    CopyConstructible<T::result_type>

T::result_type add(
    T const & t1,
    T const & t2) {
    return t1 + t2;
};

```

What we can also see in this example is how the associated type `result_type` in the `Addable` concept is mapped to `Point` within the concept map. Note that concept maps cannot be used to provide member functions, only operators and free functions are allowed.

A concept map has to satisfy all requirements of its concept instance and it must not contain declarations that are not required by the concept.

### 3.2.1 Implicit and Explicit Concepts

Concept maps describe the relation between a concrete type and a concept, but it is also quite exhaustive for the user of a constrained template to explicitly define all mappings between the types he uses and the respective concepts. A concept prefixed by `auto` will declare it as *implicit*. This lets the compiler do the checking whether the type has all necessary requirements to satisfy a concept.

Coming back to our initial concept example, if we had not made use of implicit con-

cepts, we would have needed to add these three declarations to our example to use the `sum` function on a vector of integers.

---

```

concept_map PlusAssign<int> {};

concept_map MinimalInputIterator
    <vector<int>::iterator> {};

concept_map HasCopyConstructor<int> {};

```

---

Why are implicit concepts not the default? Implicit checks can obviously only be done on syntactic properties and not semantics. For example, take a template method that is overloaded to take either a `ForwardIterator` or an `InputIterator`. With the `ForwardIterator` being a refinement of the `InputIterator` [Str00], an iterator on an input stream will be dispatched to the more specialized `ForwardIterator` method, because it syntactically satisfies the `ForwardIterator` concept. This is incorrect as the `ForwardIterator` can by definition be used in multipass algorithms, but the input stream can only be consumed once.

## 3.3 Further Uses

Concept maps can also be used as adapters between different libraries to bridge syntactical differences. Concepts and concept maps provide “promising generic, non-intrusive, efficient, and identity preserving adapters” [JMS07].

Another clever use of concept maps can be seen in the new range-based `for` loop [Gre06] that enables the following Java-like syntax:

---

```

vector<int> vec = ...;
for( int i : vec )
    std::cout << i;

```

---

The type to loop over needs to satisfy the `For` concept that defines an iterator and two functions: `begin` and `end`. The concept map to arrays in the following listing allows to use the `for` loop on arbitrary arrays.

---

```

template<typename T, size_t N>
concept_map For<T[N]> {

    typedef T* iterator;

    T* begin(T (&array)[N]) {
        return array;
    }

    T* end (T (&array)[N]) {
        return array + N;
    }
}

```

---

This example also nicely shows how concept maps can be used in combination with traditional templates. To see more code and explanations, [Gre07] provides a tour through the wording of concepts with many examples.

### 3.3.1 Type Traits

Traits are a C++ template technique and were introduced by Nathan Myers [Mye95]. Traits are useful to provide a template parameter with additional information—for example, you are writing a template class that works with various floating point types. Those types have certain characteristics, like a maximum exponent or a so-called epsilon (the smallest difference that can be expressed by the type). So where do we take these properties from? A naive approach would be to specialize the class for each different floating point type, but that would lead to a lot of code duplication and would make our class hard to extend with other floating point types. So we extract just the template-type specific properties into their own template and specialize these *traits*.

A simple example, adapted from [Mye95], can be seen in the following listing.

---

```

template <class T>
struct float_traits { };

template <>
struct float_traits<float> {

```

```

    typedef float float_type;
    static inline float_type epsilon() {
        return FLT_EPSILON;
    }
};

```

```

template <>
struct float_traits<double> {
    typedef double float_type;
    static inline float_type epsilon() {
        return DBL_EPSILON;
    }
};

```

```

template <class T>
struct matrix {
    typedef float_traits<T> traits_type;
    inline T e() {
        return traits_type::epsilon();
    }
};

```

---

We can see that the implementation of the matrix class refers to the epsilon only via the trait, without having to make an additional distinction itself.

Adding supplementary functionality to a type is also something we can do with concepts and concept maps, the above example converted to concepts looks as follows.

---

```

concept FloatTraits<typename T> :
    std::CopyConstructible<T>{
    std::CopyConstructible float_type;
    float_type epsilon();
};

concept_map FloatTraits<double> {
    typedef double float_type;
    float_type epsilon() {
        return DBL_EPSILON;
    }
};

concept_map FloatTraits<float> {
    typedef float float_type;
    float_type epsilon() {
        return FLT_EPSILON;
    }
};

template <FloatTraits T>

```

```

struct matrix {
    T::float_type e() {
        return epsilon();
    }
};

```

The differences between the two versions are marginal; the usage of the associated `T::float_type` type makes the template class more pleasant to read.

A restriction of the concept version is that we can only define types and functions, whereas in the traditional version, the trait classes can also contain constants and enums.

### 3.4 Mental Model

Now that we know how concepts work and what features they offer, it is time to take a look behind the curtain on how concepts are represented in terms of plain old templates.

Imagine that the compiler converts concepts to class templates and concept maps to full specializations of these class templates. Function signatures are represented with static member functions that either contain the implementation from the concept map or simply forward the calls. Refinement is obviously mapped directly to inheritance.

To return once more to our sum example, let us take a look at the function when concepts are replaced with templates.

---

```

template <typename Iter>

Iter::value_type sum(
    Iter first,
    Iter last,
    Iter::value_type initial) {

    typedef InputIterator<Iter>::value_type
        value_type;

    for(
        InputIterator<Iter>::operator!=(first, last);
        InputIterator<Iter>::operator++(first)) {

        PlusAssign<value_type>::operator+=(
            initial,
            InputIterator<Iter>::operator*(first));

```

```

    }
    return HasCopyConstructor<value_type>::
        value_type(initial);
}

```

---

The code looks quite complicated, but it can help with understanding how concepts work when all the magic is replaced with familiar template code.

To conclude our introduction to concepts, we shall now take a look at another notation that was proposed for the standard.

### 3.5 Use Pattern Notation

An alternative proposal for adding concepts to C++ by Stroustrup and Dos Reis [DRS06] expresses concepts with a different syntax. “The major difference between the proposals occur in two places: how operations and refinements are specified within a concept and what kinds of requirements can occur within a where clause” [JGW<sup>+</sup>06].

Concepts are specified in a *use pattern* form, “based on the observation that a C++ expression [...] as it appears in a template is far more abstract, general and readable than the set of operations and auxiliary types needed to implement it” [DRS06]. So instead of specifying a list of operations with signatures—which is non-trivial and can result in a combinatorial explosion of possibilities as soon as conversions are involved—use-patterns allow a very intuitive notation. Rather than exactly specifying all operations used in the template, the notation allows to specify an example of how the template uses the parameter—for example in

---

```

*p++ = v;
bool b = (p != q);

```

---

where `p` and `q` are iterator variables, and `v` is a variable. This technique makes reading concepts very simple. To pick up our previous sum example, the `ForwardIterator` concept is defined as follows

---

```

concept ForwardIterator<typename Iter> {
    Var<Iter> p; // (1)
    typename Iter::value_type v; // (2)

    Iter q = p; // (3)

    bool b = (p != q); // (4)

    ++p; // (5)

    v = *p + v; // (6)
}

```

---

1. We introduce a variable of the type `Iter` named `p`, which we will use subsequently in the concept.
2. Type `Iter` needs to have an associated type, which we will refer to in the form of the variable `v`.
3. The `Iter` must be copyable to be passed as the parameter to `sum`.
4. The two iterators need to be `!=` comparable and result in a `bool` type.
5. It must support the pre-increment operator.
6. Finally, `Iter` must be dereferenceable; an operator`+` that takes the result of the dereferenced value and `v` and an assignment to `v`. The concept is also satisfied if there are implicit conversions necessary to use the operators. Luckily for us, we do not have to specify them explicitly.

This notation is quite different from the signature based approach—which is probably more familiar for most programmers—but was not elected for the standard.

Now that we have already seen several concepts, let us take a look at a compiler for concepts.

## 3.6 ConceptGCC

“ConceptGCC is a derivative of the GNU C++ compiler” and implements the signature based notation of concepts [Cona].

C++ compilers use the inclusion model [GS04] to compile templates (several other models would be possible [Vel00]). In the inclusion model, the compiler instantiates a template with concrete types whenever needed, leading to distinct code for different types that can be optimized per type (in contrast to Java and C#). This is also the reason why the full template definition needs to be available when compiling code that uses the template.

[GS05] contains more information on the implementation details of ConceptGCC.

### 3.6.1 Error Messages

Every C++ developer knows that compiler error messages could be more helpful in diagnosing errors. With concepts, the compiler can provide much more specific error messages. If the programmer uses types that do not conform to a certain concept, the compiler notices this when checking against the concept and not until after instantiating the template. Coming back to our introductory example from section 1.2 on page 2—the application of `sort` to a list iterator. ConceptGCC’s error message shown in Figure 1 on the following page does not need any further explanation why the code fails to compile.

ConceptGCC can even tell you the concrete code to write when a concept map is missing for an explicit concept: “note: if the concept semantics are met, write a concept map: `concept_map CopyConstructible<Point> ;`”.

Now that we have seen the possibilities of concepts, let us take a look at how we can bring their benefits into our projects.

```

sort.cpp:6: error: no matching function for call to 'sort(std::_List_iterator<int>,
std::_List_iterator<int>)'
.. note: candidates are: void std::sort(_Iter, _Iter) [with _Iter = std::_List_iterator<int>]
<requirements>
.. note: no concept map for requirement 'std::RandomAccessIterator<std::_List_iterator<int> >'

```

Figure 1: ConceptGCC’s error message when trying to use sort with list iterators.

## 4 Refactoring

Summarizing the previous section, concepts improve the support for generic programming in C++ by enabling type checking of template definitions rather than their usages. This allows for more understandable and specific error messages. While concepts bring all these benefits, they also do not decrease the performance or limit the flexibility.

Refactoring is changing software “in such a way that it does not alter the external behavior of the code yet improves its internal structure” [Fow99]. We have seen that concepts are not directly influencing any kind of external behavior but are mostly an aid to the developer and user of templates. Existing code can be upgraded with concepts, making the code easier to understand and maintain.

Concepts are just source code as well, they can be renamed, moved, and parts can be extracted. Refinements form a class hierarchy, extract sub- and superconcept are possible. So there are a number of obvious refactorings that can be applied to concept code.

Besides that, existing template code becomes a candidate for refactoring towards concepts, and this is on what we shall concentrate in the remainder of this paper.

Obviously, to introduce concepts, the code already needs to make use of templates.

### 4.1 From Templates to Concepts

To see some practical applications of concepts, we take the C++ unit testing easier

(CUTE<sup>9</sup>) framework<sup>10</sup> and write concepts for it. CUTE is open source and rather small, and it has a lot of templates in it, which makes it a perfect candidate for our endeavor.

CUTE uses templates for various purposes: test listeners to count failures, type dependent selection of comparison operations, contexts that are instantiated for a test run and of course various functors to call the actual tests.

From the moment a template is constrained with a concept, all dependent code must be adapted before you will be able to successfully compile again. So it is not possible—at least with ConceptGCC—to make small gradual changes and to always have running code while refactoring.

Nevertheless, we can separate the introduction of concepts into several steps. Given a template class or function, perform the following steps.

1. Identify a usage of a template parameter. Look out for the following occurrences of the template parameter: is it used as the super class or as a member variable? Is it default or copy-constructed? Is it passed to other functions?
2. Create a new concept to describe the usage, or take a look at the `<concepts>` header to find an existing concept that describes the usage.
3. Add the respective `requires` clause to the template.

<sup>9</sup><http://r2.ifs.hsr.ch/cute>

<sup>10</sup>I took the liberty to make the code better suited for reading on paper by changing names and omitting parts.

4. Repeat from step 1 until done.
5. Compile the code to find missing concepts or, if non-auto concepts are used, which concept maps need to be declared. In the case of missing concepts, the compiler will complain about missing operations or constructors that are unavailable in the now (over-) constrained template.
6. When the code is in a working state again, review the concept. Is there more than one requirement on a template type? It might make the code more readable if these requirements are extracted to form a new concept with its own problem-specific name. Is there an operator involved? Giving it its own concept name might make your intention clearer.
7. Try to replace multiple concepts with just a single one. For example, the `std::Predicate` refines `std::Callable` and requires the result type to be convertible to `bool`.
8. Look at other templates in your codebase. If the same concept (in the traditional meaning) is used in several templates, consider to give it a domain specific name.

We shall now take a look at some examples.

#### 4.1.1 Listener Concept

When running a CUTE test suite, a runner can be parametrized with a Listener which gets notified when something interesting occurs (test started, failed, ended successfully, etc.). The listener is defined as follows:

---

```
template <typename Listener=null_listener>

struct runner : Listener{
    runner():Listener(){}
    runner(Listener &s):Listener(s){}
    ...
}
```

---

The default type is `null_listener`, a Null Object type [Woo97] that has empty implementations of all necessary methods. Other listeners include the `counting_listener` and several IDE listeners.

Thanks to the `null_listener`, we already have a concise enumeration of the methods a type has to provide to model the Listener concept:

---

```
struct null_listener{
    void begin(suite const &s, char const *info){}
    void end(suite const &s, char const *info){}
    void start(test const &t){}
    void success(test const &t,char const *msg){}
    void failure(test const &t,test_failure const &e){}
    void error(test const &t,char const *what){}
};
```

---

In the excerpt from the runner class, we also see that two constructors are required. We can reuse the definitions from the `<concepts>` header to specify the constructors. The resulting concept looks as follows:

---

```
auto concept ListenerConcept <typename T> :
    std::DefaultConstructible<T>,
    std::CopyConstructible<T> {

    void T::begin(suite const &, char const *);
    void T::end(suite const &, char const *);
    void T::start(test const &);
    void T::success(test const &, char const *);
    void T::failure(test const &,test_failure const &);
    void T::error(test const &,char const *);
};
```

---

We declared it as an auto concept, so adding concept maps is not necessary. But to enable the checking of the concept, we have to adapt all the templates that use a listener:

---

```
template <typename Listener=null_listener>

requires
    ListenerConcept<Listener>

{...}
```

---

#### 4.1.2 Incarnate Concept

Tests in CUTE can be functors, and if needed, they can be provided with a context. At the

time the test is run, the functor is instantiated and the context is passed to the constructor. This allows the constructor and destructor to be run as part of the test execution.

---

```

template <
    typename TestFun,
    typename ContextObj>

struct with_context {

    with_context(ContextObj ctx) : ctx(ctx) {}

    void operator() {
        TestFun t(ctx);
        t();
    }

    ContextObj ctx;
};

template <
    typename TestFun,
    typename ContextObj>

test make_with_context(ContextObj obj) {

    return test(
        with_context<TestFun,ContextObj>(obj));
}

```

---

Starting with the struct `with_context`, we can tell that the `TestFun` type needs to have an `operator()` without parameters, which is expressed in the `std::Callable0` concept.

The `ContextObj` type is assigned to a member variable in the constructor and therefore needs to model the `std::CopyConstructible<T>` concept we are already familiar with.

Furthermore, the `TestFun` constructor takes an `ContextObj` instance. This specific relationship between the types can be expressed with the `std::HasConstructor1<TestFun, ContextObj>` concept.

Regarding the `make_with_context` function, it returns a test which copies its argument internally. This means that `with_context` needs to be `std::CopyConstructible` as well.

Finally, the code with all concepts looks as follows.

---

```

template <
    typename TestFun,
    typename ContextObj >

    requires
        std::Callable0<TestFun>,
        std::CopyConstructible<ContextObj>,
        std::HasConstructor1<TestFun,ContextObj>

struct with_context {

    with_context(ContextObj ctx) : ctx(ctx) {}

    void operator() {
        TestFun t(ctx);
        t();
    }

    ContextObj ctx;
};

template <
    typename TestFun,
    typename ContextObj>

    requires
        std::Callable0<TestFun>,
        std::CopyConstructible<ContextObj>,
        std::HasConstructor1<TestFun, ContextObj>,
        std::CopyConstructible<
            cute::with_context<TestFun,ContextObj>>

test make_with_context(ContextObj obj) {

    return test(
        with_context<TestFun,ContextObj>(obj));
}

```

---

In this example, there was no need to introduce new concepts—we were able to reuse those provided by the standard library. It might make sense though to introduce new concepts and give them their own name, so they can be reused in other parts of the system. For example, we could have reformulated `std::HasConstructor1<TestFun, ContextObj>` as `FunctorWithContext<TestFun, ContextObj>`.

What have we actually gained by these changes? Let us compare the compiler errors from an invalid usage of the above template in two cases: when the user passes a type with an inappropriate operator() and a type that does not have a required constructor.

Without concepts, the error messages are much longer and the clue comes at the end, after many lines of uninteresting internals and look like this:

```
error: no matching function for call to
'NotAFunctor::NotAFunctor(int&)'
note: candidates are:
    NotAFunctor::NotAFunctor()
    NotAFunctor::NotAFunctor(const NotAFunctor&)
error: no match for call to '(NotAFunctor) ()'
```

With concepts, the errors are basically reduced to the following two lines:

```
note: no concept map for requirement
'cute::std::HasConstructor1<NotAFunctor, int>'
note: no concept map for requirement
'cute::std::Callable0<NotAFunctor>'
```

## 4.2 Experiences

Introducing concepts for a template is an all-or-nothing activity; it is not possible to constrain only some aspects of a type and leave others open for the moment. This makes perfectly sense from the compiler's point of view, but for a developer experimenting with concepts and refactoring his code, this makes it much harder and can be quite frustrating at times.

CUTE uses pointers to member functions in some templates, which seems to be unsupported [Lis09] by concepts. Wrapping the pointer in a `std::mem_fun` before passing it as a template argument circumvented the problem in this specific case.

## 4.3 Adapters with Concept Maps

As we have seen in the introduction to concept maps (Section 3.2 on page 9), concepts can be used to adapt existing types, at

least when only operators and free functions are involved—member functions cannot be adapted.

## 4.4 Tool Support

Once concepts are supported by all major compilers, there is still a lot of legacy code that needs to be converted to make concepts useful. Can parts of this task be automated? This is certainly the case, and I would like to give some ideas on what needs to be done in this section.

### 4.4.1 Concept Generation

Given parsing support for the new keywords and a representation of concepts in the abstract syntax tree, a refactoring tool performs similar steps as the programmer: identifying uses and transforming them into the concept notation. I could imagine the following procedure:

1. Identify a usage of the template parameter.
2. Find the signature of the function that is used with the parameter; special cases are constructors.
3. Repeat from step 1 until done.
4. Construct the according concepts and add the necessary requirement. Skip the concept construction if the usage is already described by an existing concept. Try to reuse existing concepts.

This very simple procedure provides only a coarse overview, it would also be necessary to handle associated types and the propagation of constraints. The task does not necessarily need to be fully automated, a kind of wizard that guides the user might be more appropriate.

When implicit concepts are generated, it is not needed to specify concept maps.

#### 4.4.2 IDE Support

Programmers expect their IDE to automate as many mundane tasks as possible; nowadays this also includes refactoring.

The Eclipse based CDT provides refactoring support for C++[GZS07], but not for refactoring towards concepts. Besides the generation of concepts as discussed above, an IDE can also assist the developer with other automation, for example providing simplifications for user written concepts by built-in concepts.

Another idea might be that the IDE can generate the code necessary to model a certain concept. For example, if you want to use an algorithm, the IDE could with the help of the algorithm's concepts generate a skeleton for the templates that are needed, or alternatively tell you what properties a template parameter type lacks to be used with the algorithm.

## 5 Outlook

We have seen that concepts bring many benefits to both template writers and users, with separate checking and more appealing error messages. These benefits come at the cost of an increased complexity that can—for template users—be eased with implicit concepts.

Concepts are included in the next C++ standard, which does not have a fixed release date yet but is expected for late 2010 or early 2011 [WG208].

ConceptGCC is a proof of concept implementation and is now being merged back into the GNU compiler collection.

What about the other C++ compilers? The Edison Design Group that writes the front-end which many commercial compilers are using is implementing concepts and expects to be done in late 2011 [EDG09].

It will certainly take a few more years until concepts are commonplace in projects, but nevertheless, I believe they are a very useful as well as necessary invention.

## References

- [Bec09] Pete Becker. Working draft, standard for programming language c++. Technical Report N2857=09-0047, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, March 2009.
- [BJZ<sup>+</sup>08] Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, Sibylle Schupp, and Andreas Priesnitz. A comparison of c++ concepts and haskell type classes. In *WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic programming*, pages 37–48, New York, NY, USA, 2008. ACM.
- [BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proc. OPPLA'98*, October 1998.
- [Com09] Boost Community. Boost c++ libraries. <http://www.boost.org>, last visited March 2009.
- [Cona] <http://www.generic-programming.org/software/ConceptGCC>. Concept GCC.
- [Conb] <http://www.generic-programming.org/faq/?category=conceptcxx>. Concept GCC FAQ.
- [DRS06] Gabriel Dos Reis and Bjarne Stroustrup. Specifying c++ concepts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 2006. ACM.
- [DS00] James C. Dehnert and Alexander A. Stepanov. Fundamentals of generic programming. In *Selected Papers from the International Seminar on Generic Programming*, pages 1–11, London, UK, 2000. Springer-Verlag.
- [EDG09] Private communication with Steve Adamczyk from Edison Design Group, 2009.

- [Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [GJL<sup>+</sup>03] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 115–134, New York, NY, USA, 2003. ACM.
- [GJS<sup>+</sup>06] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in c++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310, New York, NY, USA, 2006. ACM.
- [Gre06] Douglas Gregor. Conceptualizing the range-based for loop. Technical Report N2049=06-0119, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.
- [Gre07] Douglas Gregor. A tour of the concepts wording. Technical Report N2399=07-0259, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, September 2007.
- [GS04] Brian J. Gough and Richard M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004.
- [GS05] Douglas Gregor and Jeremy Siek. Implementing concepts. Technical Report N1848=05-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2005.
- [GZS07] Emanuel Graf, Guido Zraggen, and Peter Sommerlad. Refactoring support for the c++ development tooling. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 781–782, New York, NY, USA, 2007. ACM.
- [JGW<sup>+</sup>06] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy Siek. Algorithm specialization in generic programming: challenges of constrained generics in c++. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 272–282, New York, NY, USA, 2006. ACM.
- [JMS07] Jaakko Järvi, Matthew A. Marcus, and Jacob N. Smith. Library composition and adaptation using c++ concepts. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 73–82, New York, NY, USA, 2007. ACM.
- [Lan09] Angelika Langer. Java generics faq. <http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>, last visited March 2009.
- [Lis09] ConceptGCC Mailing List. Concepts and member functions. 28.4.2009, 2009.
- [MSL94] Kapur Musser, , Alexander Stepanov, and Meng Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1994.
- [Mye95] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.
- [Str89] Bjarne Stroustrup. Parameterized types for c++. *J. Object Oriented Program.*, 1(5):5–16, 1989.
- [Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

- [Unr94] Erwin Unruh. Prime number computation, 1994.
- [VE04] Bill Venners and Bruce Eckel. Generics in `c#`, `java`, and `c++`. <http://www.artima.com/intv/generics2.html>, January 2004.
- [Vel00] Todd L. Veldhuizen. Five compilation models for C++ templates. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.
- [Vel03] Todd L. Veldhuizen. C++ templates are turing complete, 2003.
- [VS] Bill Venners and Frank Sommers. The origins of `scala`. [http://www.artima.com/scalazine/articles/origins\\_of\\_scala.html](http://www.artima.com/scalazine/articles/origins_of_scala.html).
- [WG208] Minutes of wg21 meeting. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2697.html>, 2008.
- [Woo97] Bobby Woolf. Null object. pages 5–18, 1997.