N1510=03-0093
October 22, 2003

Concept checking –
A more abstract complement to type checking

Bjarne Stroustrup (bs@cs.tamu.edu)

Abstract

This paper discusses the problem of how to express a generic construct's requirements on its parameters. In the context of C++ templates, it proposes a notion of "concept checking" based on explicitly declared usage patterns. This notion is more abstract, more flexible, and easier to express than conventional type checking based on function signatures. The proposed notion of **concept** provides not just precise specification of template argument requirements and good compile-time detection of errors, but also supports the equivalent of overloading for templates while maintaining C++ templates' support for compile time evaluation and inlining.

This paper compares the usage-pattern approach to conventional function-signature approaches to generic parameter specification: unlike a signature-based approach, the usage-pattern approach does not require perfect foresight from a programmer or perfect agreement between collaborating developers. Concepts provide a complement to types, rather than an alternative. Concepts represent abstract requirements more directly than types. The advantages of concepts are not limited to C++; they are fundamental and would apply to many languages providing basic support for generic programming techniques.

Introduction

Conventional static (compile-time) type checking performs two roles:

- it controls the way a programmer can invoke and combine operations (e.g., it verifies that you can add two integers and you can add two complex numbers)
- it provides sufficient information for code to be generated (e.g., it ensures that a compiler has the information it needs to generate the exact code sequences to add two integers and to add two complex numbers)

The need to provide the information required for code generation when writing code overconstrains the programmer's expression of solutions. For example, to express a simple call, **f(a,b)**, the programmer must state the exact types of **a** and **b**, find the definition of **f**, and see that the parameters of **f** can accept **a** and **b**. Overloading and generic mechanisms, such as C macros and C++ templates, allow the programmer to simply state **f(a,b)** leaving the resolution of the call to the compiler (much as a dynamically checked language leaves the resolution of a call to an interpreter). Postponing type checking like this dramatically simplifies the expression of ideas, but

leaves a serious problem: How does a generic function express its expectations of an argument?

The most common conventional answer to this question is that a template argument must be of a type that's compatible with a type specified in the template. For example:

```
template<class T : Base> void f(T);          // pseudo code
```

Here, **Base** is a class and any argument to **f** must be of a class derived from **Base**. For example:

```
class X { /* … */ };    // not derived from Base
class Y : public Base { /* … */ };

f(X()); // error: an X is not a Base
f(Y()); // ok: a Y is a Base
f(2);    // error: an int is not a Base
```

Schemes like this have been repeatedly considered for C++ (and rejected) since about 1986 and is used in languages such as Eiffel, Generic Java, and Generic C# [Garcia,2003]. The key is that any template argument must somehow match the type required by the template. That is, the types of the operations on a template argument must match the types (signatures) of the operations on the type specified in the template parameter declaration. The definition of "match" can vary, but in essence all signature-based schemes require sufficient information for the interface between a generic definition and its arguments to be the logical equivalent to a set of function pointers with their argument types, return types, and any other information needed to call them (such as pass-by-value or pass-by-reference) known at compile-time.

The following sections discuss

- problems and advantages from unconstrained template arguments (as in Standard C++)
- problems with conventional signature-based approaches to constraints checking
- an alternative approach based on usage patterns
- how usage-pattern-based **concept**s can address the problems with signature-based approaches
- how usage-pattern-based **concepts** can form a flexible and coherent system for expressing and using requirements on template parameters.

The various approaches are primarily evaluated in terms of flexibility and generality of what they can express. However, performance of code written using them is also considered very important and the complexity of implementing the approaches is taken into account.

Unconstrained template arguments

Templates have been a great success in C++ as measured by the amount of code using them, the range of concepts that can be expressed using them, the efficiency of the code

generated from them, the range of innovative techniques based on them, and the number of immitators. However, the **template<class T>** is simply a variant of math's "For all T" that sacrifice precise statement of requirements on **T** for flexibility. Naturally, all template specialization code is eventually statically checked using the usual type rules, so all use of templates is type safe. However, the lack of specification of a template definition's requirements on its template arguments leads to hard to understand code, elaborate documentation conventions, workarounds, late detection of errors, and spectacularly poor error messages.

In the absence of language support, C++ users have resorted to writing "requirement" that cannot be checked directly by compilers and to express argument constraints as templates. The former approach is a crucial part of the ISO C++ standard itself as it defines the properties of arguments to standard library templates as tables of required operators and semantics of such operations. That approach is abstract, terse, comprehensible, reasonably precise, but cannot be expressed directly in C++ itself. Nor is it easy to check these constraints for consistency. The aim of any template argument constraint mechanism for C++ must be to express these standard library constraints directly. A companion paper [Stroustrup,2003c] does exactly that.

In the absence of language support, programmers have managed to express many constraints in the language itself [Stroustrup,2001] [Austern,2002] [BOOST,200?]. For example:

```
template<class T> struct  Addable {          // Ts can be added
        static void constraints(T a, T b) { a+b; }
        Value_type() { void (*p)(T) = constraints; }
};

template<class T> class My_type
        : private Addable<T> {       // any T must be Addable
        // …
};

template<class T> void my_fct(T a)
{
        Addable<T>();         // any T must be Addable
        // …
}

My_type<int> m1;              // ok: ints can be added
My_type<char*> m2;            // error: pointers cannot be added

void f(int i, char* p)
{
        f(i);     // ok: ints can be added
        f(p);     // error: pointers cannot be added
}
```

This technique gives rather good error messages and allows the programmer to express a wide range of constraints. Unfortunately, it is not perfect. For example, a constraints class cannot catch the use of an unexpected function in a template function. For example:

```
template<class Value_type> struct Forward_iterator {
        static void constraints(Forward_iterator p)
        {
                Forward_iterator q = p; p = q;      // can be copied
                p++; ++p;                           // can be incremented
                Value_type v = *p;                  // points to Value_types
        }
        Forward_iterator() { void(*p)(Forward_iterator) = constraints; }
};

template<class Iter> void my_fct(Iter p, Iter q)
{
        Forward_iterator<iterator_traits<I>::value_type>();
        // …
        p = p+2;        // oops, uncaught: + not in Forward_iterator
        // …
}

void fct(int v[], int s, istream& is)
{
        my_fct(v,v+s);          // will compile because int* has + defined
        // …
        istream_iterator<int> ii(is);
        istream_iterator<int> eos;
        my_fct(ii,eos);         // error: istream::iterator doesn't have + defined
        // …
}
```

The inability to catch such fairly common errors in template definitions leaves problems to be found late in the compilation process, sometimes years after the original definition of the templates.

Despite its limitations, the constraints template class technique deserves a much wider use than it currently has because it does address a major problem in the C++ type system. The main deficiencies of the approach is that

- constraints are part of the definition of a template rather than part of its declaration or type.
- the form of the constraint templates is perceived as odd enough to deter many programmers and is vulnerable to spurious compiler warnings (some variants of the idea are so elaborate that they have to be used via macros)
- it is not possible to select among templates based on properties of a template argument type

- constraints template classes cannot catch the use of an unexpected operation on a template argument
- there is no common style of constraints classes
- there is no basic set of constraints templates in the standard library covering common template argument constraints

Variants of this approach was used from the very earliest days of C++ and are documented in D&E [Stroustrup,1994]. One variant has become popular as part of the BOOST library [BOOST,200?].

Using constraints template classes does not address the most fundamental problem with the unconstrained template arguments in C++. Constraints cannot provide separation between the definition of a template and its use, leading to serious comprehension and implementation complexities. No mechanism within the current language could address that problem. This problem can only be addressed by a new language facility.

That said, unconstrained template arguments provide many fundamental advantages, which are the basis of the success of C++ templates. Templates

- give uniform treatment to built-in and user-defined types
- are neutral in respect to calling conventions (e.g. **a+b** can be resolved as a built-in + operator, a member function call, or as a call of a free-standing function)
- enable excellent inlining, so that templates can be used for basic containers and high-performance numeric types
- require only minimal foresight from writers of argument types (e.g. a writer of a new class does not have to specify which interfaces the class can meet before using it as a template argument)

These are major advantages that must not be lost in an attempt to improve the separation between template definitions and their use, to improve type checking of templates, and to improve the range of uses of templates. Given those constraints on a solution, the ideal is

- to completely verify the correctness of a template body in the absence of actual template arguments
- to completely verify the correctness of a template use in the absence of the template body

The base-class approach

An approach relying on specifying template argument constraints as base classes is fundamentally simple to understand (partly because it's familiar) and simple to implement. Each template argument is constrained to be a class derived from a specified base class so that a template definition is "syntactic sugar" for the use of objects of classes implementing the interface specified by the "constraining base class". No fundamentally new semantic notions have been introduced. For example:

```
struct Element { // defines interface for elements of containers that can be sorted
        virtual bool less_than(Element&) =0;
        virtual void swap(Element&) =0;
};

template<class T : Element> void sort(Container<T>& c) // pseudo code
```

```
{
        // …
        if (c[i].less_than(c[j])) c[i].swap(c[j]);
        // …
}

class Number : public Element {
        int n;
public:
        bool less_than(Element& e)  { return n<((Number&)e).n; }
        void swap(Element&e )
        {
                int tmp = ((Number&)e).n;  // extract value from argument
                ((Number&)e).n=n;          // store new value in argument
                n = tmp;
        }
        // …
};

Container<Number> nc;
// …
sort(nc);
```

Note the casts needed in the **Number** class. A language could make them implicit in the same way as a language could eliminate the need for an explicit & to denote a reference. However, a conversion is needed somewhere and somehow to get from the base class interface to the derived class types.

This explicitly parameterized **sort()** is (assuming some suitable definition of **Container**) equivalent to an unparameterized function:

```
void sort(ElementContainer& c)
{
        // …
        if (c[i].less_than(c[j])) c[i].swap(c[j]);
        // …
}
```

In a language relying on a universal base class "**Object**", this would become:

```
void sort(Container& c)
{
        // …
        Element& ci = (Element&)c[i];      // run-time checked conversion
        Element& cj = (Element&)c[j];      // run-time checked conversion
        if (ci.less_than(cj)) ci.swap(cj);
        // …
```

```
}
```

This simple mapping of a generic function (relying on parameterization) to its object-oriented equivalent (relying on virtual functions in a class hierarchy) demonstrates how this kind of constraints on arguments to generic constructs provide a simple solution to one of the most fundamental problems of C++ templates: This use of constraints cleanly separates the template definition from the definition of its template arguments, allowing for separate compilation of the two. Unfortunately, this useful separation comes with a high cost both in terms of logical properties and performance.

Consider first the lesser problem: performance. The obvious implementation – relying on a vector of functions – imposes a virtual function call overhead on every operation on a template argument. For simple operations, such as subscripting on arrays or addition of integers, that cost becomes prohibitive for high-performance applications. Naturally, the actual cost varies from machine to machine and from compiler to compiler, but the difference in speed between a simple integer add and an indirect function call performing an integer add can easily be a factor of 50. In addition, the casts to derived classes in the implementation of the generic operations can be costly. To preserve type safety, they need to involve a run-time type check.

These performance problem can be addressed in some generality through interfaces relying on non-virtual functions and though whole-program analysis. They can also be partially addressed by ad hoc techniques involving a compiler "knowing" the definition of "critical" templates generating optimized code depending on properties of argument. However, the task of generating code equivalent to what is obtained for Standard C++ templates is distinctly non-trivial and beyond most compilers for realistic programs.

The more serious problems are logical: To be an argument to a template, a type must be derived from the constraint base class (interface) used by the template. This has several nasty implications

- a template writer must
  - turn a requirement to use certain operations into a requirement to derive from a named class providing those operations
  - represent operations as members of a class (i.e. no free-standing function can be allowed and dependent types has somehow to be represented as members)
- a would-be template argument type must
  - be a class
  - name the constraints of all template arguments for which it will be used as a base class
  - be defined after the constraints of all template arguments for which it will be used
  - provide the required operations with exactly the required signatures
  - implement its operations as defined for interfaces expressed in terms of base classes.

Turning the need to use some functions into a named class is a redundant logical step that leads to a proliferation of classes. The need to name those classes turns into a barrier against using separately developed (argument) classes and templates. For example, if I need to add two numbers, I may introduce a class **Addable** as the constraint for my

template. You – working independently of me – may introduce a class called **Add** to express that same need for your template. Someone who wants to use both our templates for a class **Number** must now derive **Number** from both **Addable** and **Add**. If he didn't initially, he must modify the definition of **Number** if he wants to use our templates. However, that may not be possible because **Number** may be from a separately developed library. The solution would then be to derive a new class **His_number** from **Number** and whichever of **Addable** and **Add** that it wasn't already derived from.

Thus, representing the need to add two numbers can easily (and realistically) spawn three classes. Unfortunately, those classes may not be easy to write. Consider:

```
struct Addable {        // my constraint
        virtual Addable operator+(Addable) =0;
};

template<class T : Addable> void my_fct(const vector<T>&);

class Add {     // your constraint
public:
        virtual const Add& operator+(const Add &) =0;
};

template<class T: Add> T your_fct(T,T);
```

How would someone write a class derived from both?

```
class Number : public Add, public Addable {
public:
        Addable operator+(Addable a) ;
        const Add& operator+(const Add& a);
        // …
};
```

I now have two (separate) virtual functions, each implementing the addition of **Number**s. Thus, the requirement to express template argument constraints through derivation breeds complexity as well as performance problems.

Furthermore, the requirement to express operations on template arguments as member functions does violence to some of the most common C++ programming idioms. For example, the most common way of defining an addition is as a free-standing function:

```
Number operator+(const Number&, const Number&);
```

This ensures that any conversions to **Number** are uniformly applied to both operands of +. Similarly, the requirement to derive would force us to avoid built-in types in favor of classes defined to mimic built-in types. For example:

```
class Int : public Addable {  // class to make int meet argument requirements
        int value;
public:
        // operations
};
```

I consider this ugly and inefficient (both in terms of programmer effort and execution overhead).

Finally, implementing arithmetic and logical operations as derived classes is not at all simple in C++. Consider an argument. Passing a **Number** by value as an **Addable** would lead to slicing, so in a signature-based concept scheme we must pass arguments by (const) reference. In the derived class function, we must use a dynamic cast to access the argument. For example:

```
const Add& Number::operator(const Add& a)
{
        Number& n = dynamic_cast<Add&>(a);
        // …
}
```

This is ugly and inefficient, but could be considered acceptable. However, consider the return value. Again, we can't return by value because that would lead to slicing. On the other hand, we can return a reference to a local object, so the returned object must be on some sort of free store. For example:

```
const Add& Number::operator(const Add& a)
{
        Number& n = dynamic_cast<Add&>(a);
        Number& res = *new Number;
        // …
        return res;
}
```

Preventing that return from becoming the source of a memory leak is non-trivial. Basically, it implies the need for a form of automatic garbage collection of such returned objects.

One way of mitigating the problems with the approach of defining argument constraints as base classes is to provide a large number of "standard" base classes for "all" users to rely on. However, that doesn't solve the fundamental problems of class proliferation, indirect expression of operations, inefficiency, and inelegance. It simply alleviates it partially through "central control" of style issue. In particular, this doesn't address the problem of separate development of templates and template argument types. In a language, such as C++, where no language owner exists who could exercise central control, this approach is a non-starter. Furthermore, it does not at all address the needs of people with existing code: Using base classes to express template argument constraints

translates into the need to write rather large amounts of non-trivial and costly mediation code for existing classes.

The function-match approach

A more promising approach is to abandon the requirement to derive from a constraints base class. Instead, we could require argument classes to "match" constraints specified as functions declared by a "**match**". For example:

```
match Addable {
        Addable operator+(Addable);
};

template<class T match Addable>          // pseudo code
        void my_fct(const vector<T>&);

class X {
public:
        X operator+(X);
        X operator*(X);
        // …
};

vector<X> vx;
// …
my_fct(vx);    // ok: X has a + like Addable
```

Given a suitable definition of **match**, **X** would match **Addable** because **X**, like **Addable**, provides an **operator+()** taking an operand of its own type and returning a value of its own type.

This approach eliminates several of the disadvantages of the base-class approach. In particular,

- as a writer of a potential template argument type, I need not name (all) the constraints that I might like to match. That's good because I couldn't possibly imagine all the possible constraints classes for a class that is useful in many programs.
- the problem of having to express a type so that it can be manipulated through a base class interface is eliminated; operations can be expressed colloquially.

With the introduction of a dedicated language construct, **match**, we gain the possibility to express a wider range of constraints. For example, we might enable the programmer to express the distinction between a member function and a free-standing function to allow a wider range of requirements of arguments:

```
match Addable {      // require free-standing functions
        extern Addable operator+(Addable,Addable);
        extern void trace(Addable);
```

```
};

template<class T match Addable> void my_fct(const vector<T>&);

class X {  /* … */  };
X operator+(X,X);

vector<X> vx;
// …
my_fct(vx);    // ok: X has a + like Addable
```

In addition, we might define **match** so that built-in types matched. For example:

```
vector<int> vi;
// …
my_fct(vi);     // ok: int has a + like Addable
```

The built-in type **int** could be defined to match by considering its + to have a suitable signature.

Naturally, this extra flexibility comes at the cost of some implementation complexity. The base-class approach described above could be entirely defined in terms of existing language rules. This function-match approach can't. Similarly, the base-class approach has an obvious implementation model (abstract classes) whereas the function-match approach could be implemented either by a vector of functions (like the base-class approach) or by per-specialization code replication (like the C++ template approach) for run-time performance. In principle, the choice of implementation techniques could be made on a per-specialization basis.

Unfortunately, in the context of C++, the function-match approach shares a major weakness with the base-class approach: When defining a function, a programmer has a range of implementation choices. Consider:

```
X operator+(X,X);
X operator+(const X&, const X&);
X& operator+(X&,X&);
const X X::operator+(const X);
X X::operator+(const X&);
X X::operator+(X) const;
```

Basically, there are 3*4*4*4==192 ways of expressing a function taking two arguments and returning a value, once the basic argument types and return type has been decided upon. By considering **volatile** we get an even higher number. Naturally, only a few of these combinations are actually used, but enough are used – and used reasonably – that specifying the exact type of an operation is a serious barrier to the use of a template that constrains its arguments that way. Any language that provides more than one way of passing an argument or returning a value suffers a variant of this problem.

Thus, the function-mach approach to template argument constraints is more flexible, easier to use, require less foresight from class designers, and simplifies the generation of efficient code as compared with the base-class approach. In particular, the function-match approach does not require the designer of a potential template argument class to derive from the template's constraint class. The complementary problems for the function-mach approach is that it requires novel language constructs, a new set of function compatibility rules, and more elaborate code generation strategies to reach traditional template performance.

Unfortunately, both approaches retain serious barriers to flexible use of templates in that they require an unrealistic degree of agreement between the template argument writer and the template writer. The fundamental problem is that both approaches are signature-based: The require the programmer to state **how** operations are implemented in terms of function signatures, rather than sticking to what is of interest to a programmer, namely **what** can be done with a template argument.

Consider a simple expression **a*b+c**. To use that in a template, signature-based approaches require the programmer to name the type of the intermediate result **a*b**. This is not always easy and can be constraining. For example:

```
match Mul {
        Mul operator*(Mul);
};

match Add {
        Add operator+(Add);
};

template<class A match  Mul, class B match Mul, class C match Add>
void f(A a, B b, C c)
{
        // …
        a*b+c;
        // …
}
```

Here we had to choose a return type for **Mul** and naturally we chose **Mul**. However, nowhere is it stated that a **Mul** must be a valid input to **Add**'s +; that is, that a **Mul** has to be convertible to an **Add**. Naturally, we might be able to specify that. For example:

```
match Add {
        Add operator+(Add);
};

match Mul {
        Mul operator*(Mul);
        operator Add();        // convert Mul to Add
};
```

This is easier to do in the function-match approach than in the base-class approach. However, even here, we have to state not just that a **Mul** can be used as an operand to **Add**'s +, but how that is achieved. For example, maybe this would have been a better set of constraints:

```
match Add {
        Add operator+(Add);
};

match Mul : Add {
        Mul operator*(Mul);
};
```

To make matters even worse, there is no fundamental reason to resolve **a*b+c** as **Add(a*b)+c**. Instead, we could choose **(a*b)+Mul(c)**, which would require a completely different relationship between the concepts.

Fundamentally, the problem is that the concepts can no longer be independent. That is, we can express such requirements only by having the foresight to design sets of mutually dependent concepts. The base-class approach places a serious burden on template argument class writers: they must express their classes in terms of constraints base classes. Both signature-based approaches place another serious burden on template argument class writers: they must express their operations in terms of function types. Furthermore, both signature-based approaches place a further burden on writers of concepts (typically, the template writers): Concepts must be expressed in such a way that dependencies among argument types within the various template implementations are reflected in the constraints function signatures.

The net effect of these burdens is to limit combined use of independently-developed templates, constraints, and classes. This leads to larger and more monolithic libraries that would more resemble class-hierarchy-based libraries than template-based libraries.

One obvious question "why don't you just abandon concerns about compatibility of notation and define a syntax for the ideal semantics?" will be answered later.

The usage-pattern approach

A more direct and abstract approach to constraints checking is to simply state which operations should be available for a template argument and how they should be used. The detailed analysis of class hierarchies and function signatures can be postponed from the points of template definition and template use until the point of template instantiation (specialization) where complete information is needed for generating code. For example:

```
concept Element {
        constraints(Element e1, Element e2) {
                bool b =e1<e2;          // Elements can be compared using <
                                        // and the result can be used as a bool
```

```
            swap(e1,e2);            // Elements can be swapped
        }
};

template<Element E> void sort(vector<E>& c)
{
        // …
        if (c[i]<c[j]) swap(c[i],c[j]);
        // …
}

class Number {
        int i;
        // …
};

bool operator<(Number,Number);
void swap(Number&,Number&);

Container<Number> vn;
// …
sort(vn);       // ok: we can compare and swap Numbers

Container<int> vi;
// …
sort(vi);       // ok: we can compare and swap ints
```

This approach is based on usage patterns and has its origins in techniques for constraints checking used from the earliest days of templates [Stroustrup,1994]. In particular, its inspiration comes from constraints checks implemented in constructors. The syntax here preserves the function-like syntax for expressing a set of constraints as a way of introducing variables of the constrained type. Basically, a type matches a concept it the concept's **constraints** function compiles for arguments of that type. A constraints function is never executed so it imposed no run-time overhead. A **constraints** function does not introduce any novel syntax; its code follows the usual language rules and anything that can be expressed in C++ can now be used as a constraint on a template parameter.

The use-pattern approach can also be seen as a further abstraction of the function-match approach: The function-match approach was made more flexible than the base-class approach by eliminating the need to explicitly naming the constraints classes in template argument classes. The use-pattern approach is made more flexible than the function-match approach by eliminating the need to mention function signatures in the constraint. This simplifies the expression of constraints and also provides a direct way of expressing relationships among constraints (see below).

Naturally, we can't generate code without knowing the exact type of every object and the signature of every function. Use of **concept**s is not a substitute for type checking

– complete type checking is necessary for code generation. Rather, use-pattern **concepts** is a complement to type checking that separates the concern of the programmer trying to provide flexible and general facilities (typically) in a library from the concerns of the code generator. Using **concept**s ensures that errors are caught much earlier in the compilation process than they are for unconstrained template arguments: If a template definition uses an operation not mentioned in its concepts, an error immediately occurs. If a template specialization is used where an argument doesn't provide the operations required from its concept, an error immediately occurs.

We must consider two key questions:

- can a use of a template pass concept checking, yet (later) fail type checking?
- can a template definition pass concept checking, yet (later) fail type checking?

The key strength of the signature-based approaches is that for those, the answer to these two questions is obviously "no". For usage-pattern concepts, the answers are "no" and "yes, but no invalid specialization is ever attempted by the compiler". This implies that a user of a template gets immediate feedback on any error, and that the template definer gets some, but not perfect, feedback about potential errors. Basically,

- at the point of specialization (use), a type check of a usage-pattern happens – the concept is compared to a type
- at the point of template definition, the definition's use of template arguments is checked against the usage-patterns of those arguments' concept – every operation must be specified by a concept

The exception to this complete checking is a use of a traditional template parameter specified only as **class** (or equivalently **typename**). Arguments to parameters cannot be checked until the template definition and all its arguments are available.

Consider how we might implement the checking of a template definition. For example:

```
template<Value_type && Addable T>
T add(const T& a, const T& b)
{
        return a+b;
}
```

In the definition of **add(),** we use two operations + and copy. These are clearly provided by the concepts. However, we have not explicitly specified that the result of + must be usable as the as the source of the initialization of a **T**. Therefore, if that template definition is allowed, a specialization might fail. For example:

```
struct Odd {
        void operator+(Odd);
};

Odd x,y;
Odd z = add(x,y);      // passes concept check, fails type check
                       // can't initialize an Odd with void
```

Fortunately, that template definition would not compile. The problem occurred because the specification of the template's requirements was incomplete: the result of adding two **T**s is not necessarily a **T** or something that can be assigned to a **T**. One key difference between concepts – even usage-pattern concepts – and constraints classes is that a template definition is constrained to use only the operations specified for its argument in their concepts. If that's not desired, we can fall back on the usual unconstrained **class** template arguments (possibly supported by constraints classes). For compatibility reasons, and possibly to avoid some complicated uses of concepts, banning unconstrained template parameters is not an option for C++.

So, how might we repair the example above? Somehow, we must add the requirement that the result of + can be used to initialize a T. One obvious solution is to define a new concept:

```
concept Add {                    // We can copy and add Adds
        constraints(Add x) { Add y = x; x = y; x = x+y; Add xx = x+y; }
};

template<Add T>
T add(const T& a, const T& b)
{
        return a+b;
}
```

Below, we'll see how we can build new concepts out of existing ones so as to avoid restating requirements.

A compiler can easily detect that a template uses an operation that isn't specified in its concepts. After all, the basic checking consists of checking each operation against the set of operations specified. However, since no types are specified at the point of definition of a template, complete type checking is impossible. In particular, it is not possible to generate code from a template to be called directly from points of use relying on a vector of functions representing the operations on each argument type. At most, some semi-compiled form of the template can be produced.

Now consider a use of a template. At that point, the compiler has available both the concepts representing all the operations that a template uses and the argument types. The compiler can check both that every required operation is provided and that every use specified in the concept type checks. Thus, a successful concept check implies a successful type check (much) later. In particular, the attempt to add two **Odd**s, **add(x,y)** is caught immediately.

Concept composition

A complete technical description of **concepts** is left to a companion paper [Stroustrup,2003d]. However, the sections below shows how the problems described above and a few more advanced cases can be handled using usage-pattern-based concepts. One major question about usage-pattern concepts is how easy it is to express large classes of requirements elegantly. Only experience can tell, but by providing the

known composition mechanisms of C++ we provide a proven degree of flexibility to concepts.

Simple concepts

A simple **concept** just specifies a usage pattern that a type must match to be an argument for a template parameter specified by the **concept**. For example:

```
concept Value_type {
        constraints(Value_type a)
        {
                Value_type b = a;               // copy initialization
                a = b;                          // copy assignment
                Value_type v[] = { a };         // not reference
        }
};
```

That is, a **Value_type** is any type with object that can be copied by initialization or assignment. The only new syntax introduced is the **concept** keyword (used exactly like **class**) and the **constraints** pseudo-function. The only new semantics is the use of type checking of a **constraints** pseudo-function to determine whether or not a type matches a **concept**. For example:

```
template<Value_type V> void swap(V a, V b)
{
        V tmp = a;
        a = b;
        b = a;
}

class Glob {
private:
        void operator=(Glob&);      // prevent copying
        // …
};

void f()
{
        int a;
        int b;
        swap(a,b);      // ok: we can copy ints
        Glob x;
        Glob y;
        swap(x,y);      // error: we can't copy Globs
}
```

Note how the c**oncept** name **Value_type** fits seamlessly and naturally into the template definition syntax. Another pleasant aspect of usage-patterns based **concept**s is that a use tends to be much shorter to express than the list of function declarations needed in a signature-based alternative. It is also easy to express that a free-standing function is needed:

```
concept Std_printable {
        constraints(Std_printable x, std::ostream& os)
        {
                os << x;
        }
};

concept X_printable {
        constraints(X_printable x)
        {
                xprint(x);
        }
};

void xprint(const Glob&);

template<Std_printable T> void f(const T& );
template<X_printable T> void f(const T& );

void f(string s, Glob g)
{
        f(s);    // calls the f() taking an Std_printable
        f(g);    // calls the f() taking an X_printable
}
```

A **concept** is checked at each point of instantiation relying on the set of functions available at the point of use.

Overload resolution is as for template functions of unconstrained arguments, except that the **concept**s can eliminate functions before they become candidates for overload resolution.

With **concept**s, overloading of class templates becomes feasible. It is an error to try to instantiate a class for which there is not exactly one match (unless the ambiguity is resolved by inheritance rules).

Concept composition

The operators **&&** (and), || (or), and **!** (not) can be used to combine concepts for a template argument. For example:

```
template<Std_printable && value_type T> class X { /* … */ };
```

```
template<Std_printable || X_printable T> class Y { /* … */ };

template<Std_printable && !X_printable T> class Z { /* … */ };

template<!Std_printable && X_printable T> class Z { /* … */ };
```

That is **&&**, ||, and **!** act as declarator operators for template arguments allowing the programmer some control over selection of template definitions based on template arguments.

A template parameter specified with the traditional **class** accepts every type, so such parameters can be used to handle cases where no more specific match was found. For example, consider a slightly oversimplified definition of a pointer:

```
concept Pointer {
        constraints(Pointer p)
        {
                void* q = p;    // just conventional pointers to objects
                Pointer pp = p; p = pp;
                ++p; --p; p++; p--; p+1; p-1;
                *p;
        }
};

template<Pointer P> void f(P a) { /* … */ };
template<class X> void f(X a) { /* … */ };

void fct(int a, int* p)
{
        f(a);    // use the general X
        f(p);    // use the Pointer X
}
```

Thus, the traditional **template<class T>** fits into the concept mechanism as the most general (least constrained) case.

Parameterized and derived concepts

Since a **concept** models a set of types, the usual ways of composing new types out of existing ones, such as parameterization and derivation, naturally apply to **concept**s. For example, consider the standard library requirements for iterators.

```
template<Value_type V> concept Forward_iterator {
        constraints(Forward_iterator p)
        {
                Forward_iterator q = p; p = q; // we can copy iterators
```

```
            V v = *p; q = &v;      // the iterator points to the value type
            p++; ++p;              // we can increment an iterator
        }
}
```

A **concept** template is a concept generator. That is, a specialization of a **concept** template, such as **Forward_iterator<int>** is a **concept**. A **Forward_iterator** is parameterized by the iterator's **Value_type**. Such parameterization will be a major way of composing concepts out of other concepts.

```
template<Forward_iterator<Value_type> Iter>
        Iter find (Iter  first, Iter last);

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

How can we build **Random_access_iterator** from a **Forward_iterator**? (ignoring other iterators for this discussion). The obvious answer is concept inheritance;

```
template<Value_type V>
concept Random_access_iterator : Forward_iterator<V> {
    constraints(Random_access_iterator p) { --p; p--; p+1; p[1]; p-1; }
};
```

That is, to specify the requirements for a **Random_access_iterator** we simply add the additions to the requirements for a **Forward_iterator**. Now we can overload a template function based on its template argument concepts. As with ordinary function overloading, the most specific (most derived) concept is chosen. For example:

```
template<class T> void poke(T);
template<Forward_iterator<Value_type> T> void poke(T);
template<Random_access_iterator<Value_type> T> void poke(T);

poke(2);        // call the general poke()
int* p;
poke(p);        // call poke(Random_access_iterator)

template<Value_type T> class I {   // some iterator template
public:
    I& operator++();
    I operator++(int);
    // no + defined, so I isn't a Random_access_iterator
    // …
 };

poke(i);        // call poke(Forward_iterator)
```

Random_access_iterator is said to be more constraining than **Forward_iterator** because fewer types match **Random_access_iterator** than mach **Forward_iterator.** The least constraining concept (for a template type argument) is the traditional and special **class** that matches every type. Apart from **class**, the least constraining concept is

```
concept Noop {
        constraints() { }
};
```

Every type matches **Noop**, but is useless for most purposes because a template can perform no operation on an argument declared to be a **Noop**.

Constraints involving multiple arguments

Parameterization also allows us to express constraints involving more than one template argument. For example, we can express the general form of addition:

```
template<class L, class R> concept Add {
        constraints(L x, R y) { x+y; }
};
```

The result of applying arguments to a templated concept, such as **Add,** is a concept, so we might use **Add** like this:

```
template<Value_type T, Add<T,T> Res>
Res add(const T& a, const T& b)
{
        return a+b;
}
```

This is logical and solves the problem of how to refer to the type of **a+b**. However, it's "odd" in that the **add()** template suddenly acquired another template parameter which can always be deduced from **T**. Furthermore, the declaration of **add()** is (still) incomplete/illegal in that it fails to state that one **Add<T,T>** (say **a+b**) can be assigned to another. Let's solve the second problem first:

```
template<Value_type L, Value_type R> concept Assign {
        constraints(R x) {
                L y = x;        // initialization
                y = x;          // assignment
                L yy[] = { x }; // not reference
        }
};

template<Value_type T, Assign<T,Add<T,T>> Res>
```

```
Res add(const T& a, const T& b)
{
        return a+b;
}
```

In other words, **Res** is any type of which you can add two values and assign the result to a variable of that same type**.** As it happens, **Res** is also guaranteed to be **T** for every **T** that passes concept checking. Basically, **Assign<T,Add<T,T>>** is used as a predicate for **T** and the name **Res** isn't needed. We get a clearer syntax if we separate that predicate from the template argument lists:

```
template<Value_type T> where Assign<T,Add<T,T>>
T add(const T& a, const T& b)
{
        return a+b;
}
```

That is, after a template argument list we can add a **where** clause specifying a constraint on the argument type(s) without adding logically spurious template arguments. What happens is that when you specialize a templated **concept**, the result is a **concept**; if that specialization isn't valid, the template instantiation of which it is a part fails. Such specialization is usually done to match a templated type. However, here we just form that concept, and if that specialization is valid, the **where** clause succeeds; if not, the whole template instantiation is deemed to have failed. In other words, we use a templated concept as a predicate for its argument types in much the same way constraints template classes are currently used.

We can now generalize **add()** to handle different left-hand and right-hand argument types:

```
template<Value_type L, Value_type R> where Assign<L,Add<L,R>>
L add(const L& a, const R& b)
{
        return a+b;
}
```

We could generalized **add()** to have it's return type as a template argument. Naturally, that argument would be constrained as above:

```
template<Value_type L, Value_type R, Add<L,R> Res>
        where Assign<Res,Res>
Res add(const L& a, const R& b)
{
        return a+b;
}
```

This shows how a **where** clause can be used to add constraints to even a single argument. Equivalently, we could write:

```
template<Value_type L, Value_type R, Add<L,R> && Value_type Res>
Res add(const L& a, const R& b)
{
        return a+b;
}
```

Finally, we might like to have the return type deduced. We might try

```
template<Value_type L, Value_type R>
Add<L,R> add(const L& a, const R& b);
```

However, that will not work because **Add<L,R>** is a concept, not a type. All we said was that the **add()**'s return type must be some type that matches **Add<L,R>** and that's not a sufficiently specific for a declaration. Using the **decltype/auto** proposal [Järvi,2003] and its generalization to use concepts [Stroustrup,2003b], we could write:

```
template<Value_type L, Value_type R>
Add<L,R> add(const T& a, const R& b)
{
        return a+b;
}
```

Alternatively, we could express the return type using **decltype**:

```
template<Value_type L, Value_type R>
Add<L,R> add(const T& a, const R& b) ->decltype(a+b);
```

Finally, we can handle the **a*b+c** example without specifying the type of the temporary:

```
template<class L, class R> concept Mul {
        constraints( L a, R b) { a*b; }
};

template<class L, class R> concept Add {
        constraints( L a, R b) { a+b; }
};

template<class A, class B, class C> where Add<Mul<A,B>,C>
void f(A a, B b, C c)
{
        // …
        a*b+c;
        // …
```

```
}
```

Note that here no constraints are placed on the argument types except that they must interoperate as specified. For example, as long as we want to maintain complete generality we cannot require those types to be **Value_type**s because * and + might do their work without making copies.

Associated Types

For many kinds of generic programming it is important to be able to express types related to some other type. For example, how can we state that the iterator type associated with a container really needs to be an iterator? For example:

```
template<Forward_iterator I> void is_forward_iterator(I i) { }

template<Value_type V> concept Container {
        constraints(Container c) {
                c.begin();      // a standard container must have a begin()
                                // and begin() returns an iterator
                is_forward_iterator(c.begin());
                // …
        }
};
```

This technique relies on the "helper function" **is_forward_iterator()** that compiles if and only if its argument is a **Forward_iterator**. It would be possible to simplify this test by introducing special syntax, but that isn't necessary, and the workaround doesn't seem onerous.

Non-type template arguments

In addition to type arguments, a template can have non-type arguments. For example:

```
template<class T, int N> class Buffer { /* … */ };
```

Naturally, we'd like **concept**s to cover non-type arguments as well as combinations of type arguments and non-type arguments. Unfortunately, this requires an extension of the **concept** idea to handle values. Consider how we might generalize concept to allow us to write:

```
template<Odd_int N> class X { /* … */ };
```

Here, we'd like and instantiation **X<n>** to succeed from odd values, but not for even ones. To do that, we clearly have to do a calculation involving **N** producing a Boolean indicating success or failure. For example:

```
concept Odd_int {
        bool constraints() { return Odd_int%2; }
};
```

This is a direct parallel to the "usual" **concept**s for types. The name of the concept is used to represent the template argument. "Usually", that's a type; here, it's a value. "Usually", **constraints()** is a pseudo function that needs not be evaluated. Here, it must be evaluated at compile time, with a value **true** meaning that instantiation might succeed (if no other errors are found). Given that, we get

```
X<7> a;         // ok
X<8> b;         // error: 8 is not an Odd_int
```

Consider a more realistic example involving a **where** clause:

```
template<int n, int m> concept LE {
        bool constraints() { return n <= m; }
};

template<Value_type T, int N> where LE<sizeof(T)*N,1024> class Buffer {
        // …
};

Buffer<int,64> b1;              // ok (assuming sizeof(int)<=16)
Buffer<double,200> b2;          // error (assuming sizeof(double)==8)
```

This sketch of a design for concepts for non-type arguments is presented primarily for completeness. Whichever syntax or semantics are chosen for concepts must include non-type arguments in a consistent way.

Syntax

The syntax for usage-based concepts was chosen to require minimal changes to existing C++ concepts and syntax. It uses the **constraints** "pseudo function" to introduce names. For example

```
concept Element {
        constraints(Element e1, Element e2) {
                bool b = e1<e2;         // Elements can be compared using <
                swap(e1,e2);            // Elements can be swapped
        }
};
```

The name **constraints** might be considered misleading: from the point of view of a user it names constraints, but from the point of view of the template provider it names the operations offered. Ideally, we would not use any name at all. We could consider

cleaning up the syntax by eliminating the redundant **constraints()** and the added set of parentheses that go with it. That would make the concept body more like a function body. For example:

```
concept Element {
        extern Element e1;
        extern Element e2;
        bool b = e1<e2;         // Elements can be compared using <
        swap(e1,e2);            // Elements can be swapped
};
```

Note that we'd have to use **extern** or some other workaround to introduce variables without making an unwarranted assumption that every **Element** type must have a default constructor. If we want to eliminate **constraints()** without relying on some workaround, we need to introduce some novel syntax to introduce names. For example:

```
concept Element {
        uninitialized Element e1;
        uninitialized Element e2;
        bool b = e1<e2;         // Elements can be compared using <
        swap(e1,e2);            // Elements can be swapped
};
```

or

```
concept Element(Element e1, Element e2) {
        bool b = e1<e2;         // Elements can be compared using <
        swap(e1,e2);            // Elements can be swapped
};
```

or even

```
concept Element(e1,e2) {
        bool b = e1<e2;         // Elements can be compared using <
        swap(e1,e2);            // Elements can be swapped
};
```

However, there is too much unique syntax in C++ already and **Element** isn't really parameterized.

Any syntax chosen for **concept**s must be general enough to support non-type arguments as well as type argument.

The pseudo-signature approach

Using signatures to express **concept**s led to a problem with similar, but different, signatures and to overconstraining template argument types. We could express concepts

in terms of something that was a bit like signatures, but didn't lead to overconstraining. That would bypass the implementation concerns inherent in function signatures. For example:

```
concept Element {
        <(Element,Element) -> bool
        swap(Element,Element) -> void
};
```

The "odd" (but logical) syntax is used to indicate a new and different semantics. The idea is that for a type to match Element it must provide a < operator and function that in some way takes two element and a **swap()** that in some way takes two Elements. For example:

```
class X {
public:
        void swap(X&);
        // …
};

bool operator<(const X&, X);
```

These declarations make **X** an **Element**, and given

```
void swap(int&,int&);
```

int is an **Element**.

To avoid overconstraining return types, we need to introduce a placeholder meaning "whenever the return type is". For example:

```
concept Mul {
        *(Mul,Mul) -> deduced
};
```

For **1*1** the deduced type will be **int**, for **1.0F*1.0F** the deduced type will be **double**.

Basically, such "pseudo signatures" can express what usage-pattern can (and vise versa). Thus, the two approaches can be seen as different syntactic representations of the same idea. The advantage of the usage-pattern approach is that it minimizes the set of new syntax and appears to lead to more compact concept definitions. To complete the pseudo-signature approach, we'd have to invent syntax to express subtype relationships.

Conclusions

The unconstrained template arguments provided by ISO Standard C++ provides noticeably advantages in terms of flexibility and run-time efficiency. However, they also lead to serious implementation complexity due to lack of separation between the template definition context and the template argument context. Furthermore, the lack of template

argument constraints leads to poor error diagnostics and an inability to use usual type-based techniques, such as overloading for templates. Constraining template arguments in terms of base classes or function signatures carries a high cost in lack of flexibility and performance. This paper discusses these problems and presents an alternative, usage-pattern **concept**s, which preserve the generality and abstractness of unconstrained template arguments while allowing a precise, complete, and useful specification of template arguments.

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. Brian Kernighan was – as often before in the evolution of C++ – an attentive, active, and challenging listener. Gabriel Dos Reis persevered through many discussions, variants of concepts, and many draft papers; he is the co-author of companion papers. In particular, Gabriel pointed out the need to consider non-type template arguments.

References

[Austern,2002] Matt Austern: ???.
[BOOST,200?] ???: ???.
[Garcia,2003] Garcia, et al: *A comparative study of language support for generic programming*. OOPSLA 2002.
[Dos Reis] Gabriel Dos Reis: *Generic Programming in C++: The Next Level*. ACCU 2002.
[Jävi,2003] Jaakko Järvi and Bjarne Stroustrup: Mechanisms for querying types of expressions – Decltype and auto revisited. N1527=03-0110.
[Stroustrup,2003a] Bjarne Stroustrup and Gabriel Dos Reis: *Design criteria for concepts.*
[Stroustrup,2003b] Bjarne Stroustrup: *Concepts – syntax and composition.*
[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994.
[Stroustrup,2003c] Bjarne Stroustrup: *Expressing the standard library requirements as concepts*. To be written.
[Stroustrup,2003d] Bjarne Stroustrup: *Concepts: template argument checking for C++*. To be written.