N1522=03-0105
October 22, 2003

Concepts –

Design choices for template argument checking

Bjarne Stroustrup (bs@research.att.com)

Gabriel Dos Reis (gdr@acm.org)

Abstract

This note presents problems, ideals, and design ideas for ways of checking template arguments. The aim is to simplify the writing and use of templates without loss of run-time performance or expressive power compared to the original template design while maintaining complete backwards compatibility with the original template design.

A specification of the requirements on a set of template arguments is called a **concept**. Because specifying and using such requirements are central to thinking about templates, several apparently independent suggestions for language extensions, such as a uniform function call syntax, are also briefly presented.

The problems

Templates provide unprecedented flexibility to C++ without loss of performance. They add to compile-time type safety and precise specification compared to most popular alternatives. They are key to most parts of the C++ standard library and the basis for most novel techniques for using C++ over the last ten years or so.

However, they also present the programmer with significant problems: The expectations of a template on its arguments cannot be explicitly stated in C++98 and will in general be checked late – at template instantiation time. The result is often error messages of spectacular length and obscurity. There are techniques to minimize this effect (called constraints checking or concept checking [Stoustrup,2001] [Austern,2002] [BOOST,200?]), but they are not systematic and cannot handle all kinds of errors. The fundamental problem is that crucial aspects of the programmer's understanding of a template cannot be expressed directly in the code for compilers and other tools to use.

Furthermore, there is no effective way of selecting a template based on the kind of arguments supplied. At most, the number of arguments can be used to select among different templates. This leads to the introduction of otherwise redundant names and to overly clever programming techniques.

To provide optimal performance, templates provide little separation between the template definition, the context of the template definition, the template arguments, and the context of the template arguments. Guided by complicated rules, information is collected – potentially from

several translation units – to generate code. Naturally, this leads to compiler complexity and occasionally to confusion.

No additional language facility can address the problem of compiler complexity. However, the proposed design directly addresses the other problems. It may also minimize the amount to which code will depend on the more obscure template rules.

A way of looking at the problem

The construct **template<class T>** was introduced as C++'s version of the classical math "for all T". As most programmers know, "for all T" is often/typically qualified with a clause such as "such that T is ..." stating the properties required of a T. Despite looking, Stroustrup failed to find an acceptable way of saying that in the original C++ design [Stroustrup, 1994]. Consequently, what we are looking for here is a way of stating "such that T is ...". What we want for C++ is a way to easily specify required properties without over-constraining arguments, without limiting what we can express, and without adding overhead. We also want to use knowledge of such properties to improve error handling, to simplify the expression of ideas in program text, and potentially to help compilers generate efficient code.

Another way of saying this is that we need a type system for template arguments so that a template writer can clearly state what is expected and assumed from a template argument. That is, we must be able to classify types (and values) and use such classifications, called **concepts**, to specify what kinds of template arguments are required by a template. From our perspective, **concepts** act like predicates on template arguments. Once formalized, **concepts** can be composed and manipulated much like types themselves.

Design aims

This section describes the aims of the concept design as they have emerged through analysis of the problems with existing templates, comparison with generic facilities in other programming languages (e.g. [Garcia,2003]), discussions with programmers, and – a most importantly – consideration of how generic programming techniques could be better supported by C++. Not all of these aims can be simultaneously achieved. They are ideals that must be approximated, and occasionally we have to choose between those ideals. The order of aims expresses a rough priority order. When conflicts between aims occur we tend to resolve them in favor of the higher ranked aim.

This section provides a summary of the design aims. Consequently, each aim is accompanied by only the briefest rationale or elaboration.

1. **A system as flexible as current templates**. Templates and dynamically typed languages have demonstrated far greater flexibility and ability to sustain reuse than systems based on types and interfaces. The primary reason is that specifying the type of arguments for operations requires exact agreement between template writer and template user – that's more foresight that we can realistically expect. Consequently, constraints should not require explicit declaration of argument types and be non-hierarchical. In particular, built-in types, such as **int** and **Shape*** should remain first-class template arguments, requiring no workarounds.
2. **Enable better checking of template definitions**. The ideal is complete checking of a template definition in isolation from its actual arguments.

3. **Enable better checking of template uses**. The ideal is complete checking of a template use in isolation from the template definition.
4. **Better error messages**. The quality of error messages of current compilers are much better than what have been seen in the past – especially if constraints templates [Stroustrup,2001] are used by the programmer. However, such error messages are often produced too late (at template instantiation time) in deep nesting instantiation levels and still leave much to be desired. Three kinds of errors must ideally be caught and succinctly reported:
    a. the use of an unspecified operation in a template within a template definition (without access to the template arguments)
    b. the lack of a required operation by a template argument type (at the point of template use without access to the template definition)
    c. an invalid use of an operation or combination of operations within a template definition (this may require access to actual template arguments).

    Note that it requires increasing amounts of information to detect those three categories of errors.
5. **Selection of template specialization based on attributes of template arguments**. It must be possible to define several templates (e.g. template functions or template classes) and to select the one to be used based on the actual template arguments. Furthermore, we must be able to specify preferences among related types and among separately developed types (**class** is the minimally constraining concept).
6. **Typical code performs equivalent to existing template code**. It is easy to improve checking of templates at the cost of flexibility and run-time performance – just define a template argument to be some kind of abstract class and you have perfect separation of concerns and conventional type checking. However, you can already get that from conventional object-oriented code using abstract classes. The constraints/concept facility must sustain and enhance the compile-time evaluation and inlining that are the foundations of the performance of classical template code.
7. **Simple to implement in current compilers**. This should hold true for compile-time, link-time, and code-generation concerns. Templates have become terribly difficult to implement well. A constraint/concept facility should not make that problem significantly worse. Furthermore, it should ease the checking of most uses.
8. **Compatibility with current syntax and semantics**. Ideally, every existing piece of template code will still work as before. Furthermore, it should do so when mixed with code using concepts. The syntax used to express concepts should fit smoothly with the existing template syntax. We cannot require existing types to be systematically rewritten to be useful as arguments to templates using concepts.
9. **Separate compilation of template and template use**. This obvious ideal requires use of some sort of **vtbl** (container of pointers to functions) to implement the interface from a template to a template argument.
10. **Simple/terse expression of constrains**: The most general statement of the constraints on template arguments is that the template specialization compiles. That's what type checking does for otherwise unconstrained template arguments. To be useful and comprehensible in the absence of the template body, a constraint must succinctly specify a logical property.
11. **Express constraints in terms of other constraints**. Often, a template requires arguments that must meet a logical combination of concepts. Thus, a programmer must be able to

name constraints and express new constraints as combinations of existing ones. This could include "negative constraints", such as "not a reference".

12. **Constraints of combinations of template arguments**. Often, several template arguments are used in combination. In such cases, the ability to combine operations on arguments can be more important than the exact attributes of individual arguments. It should be possible to express requirements on the relationships among template arguments.
13. **Express semantics/invariants of concept models**: For example, a user should be able to state that his/her array class has the property that its elements occupy contiguous storage, or that an increment followed by a decrement of his/her bidirectional iterator is a no-op.
14. **The extensions shouldn't hinder other language improvements**. In particular, constraints/concepts should not preclude other suggested improvements in support of generic programming, such as template aliases, templated name spaces, and namespace template arguments.

Companion papers [Stroustrup,2003a] and [Stroustrup,2003b] explore alternatives for specifying **concept**s and ways of composing **concept**s, respectively.

The basic ideas for "concepts"

Within the aims/ideal for the design lurk two hard-to-combine fundamental notions:

- **that of types**, which provides for clean separation between implementation and use of an object (in this case a template argument) and efficient code generation;
- **that of constraints**, which provides flexibility in the matching of template arguments and their use in a template.

A **concept** defines a set of properties, sometimes called **constraints** that a type must posses to be used as a template argument. Both the notions of types and constraints have a long history in C++ and elsewhere. We leave the discussion of these alternatives to a companion paper [Stroustrup,2003a]. Here, we will simply assume a method for specifying simple requirements of a type template parameter on the corresponding argument. The expressive power of concepts is sufficient to express the standards library requirements as stated in the ISO Standard [Stroustrup,2003c]. A type either matches a concept, or it doesn't. This section discusses how such a facility can be used.

```
concept Random_accessor {
        // a type is a Random_accessor if it allows proper use of [], *, ++, --, +, and –
        // how this is specified and how a type matches a concept is discussed elsewhere
};
```

That is, a type is a **Random_accessor** if it has what it takes to provide random access.

Consider the standard library requirements.

```
concept Value_type {
        // require copy constructor and copy assignment
};
```

```
template<Value_type V> concept ForwardIterator {
        // requite forward iterator behavior for a sequence of Vs
};
```

A **ForwardIterator** is parameterized by its **Value_type**. Such parameterization will be a major way of composing concepts out of other concepts.

```
template<Value_type V, FowardIterator<Value_type> FI>
FI find (FI first, FI last, const V& value);

int a[7];
// ...
int* p = find(a,a+7, 101);
```

How can we build **RandomAccessIterator** from a **ForwardIterator**? (Ignoring other iterators for this discussion). The obvious answer appears to be concept inheritance;

```
template<Value_type V>
concept RandomAccessIterator : ForwardIterator<V> {
        // require -, --, and []
};
```

Now we can overload a template function based on its template argument concepts:

```
template<class T> void poke(T);
template<ForwardIterator<Value_type> T> void poke(T);
template<RandomAccessIterator<Value_type> T> void poke(T);

poke(2);        // call the general poke
int* p;
poke(p);        // call poke(RandomAccessIterator<int>)
class I {
public:
        I& operator++(), I operator++(int); int& operator*();
        // no []
        // …
}};
I i;
poke(i);        // call poke(ForwardIterator<int>)
```

Here, we have assumed that when two alternatives are callable, we pick the most specialized. The issue of overload resolution is discussed in [Stroustrup,2003b].

Improving notational support for generic programming

This section makes suggestions about ways of improving notational uniformity to ease generic programming. The main purpose of these suggestions is to demonstrate that some limitations of the variants of the concept notion discussed are not fundamental. The discussion here is **not** proposals for language change. However, some variants of the concept notion imply the need for some such extensions. Our favorite version of concepts – the usage-pattern approach – would require none of these notational supports, but would benefit from the uniform calling syntax.

Member types

The lack of member types for built-in types is a serious problem for generic programs. It stops programmers from simply unifying code for built-in and user defined types and leads to the introduction of traits classes.

We could address that problem by associating a fixed list of attributes to user-defined types. For example, we could simply decide that every pointer type has a member type ("pseudo member type", maybe) called "**value_type**". For example:

```
int*::value_type       // int
```

If the syntax is considered problematic (for aesthetic or parsing reasons) we could simply allow that **::value_type** only for ordinary type names. For example:

```
int*::value_type;      // error

typedef int* P;
P::value_type;         // ok: int

template<class T> void f(T t) { T::value_type v; /* ...  */ };
int i = 7;
f(&i);  // here T will be int* and T::value_type will be int
```

This restriction would not have any practical problems because we only need to use **value_type** when we don't have the exact type handy.

Traits classes like **std::iterator_traits** are workarounds for the lack of such member types for pointer types. Consequently, the obvious set of member types for a built-in pointer type would include the members of **std::iterator_traits**:

```
iterator_category
value_type
difference_type
pointer
reference
```

However, improvements to that list are possible. Notice that most of those members are duplicated in **std::iterator**.

One generalization of this idea is a general mechanism for attaching attributes to a type. For example:

```
int* has_member typedef int value_type;
```

or

```
template<class T>  operator:: (T*, value_type)  = T;      // "::" as binary op. on names.
```

However, that would be a major change with implications that are hard to fathom. Traditionally, we don't add attributes to a type after its initial definition.

Anyway, **type_traits** work very well for many uses. Such traits could be directly supported by the compiler to provide a more general set of compile time attributes for types.

Operator functions

To use an operator, such as +, we just mention it. For example:

```
a+b
```

When we need to refer to it, say in a definition, we name it. For example:

```
complex operator+(complex a, complex b) { return a+=b; }
```

However, we can't name built-in operators. This is a problem for every proposal that relies on type-based specification and also tries to cope directly with built-in types and operations. Consider

```
template<BinaryFct call>
    call::result_type f(call p) { return p(); }
```

How can we call that for an integer add? There is no direct way:

```
f(+);             // syntax error
f(operator+)  // error: which operator+
```

We could use a workaround:

```
struct Int_plus {
        typedef int result_type;
        result_type operator()(int i, int j) { return i+j; }
};

f(Int_plus());
```

Had it not been for the (sneaky, but realistic) use of **result_type** we could have used a free standing function:

```
int int_add(int a, int b) { return i+j; }

f(int_add);
```

or more generally

```
template<class L, class R, class Res>
   Res operator+(L a, R b) { return a+b; }

f(operator+<int,int,int>);


template<class L, class R, class Res>
   struct Plus {
   public:
         typedef Res result_type;
         Res operator()(L a, R b) { return a+b; }
   };

f(Plus<int,int,int>());
```

Before we accept any scheme that requires users to name functions, we must consider standardizing a library of such templates or making some such names built-in. Note that this was anticipated in the original template proposal [Stroustrup,1988], but deemed "not yet necessary; more experience is needed".

Uniform call syntax

There are two basic forms of calling syntax in C++: member function calls and "ordinary" function calls (i.e. calls of free-standing functions). For example:

```
class X {
public:
         void f(int);
};

void f(X,int);

void g(X x)
{
         f(1,2);  // ok
         1.f(2);  // error
         x.f(1);  // ok
         f(x,1);  // error
}
```

This complicates specification of concepts, complicates generic programming (as does every non-uniformity of syntax), and forces the programmer to remember how a function was defined.

Note that this problem doesn't exist for operators. For example **a+b** means that either + resolves to a built-in operator or it means **a.operator+(b)** or it means **operator+(a,b)**. The compiler can and does choose among these alternatives. In principle, it could do so for the example above also. For example:

```
void h(X x)
{
        f(1,2);  // ok
        1.f(2);  // #1 ok: means f(1,2)
        x.f(1);  // ok
        f(x,1);  // #2 ok: means x.f(1)
}
```

For #1 to work, there would need to be a rule that said that a free-standing function could be called as a member function with its first argument presented as an object before the dot.

For #2 to work, there would need to be a rule that said that a member function could be called as a non-member function with an extra initial argument. Alternatively, one might consider a general "rewriting system" whereby some patterns may be rewritten based on constraints/concepts.

There is a host of details to specify for before this idea is a proposal, but the fact that its equivalent exists for operators shows that these details can be worked out, and roughly how.

In the context of **concept**s, the main importance of uniform call syntax would be that

- requirements (concepts) could be expressed without distinguishing between member functions and free-standing functions
- an operation missing for an argument class could be added without modifying the argument class

For example, consider this example of a usage-based concept [Stroustrup,2003b]:

```
concept Comp {
        constraints(Comp a, Comp b)
                { a==b; a!=b; a==1; a!=1; before(a,b); after(a,b); }
};

template<Comp C> class T1 {
        void f(C c1, C c2)              // check use in f() against concept
        {
                c1.before(c2); // fine
                after(c1,c2);   // fine
                c1.after(c2);   // fine
                before(c1,c2); // fine: before() defined for a Comp
                somefct(c1);   // error: somefct() not defined for a Comp
        }
};
```

```
class X {
public:
        bool before(X);
        bool after(X);
        // ==, !=, …
};

class Y {
public:
        // ==, !=, …
};

bool before(const Y&, const Y&);
bool after(const Y&, const Y&);

void use(X a, X b, Y x, Y y)
{
        f(a,b); // check type X against concept Comp
        f(x,y); // check type X against concept Comp
}
```

Interestingly, this relaxation of notation will give the appearance of non-member virtual functions.

Naturally, we could decide to allow member functions to be called using the non-member syntax while not allowing non-member functions to be called using the member syntax (or - less obviously useful - vise versa). Our preference is to support the traditional non-member-call syntax as a uniform syntax (i.e. to allow the non-member-call syntax for member functions, but not to support the member-class syntax for non-members).

Run-time semantics and invariants

The expressive power of generic constructs can be enhanced and even more efficient code may be generated, or more run-time checking can be performed, if compilers have knowledge of properties/attributes of a type. Such information can be provided in the type's **concept**s. For example, if **Op** is an **EquivalenceRelation,** we have

Op(x, x) == true
If Op(x, y) then Op(y, x)
If Op(x, y) and Op(y, z) then Op(x, z)

It is desirable to find notation to express such invariants; a notation resembling the following was suggested in [Dos Reis, ACCU2002]:

{ Op(x, x) } == { true}
{ Op(x, y) == true } -> { Op(y, x) == true }
{ Op(x, y) == true && Op(y, z) == true } -> { Op(x, z) == true }

Of course, the syntax is not ideal, but the core idea is there. Similarly, if the operators == and **!=** model an **EqualityOperator** on a given, then it is desirable to express the following invariants

{ x != y } == { !(x == y) }

Also, if **T** is an **Value_type**, we would to express that that initialization with a given value is semantically equivalent to default initialization followed by assignment.

{ T x = y } == { T x; x = y }

If **p** is of type that complies to **BidirectionalIterator**, we would like to express the invariant

{ ++p; ---p } == { p }

Such run-time invariants may be used by a compiler either as basis for optimization or as basis for run-time checking.

Could concepts be classes?

Do we need **concept** as a language feature? Could we use existing language features to support out ideals well enough to make a language feature a luxury? For example:

```
template<class T> class Value_type {        // T can be copied
        static void constraints(T a) { T a = b; a=b; }
public:
        Value_type() { void (*p)(T) = constraints; }
};

template<class T> class Z : private Value_type<T> {      // check constraints
        // …
};

template<class T> void f(T t)
{
        Value_type<T>();     // check constraints
        // …
}
```

The definition of **Value_type** is considered "odd" by many and the application of the constraint in a function definition is unfortunate. Like type information, constraint checks belong in declarations, not just in definitions, and should be declarative.

Worse, unless we start changing language rules, there is no way of ensuring that a template doesn't use operations that are not specified among its constraints. For example, a template may be said to accept a forward iterator, yet use a + to advance it. Such code will work for many iterators (incl. all pointers into arrays), but that + will "lurk" in the code waiting to be a bug when someone

actually instantiate a template with an argument that is only a forward iterator. Such problems decrease our confidence in template code.

To compensate for such problems would involve major extensions. In essence, the result would be a **concept** mechanism.

Implementation complexity

Any proposal for checking of template arguments will affect some of the key areas of the language and involves modifications to known difficult parts of a compiler. We try to minimize implementation complexity by keeping the rules for concepts as close to already existing rules as possible – that often means identical to. Often, it is useful to think of a concept as an oddly used class. Internally to a compiler, a concept would most likely best be represented and handled as a kind of class.

Historical note

The problems caused by template arguments not being explicitly typed and not checked in isolation from the template body were appreciated at the time of the initial template design. "The Design and Evolution of C++" [Stroustrup,1994] devotes four pages (sec 14.5) to the problem and to various ways to "constrain" template arguments. The use of "constraints functions" triggered by constructors was used in the earliest template code. They are a direct ancestor of today's constraint and concept classes and of the usage-based concept idea proposed in [Stroustrup,2003a].

Most proposals to add some checking to template arguments – in C++ and related languages – have focused on the idea of expressing a template argument as a base class. This leads to requirement that each type that are to be used as a template argument must be derived from some class known to (and often defined by) the template writer. This view has repeatedly been rejected as being too rigid and requiring too much foresight from writers of both templates and template arguments. Additionally, it makes it very tempting (but not necessary) to define operations on template arguments as virtual functions – implying either a run-time overhead for simple uses (such as a simple template container) or compiler magic for examples of such uses deemed especially important.

The C++ templates are fundamentally more abstract and flexible than type-based approaches (with its inspiration in math rather than in type theory).

Acknowledgements

Alex Stepanov provided the impetus to this work on concepts as a language feature through years of work with generic programming and constant pressure to provide the cleanest, most fundamental, and most general facilities for supporting it.

References

[Austern,2002] M. Austren: ???.
[BOOST,200?] ???: ???.
[Dos Reis,2002] Dos Reis: *Generic Programming in C++: The next level.* ACCU2002

[Garcia,2003] Garcia, et al: *A comparative study of language support for generic programming*. Proc. OOPSLA 2003.
[Stroustrup,1988] B. Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference. October, 1988.
[Stroustrup,1994] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994.
[Stroustrup,2001] B. Stroustrup: *Why can't I define constraints for my template parameters?* http://www.research.att.com/~bs/bs_faq2.html#constraints
[Stroustrup,2003a] B. Stroustrup: *Concept Checking: A more abstract complement to type checking*.
[Stroustrup,2003b] B. Stroustrup: *Concepts – syntax and composition*.
[Stroustrup,2003c]: B. Stroustrup: *Expressing the standard library requirements as concepts*. To be written.