N1536=03-0119
October 22, 2003

Concepts – syntax and composition

Bjarne Stroustrup (bs@research.att.com)

Gabriel Dos Reis (gdr@acm.com)

Abstract

Assume that we have a basic notion of a **concept**. A concept is a specification of the properties a type must have to meet a template definition's requirement on a template type parameter. In other words, concepts provide a type system for types. What syntax would be suitable for concepts in C++? What rules for composition of concepts are needed to express general requirements elegantly and to express common requirement simply? How do concepts integrate with other aspects of templates and generic programming techniques? This note answers such questions providing a perfect integration of concepts into the C++ syntax and type system, providing parameterized concepts, concept inheritance, overloading of templates based on concepts, and a simplified notation for template definitions and use. For ease of implementation, this system maps directly onto existing mechanisms.

Basic concepts

Fundamentally, a concept is the type of a type. How to express simple concepts is left to a companion paper [Stroustrup,2003b]. Here, we will simply assume that basic concepts can be expressed:

```
concept Value_type {
        // can be copied
};

concept Arithmetic {
        // can be added, subtracted, multiplied, and divided
};
```

For example **int** and **complex<float>** match both **Value_type** and **Arithmetic**, whereas **int*** and **std::string** match **Value_type** but not **Arithmetic**.

Template declaration syntax

Given concepts, how can we use them in template declarations?

Basic syntax

Consider a traditional template:

```
template<class T> void f(T);
```

Here, "**class T**" is read "for all types T". If **f()** takes only arithmetic types, we need a specification "for all arithmetic types T" or "for all types T, such that T is arithmetic". That is:

```
template<Arithmetic T> void f(T);
```

We can then write

```
f(1);                       // ok: int is Arithmetic
f(complex<int>(1,2));       // ok: complex is Arithmetic
f("asdf");                  // error: char[] is not Arithmetic
```

That's the basic use and the syntax is obvious and fits smoothly with the traditional **<class T>** or **<typename T>** notation. How template arguments are checked for conformance to the requirements on their corresponding parameters is discussed in [Stroustrup,2003b]. As ever, each template parameter can name a distinct type. For example:

```
template<Random_access_iterator Ran>
        void sort(Ran, Ran);
template<Forrward_iterator For1, Forward_iterator For2>
        For1 merge(For1,For1,For2,For2);
```

Here, **sort()** takes a pair of **Random_access_iterator**s**,** each of the same type, whereas **merge()** takes two pairs of **Forward_iterator**s where each pair must be of the same type and each pair can be of a different type from the other. That's exactly as specified by the standard for **sort()** and **merge()**. The difference is that here the rules are expressed directly in the language, so that a compiler can use that information and a human reader has a better chance to understand the code. In other words, requirements that are not understood by C++98 compilers can now be expressed and systematically checked.

Of course, **class** can be used as ever with its meaning unchanged; that is, to indicate that a template argument must be a type, and not for example an **int** value. For example:

```
template<class T> void f(T);          // T can be any type
template<Forward_iterator For, class T> For find(For, For, const T&);
```

Clearly, this syntax does not introduce any new ambiguities.

A simplified syntax

The C++ template syntax is often criticized for being "clumsy" or verbose. The counter-opinion is that it's simple, clear, and unambiguous. With the introduction of concepts, it becomes possible to eliminate the prefix **template<** … **>** in many cases where it is not necessary to name the template argument type. We simply use a concept rather than a type to introduce an argument. For example:

```
void f(Arithmetic a);     // equivalent to template<Arithmetic A> void f(A a);
```

That is, a concept used in place of a function argument type implicitly declares a function template with a template argument of a type that matches that concept. The old syntax is still needed where a return type is a template argument (because the return type appears lexically before function arguments) or when two template arguments must be of the same type, and for class templates. For example:

```
template<Arithmetic A> A f1(A a);
template<Arithmetic A> A f2(int);
void f3(Arithmetic a);
```

For **f1()** we need to name the **Arithmetic** type **A** so that we can use it twice, for **f2()** we need to name the type to use it as a return type, whereas for **f3()** we can do without naming it.

```
template<Random_iterator RA> void g1(RA p, RA q);
void g2(Random_iterator p1, Random_iterator p2);
```

For **g1(),** we name the **Random_iterator** type and use it twice. That implies that **p** and **q** are of the same type. For **g2(),** we did not name the **Random_iterator** types and (consequently) didn't state that the type of **p1** and **p2** had to be the same. Therefore, **p1** and **p2** can be of different types.

Finally consider class templates. For example:

```
template<Value_type V> class vector { /* … */ };
```

In analogy to the function template case, we could consider a simplification of this case:

```
class vector<Value_type> { /* … */ };
```

This looks like a specialization. If it was accepted, we'd have to except this usage from the rule that we can't specialize without a general template. Furthermore, in most class templates, we'd like to refer to a template argument type within the template. For example:

```
template<Value_type V> class vector {
        V* elements;
        // …
```

```
};
```

For the "simplified" syntax, we don't have a name to use, so we'd need yet-another new rule to allow us to refer to the argument type. Thus, that "simplification" is best avoided.

In function templates, we can use **decltype** as proposed in [Järvi,2003] to refer to an otherwise unnamed argument type. For example:

```
void f(Arithmetic a)  // equivalent to template<Arithmetic A> void f(A a)
{
        decltype(a) b;          // b is of the same type as the variable a
        auto c = a;             // c is of the same type as the value a
        // …
}
```

Relation to the decltype/auto proposal

With the simplified syntax, what would be the equivalent to the traditional **template<class T>**? That would be **auto** arguments as suggested in [Järvi,2003]. For example:

```
void f(auto a);         // equivalent to template<class T> void f(T a)
```

That is, **auto** in place of a function parameter type implicitly declares a template with a type template parameter that matches any type template argument. Looking at it that way, it becomes obvious how the syntax could be simplified for many return types:

```
auto f(auto a);           // equivalent to template<class T, class U> T f(U a)
Iterator f(Iterator p); // equivalent to template<Iterator T, Iterator U> T g(U)
```

This is nice for consistency, but probably not particularly useful because there typically is a relationship between the argument type and the return type. However, the other half of the auto proposal, **decltype()**, could be used here. For example:

```
auto f(auto a) -> decltype(a);          // equivalent to template<class T> T f(T)
```

That is, we can mention a template argument type without actually naming it.

So, **auto** simply denotes the most general (the least constrained) **concept**. It follows that any concept might be used in place of **auto**. For example:

```
Iterator p = f(x);      // p gets the type of the value returned by f(x)
                        // provided that value is of an Iterator type

Iterator f(Iterator a) -> decltype(a); // f returns an Iterator of the type it accepts
```

To complete the generalization based on the observation that **auto** is the name for the most general **concept**, it could be allowed instead of class to introduce an arbitrary template type argument:

```
template<auto T> T f(T);     // equivalent to template<class T> T f(T)
```

It is a separate question whether the complexity of having two ways of saying things outweighs the advantage of being able to say simple things simply. My inclination is to accept both styles.

It may be possible to extend the meaning of **auto** so as to accept non-type arguments. Gabriel Dos Reis is considering the feasibility and desirability of distinguishing **auto** from **class** in this way. For now, we will assume that **auto** matches any type but not any non-type.

Composing concepts

Concepts model types. Consequently, most (possibly all) of the mechanisms we use for constructing types from other types are useful for constructing concepts out of other concepts.

Parameterized concepts

Parameterized (templatized) concepts are essential for describing types that are constructed from templates. For example:

```
template<Value_type T> concept Forward_iterator {
        // iteration operations for a sequence of Ts
};

template<Value_type T> concept Container {
        // iteration over a sequence of Value_types
};
```

These concepts might be used like this:

```
template<Value_type T> class vector {
        / / …
};

template<Container<int> C>  C::iterator find(C& c, int val)
{
        return find(c.begin(),c.end(), val);
}

vector<int> vi;
int a[10];
```

```
// …
vector<int>::iterator p = find(vi,7);          // ok: vector<int> is a container
int* q = find(a,7);                            // error: an array is not a Container
```

Note that **typename** is not needed to prefix **C::iterator** because **Container** must be in scope and is known to name a type.
The function template **find()** can equivalently be expressed using the terser "simplified syntax":

```
auto  find(Container<int>& c, int val)
{
        return find(c.begin(),c.end(), val);
}

vector<int> vi;
// …
auto p = find(vi,7);
```

How do we generalize this to any container of any value type? It is most obviously expressed using the traditional template syntax

```
template<Value_type T, Container<T> C, Value_type U>
C::iterator find(C& c, U val)
{
        return find(c.begin(),c.end(),val);
}

vector<int> vi;
list<int> lsti;
vector<double> vd;
// …
vector<int>::iterator p1 = find(vi,7);
vector<int>::iterator p2 = find(vi,7.7);
list<int>::iterator p3 = find(lsti,7);
vector<double>::iterator p4 = find(vd,7);
```

This can obviously be reduced to

```
template<Value_type T> auto find(Container<T>& c, Value_type val)
{
        return find(c.begin(),c.end(),val);
}

vector<int> vi;
list<int> lsti;
vector<double> vd;
```

```
// …
auto p1= find(vi,7);
auto p2 = find(vi,7.7);
auto p3 = find(lsti,7);
auto p4 = find(vd,7);
```

To further reduce this example, we need to extend the rule that a concept can stand for a type that matches it in a function argument declaration to the use of a concepts in a qualification of a concept:

```
auto find(Container<Value_type>& c, Value_type val)
{
        return find(c.begin(),c.end(),val);
}

vector<int> vi;
list<int> lsti;
vector<double> vd;
// …
auto p1= find(vi,7);
auto p2 = find(vi,7.7);
auto p3 = find(lsti,7);
auto p4 = find(vd,7);
```

How would we declare **find()** without also defining it?

```
template<Value_type T, Container<T> C, Value_type V)
        C::iterator find (C&c, V val);
```

or

```
auto find(Container<Value_type>& c,Value_type val)->decltype(c)::iterator;
```

Neither declaration is particularly pretty.

Multi-argument constraints

Some template code makes sense only if several template arguments are suitably related. For example:

```
template<class T, class U> void f(T t, U u) { g(t,u); }
template<class A, class B, class C> void h(A a, B b, C c) { a = b+c; }
```

Obviously**, f()** must require that a function **g()** exist for a give pair of template arguments **T** and **U**. Similarly, **h()** must require that for a given set of template arguments { **A**,**B**,**C** } the sum of an **B** and a **C** must be assignable to an **A**.

We might want to place constraints on the arguments to **f()** for logical reasons. However, the only constraints we need to place on **f()**'s arguments for implementation reasons is that they can be passed along to **g()**. How might we express that constraint? The constraints are easy to express. For example:

```
template<Value_type X, Container Y> concept G {
        constraints(X a, Y y) { g(x,y); }
};
```

However, there is no elegant way of applying that concept to **f()**. By twisting our logic a bit, we can express the constraint involving two arguments as a constraint on a third argument:

```
template<class T, class U, G<T,U> Unused> void f(T t, U u) { g(t,u); }
```

That **Unused** parameter will always be deduced from the first two parameters. That is, it isn't really a parameter; it is not used to specify an argument. All it does is to constrain the other two arguments: If **G<T,U>** can be formed for specific types **T** and **U**, a specialization of f**()** will be valid; otherwise that specialization will fail.

Apparently, this suffices to specify any constraints involving multiple template arguments, so the question is whether the notation is acceptable. After looking at a few examples, my opinion is that that idiom is important enough to deserve to be supported by syntax and odd enough to be condemned as "yet another obscure hack" if it isn't. Therefore, we can use a "**where** clause" to express multi-parameter constraints:

```
template<class T, class U> where G<T,U> void f(T t, U u) { g(t,u); }
```

The semantics is as specified above.

Derived concepts

Consider the standard library iterators. Their requirements form a hierarchy. It follows that a good model of iterators also form a hierarchy. For example, consider just **Forward_iterator** and **Randon_access_iterator** (ignoring the rest for simplicity):

```
concept Forward_iterator<Value_type> {
        // forward iterator facilities
};

concept Random_access_iterator<Value_type T> : Forward_iterator<T> {
        // facilities provided by a random access iterator
        // over and above what a forward iterator provides
};
```

In other words, a **Random_access_iterator** is a **Forward_iterator**, so that any type that is a **Random_access_iterator** is also a **Forward_iterator**. We see no point in defining

visibility of concept bases. Obviously, as with classes, parameterization and derivation can be used in combination for concepts.

We see no reason to complicate the notion of concept with private or protected inheritance of concepts.

Composition operators

Consider a couple and useful simple concepts:

```
concept Comparable {
        // can be compared using <. <=, ==, and !=
};

concept Value_type {
        // can be copied
};
```

How can we require that a type must be both Comparable and a Value_type? Multiple inheritance provides an obvious solution:

```
concept Comparable_value : Comparable, Value_type { };

void f(Comparable_value cv);
```

However, that requires the introduction of a new named concept. That may be inconvenient. Instead, we might express the idea directly:

```
template<Comparrable && Value_type T> void f(T cv);
```

or even more directly:

```
void f(Comparable && Value_type cv);
```

Concepts can be combined using the three logical operators **&&** (and), || (or) and **!** (not). Assuming that **Pointer** and **Reference** are concepts, this is another example

```
void g(Comparable && !Reference);
void h(Pointer || Reference);          // access indirectly
```

Here is a more realistic example:

```
template<RandomIterator<ValueType && Comparable> Iter>
void sort(Iter first, Iter last);

template<Container<ValueType && Comparable> C>
void sort(C& c)
{
```

```
        sort(c.begin(),c.end());
}

vector<int>vi;
vector<complex<float>> vc;
int ai[10];
// …
sort(vi);
sort(vc);       // error: vc is not a Container of Comparables
sort(ai);       // error: ai is not a Container
```

This last function could also be defined using the simplified syntax

```
void sort(Container<ValueType && Comparable & c);
```

Template overloading

Given concepts, we can obviously select templates based on the template argument types. If all arguments match their corresponding concepts, an instantiation is possible, otherwise not. For example:

```
template<Pointer P> class List;      // List of pointers
template<Comparable> class List; // List of types with compare operations
template<Value_type> class List;   // List of types with copy operations

List<int*> lst1;                 // List of pointers
List<int> lst2;                  // ???
List< complex<int> > lst3;   // List of value types (complex has no <)
List< int& > lst4;               // error: no match

class Blob {
        // <, <=, …
private:
        Blob& operator=(const Blob&);     // prevent copy
        // …
};

List<Blob> lst5;        // List of Comparable
```

How about **lst2**? For any reasonable definition an **int** is both a **Comparable** and a **Value_type**. If neither **Comparable** nor **Value_type** are derived from each other, we have an ambiguity and the declaration of **lst2** is an error. However, we could resolve the problem by providing a hierarchy of types and apply the usual overload resolution rules and choose the most specialized implementation:

```
concept Value_type { /* … */ };
```

```
concept Comparable : Value_type { /* … */ };

List<int> lst2;          // List of Comparable
```

Naturally, this requires foresight in the definition of the concepts. If that isn't feasible or convenient, foresight can be applied to the definitions of List. For example:

```
template<Comparable && Value_type> class List;                  // #1
template<Value_type && !Comparable > class List;
template<Comparable && !Value_type> class List;

List<int> lst2;          // List #1
```

This is messy. If having to choose among valid definitions of a template is shown to be common and important, we must provide a mechanism for expressing a preference order among templates or a way of explicitly specifying a template at a point of use. For example;

```
preference List<Comparable>, List<Value_type>;
```

A template mentioned before another in a preference-list will be preferred.

Defined concept match

When I define a type, I might like to say that it is intended to match a know concept. For example:

```
concept C { /* … */ };
class X { /* … */ };    // X matches C
```

Unfortunately, that comment is only a comment. A compiler can't verify that pious wish, and despite my best efforts, I might have made a mistake in the definition of **X**. However, I can easily check:

```
void is_C(C) {}
void X_checker(X x) {  is_C(x); }
```

Now, **X_checker()** will compile only if **X** is a **C**.

This can be seen as an elegant technique, or as a sleazy hack. If the latter view prevails, we could introduce special syntax to say "**X** matches **Y**". The obvious syntax is the base class syntax:

```
class X : C { /* … */ };
```

After all, "**:**" or "**: public**" are often pronounced "is a" or "is a kind of". If a type needs to be checked against a concept in a place separate from it's definition, the **is_C()** technique seems perfectly adequate.

It is essential that there be no language requirement or widespread "style rule" to the effect that a class should be defined with its set of concepts.

References

[Järvi,2003] Jaakko Järvi and Bjarne Stroustrup: Mechanisms for querying types of expressions – Decltype and auto revisited. N1527=03-0110.

[Stroustrup,2003a] B. Stroustrup and G. Dos Reis: *Design criteria for concepts*.

[Stroustrup,2003b] B. Stroustrup: *Concept checking – a more abstract complement to type checking*.