A concept design (Rev. 1)

Bjarne Stroustrup[1] and Gabriel Dos Reis

Texas A&M University

bs@cs.tamu.edu and gdr@cs.tamu.edu

Abstract

We describe a design of "concepts" (a type system for types) for improved support of generic programming in C++. This paper presents a current best-effort design based on previous work followed by lengthy discussions of technical details. The design is based on the principles that simple things should be simple (whereas more advanced programming techniques may require some extra work), that there should be no performance degradation compared to current template techniques, and that essentially all current template techniques should be improved by the use of concepts. The design achieves perfect checking of separately compiled templates at the cost of minor restrictions on the instantiation mechanism.

Changes for Rev. 1

We have corrected and clarified many points, cleaning up the presentation.
We have strengthened the argument for optional asserts in appendix A.
We have explained the use of negative asserts and their importance (Section ???).
We have factored out the (still incomplete) discussion of combinations of concept-checked and old-style template parameters (Appendix C).
We addressed the issue of how to deal with concepts that differ only in semantic (Section ???).

1 Introduction

This paper describes a best-effort design of concepts for C++. For clarity, most discussion of alternatives are placed elsewhere [Stroustrup, 2003a-c] and motivation is kept brief. We do not assume that this design is perfect; rather, our assumption is that progress can best be made based on a clearly stated concrete design. The design is based on previous work, including [Stroustrup, 2003a-c], [Siek, 2005].
"Concepts" is a type system for C++ types. Its main aims are

- Perfect separate checking of template definitions and template uses
- No performance degradation compared to C++ that doesn't use concepts
- Simplify all major current template programming techniques

[1] And AT&T Labs – Research

- No loss of expressiveness compared to current template programming techniques
- Simple tasks are expressed simply (close to a logical minimum)

We consider the design coherent and also complete in the sense that it addresses all the major issues people usually bring up in the context of concept.

It has been suggested [Siek, 2005] that the word "constraints" is a better characterization for "concepts" than "types of types" [Stroustrup, 2003a]. However, a concept describes the (syntactic and semantic) properties of a type. That's the traditional role of types; for example, the type of a variable describes its syntactic and semantic properties. A concept does more than simply constrain a type.

The order of topics aims to reflect the likely number of people directly affected:

- §2-5 – Use and definition of concepts
- §6-7 – Technical details
- §8 – Concepts in the specification of the STL

Appendices
- A – Why asserts must be optional
- B – Haskell type classes
- C – Mixing concept parameters and unconstrained parameters (incomplete)
- D – semantics properties in concepts (incomplete)

2 Using concepts

First we show some uses of concepts (types of types); later, we show how to define them.

2.1 Catching errors early

Consider the STL **fill()**:

```
template<class ForwardIterator, class ValueType>
void fill(ForwardIterator first, ForwardIterator last, const ValueType & v)
{
        while (first!=last) { *first = v; ++first; }
}
```

The names **ForwardIterator** and **ValueType** are descriptive and their requirements are specified in the standard. However, those requirements are not known to the compiler. Using concepts, we can express the requirements of **fill()**'s arguments directly:

```
template<Forward_iterator For, Value_type V>
        where Assignable<For::value_type,V>
void fill(For first, For last, const V& v)
```

```
{
        while (first!=last) { *first = v; ++first; }
}
```

That is, The first two arguments of **fill()** must be of the same type and that type must be a **Forward_iterator**. The third argument must be of a **Value_type** and it must be possible to assign an object of the **Value_type** to an object of the **Forward_iterator**'s **value_type**. This is exactly the standards requirements.

Note in particular that it is not possible to state the standard's requirements purely on each argument type in isolation from the other. The assignment is an essential and reasonable part of **fill()**'s specification that can only be stated as a relationship between the argument types. For example, the **Forward_iterator** could be a **double*** and the **Value_type** a **short**.

The use of a concept as the type of a template parameter type is simply a shorthand for expressing it as a predicate. For example, this – more longwinded – version of **fill()** is equivalent to the last:

```
template<class For, class V>
        where Forward_iterator<For>
                && Value_type<V>
                && Assignable<For::value_type,V>
void fill(For first, For last, const V& v)
{
        while (first!=last) { *first = v; ++first; }
}
```

We can read this as "for all types **For** and for all types **V**, such that **For** is a **Forward_iterator** and **V** is a **Value_type** and a **V** can be assigned to a **For::value_type**". If you like Math, you'll find the formulation familiar. When looking at templates and concepts, it is useful to remember that **<class T>** means "for all types **T**" and **<X T>** means "for all types **T**, such that **T** is an **X**".

Note the way **<class T>** allows every possible operation on **T** to be written in the template definition whereas **<C T>** allows only operations explicitly specified by **C** to be used. So (as ever), if a template specifies a parameter **<class T>** (or equivalently **<typename T>**), absolutely every type can be passed as a template argument and the template definition may apply absolutely any operation to it (checking comes later when we know an actual type). When we add a **where** clause, such as **C<T>**, we constrain arguments to types for which **C<T>** is true and limit the operations that the template definition can apply to **T** to those specified in **C**.

We consider the use of **Assignable**, that is "can be assigned using =" typical. The simpler test "the types are the same" (see Section 7.4) is far less frequent in common C++ usage. Similarly, "comparable to" or "convertible to" are far more important than "same type"

constraints. Obviously, this complicates specifications and checking, but this has been a fact of life in C and C++ for more than 25 years, so we have to deal with it. For a further simplification/refinement of the specification of **fill()**, see Appendix C.

2.1.1 Use errors

Consider some errors:

```
int i = 0;
int j = 9;
fill(i, j, 9.9);
```

The "old style" first definition of **fill()** has no problems with this until we come to dereference **first**: ***first=v;** we can't dereference an **int**. Unfortunately, this is discovered only at the point of instantiation and is reported (typically obscurely) as an illegal use of ***. Using concepts, however, the second "new style" definition of **fill()** will give an error at the point of use: an **int** is not a **Forward_iterator** because it lacks the dereference operation *.

The logical distinction becomes obvious if we have only a declaration and not a definition of **fill()** available at the point of call:

```
template<class ForwardIterator, class ValueType>
void fill(ForwardIterator first, ForwardIterator last, const ValueType & v);

int i = 0;
int j = 9;
fill(i, j, 9.9);
```

There is nothing here to allow the compiler to diagnose an error. To contrast:

```
template<Forward_iterator For, Value_type V>
        where Assignable<For::value_type,V>
void fill(For first, For last, const V& v);

int i = 0;
int j = 9;
fill(i, j, 9.9);
```

Here the compiler knows that **For** must be a **Forward_iterator** and can easily determine that **int** is not a **Forward_iterator**.

On the other hand, consider:

```
int* i = &v[0];
int* j = &v[9];
```

```
fill(i, j, 9.9);
```

This will work with both definitions of **fill()** and generate the same code. The only difference is that when we use the "new style" concept-based definition, we know that the code will compile as soon as we have checked the point of call.

Note that the size of the new style and old style declarations are quite similar: (23 tokens, 124 characters) vs. (32 tokens, 122 characters).

2.1.2 Definition errors

In addition to catching errors at the point of use of a template, concept checking catch errors in the definition of a template. Consider

```
template<Forward_iterator For, Value_type V>
        where Assignable<For::value_type,V>
void fill(For first, For last, const V& v)
{
        for (; first<last; --last) *last=v;
}
```

This will not compile because a parameter defined using a concept can only use properties defined for that concept. The concept **Forward_iterator** provides ++ and != but not -- or <. The only operations that can be used for a parameter defined using a concept are those defined in the concept (see Section 7 for details). Any use beyond that causes the definition not to compile.

2.2 Overloading on concepts

Overloading of templates has been confounded by the fact that there is no general way of choosing between two templates with the same number of template parameters. For example, we'd like to have versions of **sort()** that apply directly to a container and versions of **sort()** that accept bi-directional iterators, in addition to the current Standard versions that take random access iterators:

```
template<class RandonAccessIterator>
        void sort(RandonAcessIterator first, RandonAcessIterator last);
template<class RandonAcessIterator, class Compare>
        void sort(RandonAcessIterator first, RandonAcessIterator last,
                Compare comp);

template<class BidirectionalIterator>
        void sort(BidirectionalIterator first, BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
        void sort(BidirectionalIterator first, BidirectionalIterator last,
                Compare comp);
```

```
template<class Container> void sort(Container c);
template<class Container, class Compare>
        void sort(Container c, Compare comp);
```

Note that because template parameter names are not semantically significant, we did not – despite appearances – manage to define separate versions of **sort()** for random access and bi-directional iterators. This can be achieved only through a combination of traits and reliance on language subtleties. Consider how you would ensure that these calls are appropriately resolved:

```
void f(list<double>& ld, vector<int>& vi, Fct f)
{
        sort(ld.begin(), ld.end());     // call bi-directional iterator version
        sort(ld);
        sort(vi);
        sort(vi, f);                    // call container version
        sort(vi.begin(), vi.end());     // call random access iterator version
}
```

Given ingenuity and time, it can be done. However, many standard library algorithms have restrictions on their interfaces that can be best explained as implementation complexity showing through in the interface.

Using concepts we get:

```
template<Randon_access_iterator Iter> void sort(Iter first, Iter last);
template<Randon_access_iterator Iter, Compare Comp>
        where Convertible<Comp::argument_type,Iter::value_type>
void sort(Iter first, Iter last, Comp comp);

template<Bidirectional_iterator Iter> void sort(Iter first, Iter last);
template<Bidirectional_iterator Iter, Compare Comp>
        where Convertible<Comp::argument_type, Iter::value_type>
void sort(Iter first, Iter last, Comp comp);

template<Container Cont> void sort(Cont c);
template<Container Cont, Compare Comp>
        where Convertible<Comp::argument_type, Cont::element_type>
 void sort(Cont c, Comp comp);
```

Now the example calls are easily resolved based on the knowledge that **vector** and **list** are containers (and not iterators) and that **vector** iterators are random access iterators whereas **list** iterators are (just) bidirectional iterators.

The same overloading mechanism provides a straightforward solution to the implementation of **advance()** that currently requires a trait class:

```
template<Forward_iterator Iter> void advance(Iter& p, int n)
{
        for (int i=0; i<n; ++i) ++p;
}

template<Random_iterator Iter> void advance(Iter& p, int n)
{
        p += n;
}

void f(list<double>& ld, vector<int>& vi)
{
        auto p = ld.begin();
        auto q = vi.begin();
        // …
        advance(p, 7);          // Forward_iterator version
        advance(q, 3);          // Random_iterator version
}
```

This implies that many problems that currently require traits can be expressed as simple overloading (or specialization; see sections 6.3-6.4) without the need for auxiliary traits classes. Basically, concepts act as traits for the purpose of "dispatching" or rather traits is a technique for dispatching on types.

2.3 The "do the right thing" wart

Consider this fraction of the standard library **vector** and an example of its use:

```
template<class T> class vector {
        // …
        vector(size_type n, const value_type& x = value_type());
        template<class InputIterator>
                vector(InputIterator first, InputIterator last);
};

vector<int>(100,1);
```

For this source code, the template constructor will be chosen over the non-template one because **size_type** is an unsigned type so that the non-template constructor requires a conversion. Usually, that is not what the user intended, but fortunately that use of the template constructor will not compile. The library has the "do the right thing" rule specifically to handle this problem. Such special rules are warts. Concepts provide a straight forward alternative:

```
template<Value_type T> class vector {
        // …
        vector(size_type n, const value_type& x = value_type());
        template<Input_iterator Iter> vector(Iter first, Iter last);
};

vector<int>(100,1);
```

Again, **int** isn't an **Input_iterator** so the template constructor will not be called.

2.4 Helper functions: Advance

For entities dependent on a template parameter specified using concepts, a template definition can use only operations specified for those concepts. However, a template definition is not restricted to use only members of the template parameter types. As ever, we can call functions that take arguments of template parameter types. For example:

```
template<Forward_iterator Iter> Iter advance(Iter p, int n)
{
        while (n-->0) ++p;
        return p;
}

template<Forward_iterator Iter> void foo(Iter p, Iter q)
{
        // …
        Iter pp = advance(p,4);
        // …
}
```

The details of how overloading of **advance()** is handled (without loss of generality or performance compared to current usage) are in Sections 6.3-6.5.

3 Defining concepts

So, how do we define a concept? We simply list the properties (functions, operations, associated types) required:

```
concept Forward_iterator<class Iter>{      // see also Section 8
        Iter p;                            // uninitialized
        Iter q =p;                         // copy initialization
        p = q;                             // assignment
        Iter& q = ++p;                     // can pre-increment,
                                           // result usable as an Iter&
```

```
        const Iter& cq = p++;          // can post-increment
                                       // result convertible to Iter
        bool b1 = (p==q);              // equality comparisons,
                                       // result convertible to bool
        bool b2 = (p!=q);
        Value_type Iter::value_type;       // Iter has a member type value_type,
                                           // which is a Value_type
        Iter::value_type v = *p;       // *p is assignable to Iter's value type
        *p = v;                        // Iter's value type is assignable to *p
};
```

Basically, a concept is a predicate on one or more types (or values). At compile time, we determine if the arguments have the properties defined in the body of the concept. The parameters (formal arguments) have the same form as template parameters, but note that a concept parameter has no properties beyond the ones explicitly mentioned, thus:

```
concept No_op<class T> { };
```

This defines a concept **No_op** that defines no properties for **T**: Any type is a **No_op**, but since no operations can be used on a **No_op**, a **No_op** parameter is useless except as a place holder.

The concept body for **Forward_iterator** is roughly a copy of the syntactic part of the standard's requirements tables relating to a forward iterator. With two exceptions the notation is just the ordinary. For example **p!=q** simply says that two **Forward_iterator**s can be compared with **!=** and that the result can be converted to **bool**. Note that this says nothing about exactly how **!=** is implemented for a type **T**. In particular, **!=** need not be a member function or take an argument of type **const T&** or return a **bool**. All the usual implementation techniques can be used [Stroustrup, 2003a]. Note also that we specify only that the result can be converted to a **bool**; no other operations are required for or allowed on the result of **p!=q**. See also §6.6.

Here,

```
Iter p;
```

introduces the name **p** of type **Iter** for us to use when expressing concept rules. It does not state that an **Iter** require a default constructor. That would be expressed as

```
Iter();
```

or

```
Iter p = Iter();
```

This overloading of the meaning of **Iter p;** is unfortunate, and could be eliminated by some special syntax, such as

```
unitinitalized Iter p;
(Iter p);
Var<Iter> p;
```

Such new syntax has its own problems, however.

The declaration,

```
Value_type Iter::value_type;
```

states that the member name **value_type** must be applicable for a **Forward_itarator** in a template definition. By default, that means that **Iter** must have a member type **Iter::value_type**. However, there is a way to make **value_type** refer to the type **T** for a pointer type **T*** (Section 4). Such types are often called "associated types" [Siek, 2005].

Note that we require that we can say **Iter&**. By itself, that means that **Iter** cannot be **void** (because **void&** is not valid). Similarly, had we required **Iter***, that would have implied that **Iter** couldn't be a reference (because **T&*** is not valid).

Expressions within concept definitions, except the ones in **where** clauses, are not evaluated. "Concepts" is a purely compile-time mechanism.

Basically, a concept expresses the required properties for a type as a series of examples of its use – through a usage patterns. For each operation the minimal requirements on the operand types and (optionally) the minimal requirements on a return type are specified.. The usage patterns used to define concepts here is one of two obvious and roughly equivalent alternative notations [Stroustrup, 2003a]. We prefer it to the alternative (abstract signatures) because we consider it a simpler and more direct notation; see also section ???.

3.1 Applying concepts

The concept **Forward_iterator** defines a predicate – a mapping from the single type **Iter** to **true** or **false**. For a type **X**, **Forward_iterator<X>** may be **true** if **X** has the properties required by the concept. For details of when and how such predicates can be used, see Section 4; for now, consider an example:

```
class Ptr_to_int {
        typedef int value_type;
        Ptr_to_int& operator++();          // ++p
        Pter_to_int operator++(int);       // p++
        int& operator*();                  // *p
        // …
};

bool operator==(const Ptr_to_int&, const Ptr_to_int&);
```

```
bool operator!=(Ptr_to_int, Ptr_to_int);
```

Is **Ptr_to_int** a **Forward_iterator**? It may be because

- it provides the two increment operators
- it provides the dereference operator
- It has the default copy operators
- It has the member typedef **value_type**
- the **int** returned by * has the default copy operations
- the two equality operations **==** and **!=** are defined by the global functions
- these operations have return types that can be used as specified

See Section 4 for more details.

3.2 Composing concepts

Requirements are usually not simple lists of individual attributes; they are partially composed out of other requirements that we have found widely useful and named. We do the same for concepts. For example:

```
concept Assignable<class T, class U = T> {
        T a;
        U b;
        a = b;  // can assign
};
```

In addition to be used in template definitions, a concept can be used to define other concepts:

```
concept Copyable<class T, class U = T>
        where Assignable<T,U> {
        U a;
        T b = a;        // can copy construct
};
```

A concept that can be used as a predicate of a single argument can be used as the type of a concept argument, exactly as it can for a template parameter:

```
concept Trivial_iterator<Copyable Iter> {
        typename Iter::value_type; // value_type names a type
        Iter p;
        Iter& r = ++p;
        const Iter& = p++;
};
```

Here, a **Trivial_iterator<T>** must for starters be **Copyable<T>**. In addition, it must provide the name **value_type** and the increment operators.

```
concept Output_iterator<Copyable Iter> {
        Iter p;
        Value_type Iter::value_type;
        Iter::value_type v;
        *p = v;                 // we can write
        // we can't compare output iterators
};
```

Here, **Output_iterator** adds a requirement on the **value_type**: it must be a **Value_type**. For a **Trivial_iterator**, absolutely any type could be the **value_type** and consequently a template definition could not use any operation on it at all (none is guaranteed to be provided). By requiring a **Value_type**, **Output_iterator** limits the kind of type that can be its **value_type** while ensuring that it can use copy operation.

If more requirements are needed for a concept parameter, we use a **where** clause exactly as for template arguments:

```
concept Input_iterator<Trivial_iterator Iter>
        where Equal_comparable<Iter> {
        Iter p;
        Iter::value_type v = *p;       // we can read
};
```

Alternatively, we can use **&&** to express that a concept is built out of two other concepts:

```
concept Forward_iterator<Input_iterator && Output_iterator Iter>  {
};

concept Bidirectional_iterator<Forward_iterator Iter>  {
        Iter p;
        Iter& r1 = --p;
        const Iter& r2 = p--;
};

concept Random_access_iterator<Bidirectional_iterator Iter> {
        Iter p;
        p = p+1;                        // can add an int
        p = p-2;                        // can subtract an int
        Iter::value_type v = p[3];      // can subscript with an int
};
```

Here **1**, **2**, and **3** are just examples of **int**s used to express a usage pattern. If you like, you can think of an integer literal used in a concept as "any **int**".

We chose to use **where** clauses, rather than inheritance to express that a concept includes the properties of others. This mostly reflects a syntactic preference, but since no issues of overriding is involved, the predicate syntax seems more natural.

3.2.1 General predicates

Predicates that say that a given type is an example of a concept are central to concept checking, but they are not the only predicates involving types. As shown, template functions require certain relations among their argument types. The standard library **find()** is another example:

```
template<Forward_iterator For, Value_type V>
        where EqualComparable<For::value_type,V>
For find(For first, For last, const V& v)
{
        while (first!=last && *first!=v) ++first;
        return first;
}
```

We can define **EqualComparable<class T, class U>** like this:

```
concept EqualComparable<class T, class U = T> {
        T t;
        U u;
        bool b1 = (t==u);
        bool b2 = (t!=u);
};
```

Calling **EqualComparable** a concept could be considered a bit of a misnomer. As defined in the context of template argument checking, a concept is really a predicate on a single type argument. However, it seems a waste to introduce a new keyword for the generalization to more arguments. So, we use the name of the most prominent use, “concept”, for the more general notion, “type predicate”. Also, one way of viewing a type is exactly as a predicate.

Note that any type that can be a template parameter can be used as a concept parameter. This implies that integers can be used as concept arguments. For example:

```
concept Small<class T, int limit>
        where sizeof(T)<=limit
{
};
```

This might be used for selection like this:

```
template<Value_type T> where Small<T,200>
        void f(const X& a) { X aa = a; /* … */ }

template<Value_type T> where !Small<T,200>
        void f(const X& a) { X& aa = new X(a); /* … */ }

template<Value_type T, int N> struct Array { T a[N]; };

void g()
{
        Array<double,100> ad;
        Array<int,10> ai;
        f(ad);  // not small
        f(ai);  // small
}
```

In this example, the function **f()** is overloaded based on the size of its type argument. It does its internal job by allocating temporaries on stack if the type is estimated small, otherwise free store. The overloading is guided by the evaluation of the **Small** predicate (concept). For more details on concept-based overload resolution, see section 6.3.

3.2.2 Use of associated types

Consider again **Forward_iterator**. We specified that its associated **value_type** must be a **Value_type**. That constrains its use to the operations specified by **Value_type**. For example:

```
template<Forward_iterator Iter1,
        Forward_iterator Iter2,
        Forward_iterator Iter3>
void f(Iter1 p1, Iter2 p2, Iter3 p3)
{
        int x1 =  (*p1).m;              // maybe m is an int
        void* x2 = (*p2).m;             // maybe m is a void*
        int x3 = (*p3).m(7);            // maybe m is a member function
                                        // taking a double and returning an int
}
```

At first glance, the definition of **f()** looks fine. However, we apply **.** (dot) to an object of **Iter**'s **value_type**. But dot is not defined for **Value_type**, so clearly an error that's easily caught by the concept check.

It is up to the user of a **Forward_iterator** to specify assumptions about the **value_type**. The obvious place to do that is in function template declarations. For example, we saw

how the writer of **fill()** needed to specify **where EqualComparable<For::value_type,V>** to get complete checking.

Consider a function that traverses sequences and wants to directly read/write a name string and also output the elements. We could of course write a function object for doing the work and apply that, but let's do the work directly:

```
concept Record<class T> {
        T x;
        string& a = x.name;  // can use name as a string (for reading and writing)
        cout << x;           // can use << for some ostream
};

template<Forward_iterator Iter>
        where Record<Iter::value_type>
void f(Iter first, Iter last)
{
        for (; first!=last; ++first) {
                if ((*first).name<=name) {
                        cout<<*first;
                        cout << '\n';
                }
        }
}
```

This **f()** will handle any sequence of any element type with the desired embers (**name** of type **string**) and output operation. Applying **.mem** to a variable of a concept argument is a requirement for a type matching the concept to have a member **mem**. Note that requiring a member **mem** doesn't make any assumptions about **mem**'s relative position in an object. For example, **f()** would work equally well with

```
struct S1 {
        string name;
        // …
};

ostream& operator<<(ostream&, S1);
```

And

```
struct S2 {
        int x,y,z;
        string name;
        // …
};
```

```
ostream& operator<<(ostream&,const S2&);
```

It would not work with

```
struct S1 {
private:
        string name;
        // …
public:
        // …
};
```

Properties of a type required by a concept must be public.

4 Which types match a concept?

We say that a type matches a concept if it meets all of the requirements of the concept. It would be simpler and more correct to say that a type was of a concept (e.g. **int*** is a **Forward_iterator**) in the same way as we say that an object is of a type (e.g. **8800** is an **int**). However, that "is a" would easily be confused with the class hierarchy use of "is a". Others use "models" in much the same way we use "matches".

How does the compiler know that **int*** matches **Forward_iterator** and **int** doesn't? The compiler does the matching when needed; that is the matching of **C<T>** for a concept **C** and a type **T** is done when the predicate is explicitly used in the source code or when **T** is used as the argument to a template requiring a **C**. Obviously the check for a match is done at compile time and needs to be done only once per **C<T>** combination. If **C<T>** is used many times, the compiler can simply use the answer computed the first time.

Sometime we want to explicitly state that we expect **C<T>** to be true. If so, we use an assert:

```
static_assert C<T>;
```

If **C<T>** is false, the compilation fails. Note that we are using the original and proper meaning of an assert: that a predicated must be true or the compilation fails. Therefore, we have not compunction about "reusing" the keyword **static_assert**.

Consider a simple pointer class that can be used as an iterator because it has the required operations with the required semantics:

```
class Ptr_to_int {
        typedef int value_type;
        Ptr_to_int& operator++();          // ++p
```

```
        Pter_to_int operator++(int);        // p++
        int& operator*();                   // *p
        // …
};

bool operator==(const Ptr_to_int&, const Ptr_to_int&);
bool operator!=(Ptr_to_int, Ptr_to_int);
```

We can use Ptr_to_int in the obvious way:

```
const int max = 100;
int a[max];
Ptr_to_int pi(a);
Ptr_to_int pi2(a+100);
fill(pi, pi2, 77);
```

When the compiler sees the call of **fill()** it evaluates **Forward_iterator<Ptr_to_int>** and finds that **Ptr_to_int** has all the operations needed to be a **Forward_iterator**.

Now consider **int***:

```
const int max = 100;
int a[max];
fill(a, a+max,77);
```

Obviously, we want this to compile. We want to accept **int*** as an iterator, but the **Forward_iterator** concept requires the member name **value_type** for the type pointed to and **int*** offers no such name. So, as stated so far, **int*** is not a **Forward_iterator**. We must somehow say that for an **int***, the **value_type** is **int**. That is done by an assert for pointers augmented by a specification of the meaning of **value_type** for a pointers:

```
static_assert template<Value_type T>  Forward_iterator<T*> {
        typedef T* pointer_type;     // auxiliary name for predicate argument
        typedef  T pointer_type::value_type;
};
```

This means that any **T*** is a **Forward_iterator** (the compiler checks that the operations are provided) and that when we need to use **value_type** for a **T*** we use the **typedef** for **T**. As long as this assert appears lexically before the first use of **int*** where a **Forward_iterator** is required, all works as expected.

The syntax for naming the value type for a pointer is a bit awkward. We'd have liked to say:

```
static_assert template<Value_type T>  Forward_iterator<T*> {
        using T*::value_type = T;
```

```
};
```

However, that would combine the new syntax for template aliases with the syntactic innovation of applying **::** to a type expression. However, writing it this way may help show what's really going on: When we are compiling a template definition that requires a **Forward_iterator** and the template argument is a pointer, we use **value_type** as if it was a member of the **T*** meaning **T**.

If desired, a user can provide asserts for a wide variety of concept type combinations. Some users will want to make such assertions to make their intent more explicit. For example, if a **Ptr_to_int** is intended to be used as a **Forward_iterator** a user may like to assert that:

```
static_assert Forward_iterator<Ptr_to_int>;
```

That way, any deficiency of **Ptr_to_int** when – as intended – it is used as a **Forward_iterator** is caught before anyone tries to use it.

However, explicit asserts are not required except where needed to map properties of a type to the requirements of a concept. Consistent use of **static_assert** is unmanageable and complete use is impossible for the current range of generic programming techniques; see Appendix A.

4.1 Negative assertions

We can assert that something is not true. For example:

```
static_assert !Forward_iterator<My_iterator>;
```

This says that the compiler should not (ever) evaluate the predicate **Forward_iterator<My_iterator>** but consider it **false** whenever it occurs. That assert does *not* mean "evaluate the predicate and give an error if it is **true**". The main point of a negative assertion is to give the compiler information that it might have problems figuring out for itself. For example, **My_iterator** might be a type that required complex template instantiation that we specifically don't want done or maybe **My_iterator** does have the syntactic properties of a **Forward_iterator**, but for some semantic reason we don't want it considered a **Forward_iterator** (see 6.6).

Note that by making a negative assertion we can enforce the use of (positive) asserts for a concept. For example:

```
static_assert template<class T> !C<T>;     // in general a T isn't a C
// ...
static_assert C<My_type>;  // but My_type is a C
```

See 6.6.

4.2 Assertion checking

An assert appears at a specific place of a program. However, the complete implementation of a template that it makes an assertion about may be scattered all over the program. For example:

```
template<Value_type T> class Ptr {
        T* p;
        // …
        Ptr(T* pp) : p(pp) { /* … */ }
        void release();
        void rebind(T*);
        T* peek();      // defined in somewhere_else.cpp
        // …
};

template<Value_type T> void Ptr<T>::release() { /* … */ }

static_assert Forward_iterator<Ptr<My_type>>;

template<Value_type T> void Ptr<T>::rebind(T* pp) { /* … */ }
```

We suggest that the assert triggers a check of the class and every member function in scope. Member functions that are not in scope of the assert as not checked until they are used, where the concept checking will be implicitly done.

5 FAQ

One paper can't answer all questions and especially can't answer all questions exhaustively. So here are a few key questions with brief answers:

- What's concepts good for? They allow us to get much better error messages when using templates, and to get those error messages as we compile individual source files, rather than later during linking. They allow us to write much simpler generic code, eliminating most traits, helper functions, and "enable if" workarounds.
- Why aren't constraints classes and archetypes sufficient? Constraints classes (concept classes) are part of the definition of templates, not part of their declaration (interface). A constraints class can detect the lack of an operation early, but can't catch the use of an unspecified operation that happens to be available to a given type. Archetypes (used to address that last problem) are hard to write and require a special effort to use.

- Why don't we just do "conventional type checking" and specify the exact type of every operation? After all that's easy to implement, easy to understand, and "that's what everybody else does". Because we'd loose most of generic programming, because the rules for C++ built-in types are defined in terms of conversions (not of type equality), and because C++ use depends on overloading. If that's what we want, we can use abstract classes.
- Why do we use usage patterns to define concept constraints rather than abstract signatures? Because they are easier to write, easier to read, a terser notation, and require less language invention than the alternative: abstract signatures. (Abstract signatures can be seen as the assembly code for usage patterns; see ???).
- How can I be sure that my templates are checked at compile time rather then later at instantiation time? Make sure every template type parameter has a concept.
- Why don't we specify the concept as part of a type declaration? Because then we end up with the rigid hierarchical structures of object-oriented programming. If that's what we want, we can use inheritance.
- Why don't we require explicit "matching" (modeling) declarations to associate types with concepts? Because programs with sufficient assertions of this kind becomes unmanageable.
- Is the matching simply "structural conformance"? No. By using **static_assert**s and **where** clauses we manipulate the type system based on named predicates.
- Does concept use involve overheads compared to ordinary class parameters? No.
- Does concept use limit flexibility compared to ordinary class parameters? Not that we know of and overloading based on concept opens new possibilities.
- Are concepts hard to use? Every new feature is initially seen as complicated, but concepts are significantly easier to use than the current techniques used to compensate for proper typing of template type arguments.
- Is there anything useful that we can express with **<class T>** that we can't express with **<C T>** where **C** is some concept? Yes, a completely unconstrained type, such as the type **T** pointed to by **T***. Even a **void*** can't point to a function.
- Can we eliminate **<class T>**? In the abstract, maybe. In reality, no. Many template meta-programming techniques seem to fundamentally depend on unconstrained types parameters (see appendix D). Also, there will always be pre-concept C++ code around.

6 Technicalities

This section records discussions of more technical aspects of the concept design presented in this paper. It considers interoperations with existing template mechanism and interaction with overload resolution, partial specializations, and name resolution.

6.1 Type argument syntax

If you don't use concepts, templates work exactly as before. You can use any combination of concepts and old-style template arguments. The intent is that the only

difference should be when and where an error is found (and the quality of the resulting error message). We only approximate this ideal; see Section 7.

The two uses of concepts to specify an argument

```
template<My_concept T>
```

and

```
template<class T> where My_concept<T>
```

are completely interchangeable.

A **where** clause starts with the keyword **where** followed by a (non-empty) list of concept predicates combined using **&&**, ||, and **!**. For example:

```
concept Forward_iterator<class Iter>
        where Trivial_iterator<Iter> && Equality_comparable<Iter>
```

is equivalent to

```
concept Forward_iterator<Trivial_iterator Iter>
        where Equality_comparable<Iter>
```

The &&, ||, and ! operators can also be used in the concept parameter so this is also equivalent to

```
concept Forward_iterator<Trivial_iterator && Equality_comparable  Iter>
```

6.2 Syntax

The concept syntax is closely based on the template syntax. It introduces two new keywords, **concept** and **where**. In addition, it "hijacks" the closely related keyword **static_assert**. The angle brackets are retained to emphasize the compile-time nature of the predicates and their relation to templates.

6.3 Overloading based on concepts

We can overload based on concepts. Consider a set of templates and a use. Each argument in the use is matched against the concepts of all the templates

1. If the arguments in the use match none of the templates obviously, the use fails
2. If the arguments in the use match exactly one of the templates, that template is used
3. If the arguments match two (or more) of the templates, we try to find a best match. To be a best match a template must be such that

a. Each argument must be a better match or as good a match as the corresponding match for that argument for all other candidate templates
b. At least one match must be better than the corresponding match for that argument for all other argument templates

An argument is a better match for one concept than another if it is more constrained than the other; that is, if the predicates of the other concept is a subset of the predicates of the first. This set of rules is a (radically) simplified version of the rules for function overloading.

Note that we only consider complete named predicates. We do not "look into" the definition of a predicate when comparing predicates. For example:

```
concept Assignable<class T> {
        T a;
        T b;
        a = b;
};

concept Copy_constructible<class T> {
        T a;
        T b = a;
};

concept Copyable1<class T> {
        T a;
        T b = a;
        a = b;
};

concept Copyable2<Copy_constructible T> {
        T a;
        T b;
        a = b;
};

concept Copyable3<class T>
        where Assignable<T> && Copy_constructible<T> {
};
```

Here, **Copyable3** is both **Assignable** and **Copy_constructible**. However, **Copyable2** is **Copy_constructible** but not **Assignable**, and **Copyable1** is neither **Copy_constructible** nor **Assignable**. The fact that any type that matches one of the three **Copyable**s also matches the other two is not relevant to overload resolution as it is "just an implementation detail".

6.4 Concepts and specialization

Some functions, like **advance()**, work on series of refined concepts. Typical implementations are based on uses of traits for type dispatching. Concepts can be used to simplify and express directly the implementation of such functions. For example

```
template<Forward_iterator Iter>             // #1
void advance(Iter& p, int n)
{
        while (n--)
        ++p;
}

template<Random_access_iterator Iter>   // #2
void advance(Iter& p, int n)
{
        p += n;
}
```

Definition #2 is considered a specialization of definition #1, because the concept **Random_access_iterator** has been defined as a refinement of **Forward_iterator**; that is, **Forward_iterator<T>** is true whenever **Random_access_iterator<T>** is true. Consider the following case

```
//  Both advance()templates are in scope
template<Forward_iterator Iter>
void mumble(Iter p, int n)
{
        // …
        advance(p, n / 2);
        // …
}

int ary[] = { 904, 47, 364, 652, 589, 5, 35, 124 };
mumble(ary, 4);
```

The definition of **mumble()** passes concept-checking because there is a declaration of **advance()** in scope (#1 that takes **Forward_iterator** or "higher"). Furthermore, because the version #2 of **advance()** is a specialization of #1, it is selected when **mumble()** is instantiated with **int***. That way, the use of concepts does not automatically imply performance pessimization compared to templates not using concepts: the most specialized (and often the most efficient) version is used. A way to think of a specialization is as a definition that happens to provide a better or more efficient implementation for template arguments with known refined concepts. In particular, specializations do not participate in concept-based overload resolution (Section 6.3).

Only after concept-based resolution takes place, are they selected to provide implementations for the most abstract templates they are specializing.

So, exactly how do we know that a template definition **L** is a specialization of template definition **K**? First, both must have the same number of template-parameters. Second, we can write their definitions in the more verbose style

> **template<class T /*… */ > where** … apply predicates to parameters ...

This makes it clear that the corresponding parts of the declarations without the **where**-clauses are undistinguishable (modulo the usual template parameter renaming rule). In the **advance()** case, we would have

```
template<class Iter> where Forward_iterator<Iter>              // #1
void advance(Iter&, int);

template<class Iter> where Random_access_iterator<Iter>       // #2
void advance(Iter&, int);
```

If we ignore the **where**-clause parts, both declarations declare the same template. Third, in the rewritten form, the predicate of **L** must imply the predicate of **K**.

This rule is simple, but we believe it suffices to guarantee that specialized templates will be used where available, therefore reducing the risk of potential pessimizations due to concept uses. The effects of this rule are very similar to "function template partial specialization" (for which we do not have syntax in current C++) which would require use of traits and forwarding:

```
template<class iterator_category_tag>  struct advance_helper;
// specialize for forward iterator
template<>
struct advance_helper<forward_iterator_tag> {
        template<class Forward>
        static void do_it(Forward& p, int n) { /* … */ }
};

// specialize for random access iterator
template<>
struct advance_helper<random_access_iterator_tag> {
        template<class Rand>
        static void do_it(Rand& p, int n) { /* … */ }
};

// This function template forwards the real work to the specific
// implementation based on the category of the iterator.
template<class Iter>
```

```
void advance(Iter& p, int n)
{
        typedef  typename iterator<Iter>::category category;
        advance_helper<category>::do_it(p, n);
}
```

Notice that in this form, it is obvious that only the implementation of **advance()** changes, not its type. The rule for function template specialization based on concept guarantees that same type-safety. Note also how much simpler the concepts version is.

6.5 Name resolution in templates using concepts

In a template definition where template-parameters are defined with concepts, we cannot use operations on parameters that are not supported by the relevant concepts. If strictly applied, that rule would render use of concepts painful, as every template would now have to list all references to dependent names in a separate concept.

A template definition can reference a dependent name not explicitly listed in the relevant concept if there is a corresponding declaration in scope and the concept can be satisfied. For example, consider the following fragment

```
template<Forward_iterator Iter>void advance(Iter&, int);

template<Bidirectional_iterator Iter> void bar(Iter first, Iter last)
{
        advance(first, 2);              // ok
}
```

This is valid even though **advance()** is not listed directly in the **Bidirectional_iterator** properties. The reason is that, there is a declaration in scope and its concept is matched by any **Bidirectional_iterator** (the **Forward_iterator** is a predicate that is part of the **Bidirectional_iterator** predicate). In other words, this is the instance of contra-variance in arguments that is type-safe: **bar()** can be called only with something that is a **Bidirectional_iterator** and every **Bidirectional_iterator** is a **Forward_iterator**. On the other hand, had **advance()** been defined to require a **Random_access_iterator** then the call would have failed concept-checking because **Random_access_iterator** requires more than **Bidirectional_iterator**. That is one kind of reference to dependent names in template definitions.

When a dependent name used in a template definition appears explicitly in the properties of the type's concept, and happen to be also declared in the scope containing the template definition, we have a conflict that can be resolved in three different ways:

1. prefer the name listed in the concepts; or
2. prefer the name declared in scope; or

3. consider the use ambiguous or merge the declarations and do overload resolution.

Arguments can be constructed for each case. We resolve this conflict using (1). The rationale is that for a template parameter declared with a concept, the user has to supply all required operations and the expectation is that those operations will be used in the instantiation as opposed to whatever happen to be around at the definition site. On the other hand, the author of the template definition may expect his definition be used as opposed to whatever happens to be supplied by user at instantiation site. However, the resolution (1) support the notion of "points of customization", whereby a template author can advertise in the template interface what operations are expected, so that a user can supply them. Thus, that resolution provides control over the "argument dependent name lookup" rule. For example:

```
concept Sorting_iterator<Random_access_iterator Iter> {
        Iter::value_type x, y;
        swap(x, y);                     // swap() must be found at use point
};

namespace lib {
        template<Value_type V> void swap(V&, V&);
        template<Sorting_iterator Iter> void bubble_sort(Iter first, Iter last)
        {
                Iter p, q;
                // …
                swap(*p, *q);           // use swap() from use point.
                // …
        }

        template<Random_access_iterator Iter>
        void quicksort(Iter first, Iter last)
        {
                Iter p, q;
                // …
                swap(*p, *q); // use lib::swap(), no ADL.
        }
}
```

In the **lib::bubble_sort()** case, the use of **swap()** is resolved to the name found at the use point, in particular no argument dependent name lookup is performed. For the **lib::quicksort()** example, the call to **swap()** is resolved to the one in scope at the point of template definition; no ADL is performed.

6.6 Semantics differences

There are situations where concepts differ only in semantics and not in syntactic requirements. One could argue that such a situation should not happen; that is, "if you

use the same syntax for two different things you are asking for trouble". However, such situations do occasionally happen for good reasons. Fortunately, we can handle such syntactically ambiguous situations simply through explicit assertions. Consider:

```
static_assert !C<T>;
```

This means that the compiler should never ever try to check whether type **T** matches concept **C**. Rather, the compiler must simply assume that the type **T** does not match concept **C**; that is, **C<T>** is **false** (whatever an actual check of the definition of **C** might say). That is a "trust the programmer" part of the concept system. Assume we have a concept that says that every type is nice unless explicitly asserted otherwise:

```
concept Nice<typename T> { };      // everybody is nice

struct Grumpy { };
static_assert !Nice<Grumpy>;       // except Grumpy.

template<Nice T>
void f(T t);                  // Nice people can  call f().

int main()
{
        f(42);                // OK – int is Nice, by default
        f(Grumpy());          // error – no Grumpy is Nice.
}
```

This is a typical situation where concepts differ not it the operations they provide, but in the way users are supposed to use those operations. For example, the syntactic requirements of an input iterator and a non-mutating forward iterator are identical. However, the semantics of a genuine input iterator is profoundly different from a forward iterator. In particular, we cannot make a purely syntactic distinction between a **list<int>::const_iterator** and an **istream_iterator<int>**. The difference is semantic: the former can be used in multi-pass algorithms whereas the latter cannot. In other words, an input operator and a forward iterator differs (only) in the protocol required for their use.

```
concept Input_iterator<Trivial_iterator Iter> {
        // list of input iterator properties
};

concept Forward_iterator<Input_iterator Iter> {
        // list of forward iterators properties
};

// we find that all non-mutating (const) Forward_iterators are also Input_iterators
// but some shouldn't be
```

```
// some of istream iterators are really special:
static_assert
        template<Value_type T> !Forward_iterator<istream_iterator<T>>;
```

Note that implementers of genuine input iterators have to explicitly state that their iterators do no support multi-pass algorithms (are not Forward iterators) – just as they have to do in the today uses of the standard library (there is no free lunch). However, implementers of iterators that support the "multipass protocol" – the vast majority don't have to say anything special.

6.7 Usage patterns

The definition of a concept is expressed in terms of usage patterns, showing how a type that matches the concept must be usable. In general the declarations, statements, and expressions used to express these usage patterns are simply ordinary constructs with the same syntax and semantics as ever. However, the constructs are not meant to be executed so a sequence of statement is not meant to make sense.

We rely on conventions to introduce names and values into a concept definition:

- **T x;** where **T** is a concept parameter type introduces **x** as a variable of type **T**. Note that we do not apply any of the initialization rules. This use is much as the way we introduce a function parameter. A mentioned in Section ???, we might prefer a different syntax for this.
- **C T;** introduces **T** as the name of a type of concept **C**, just like in a template parameter declaration.
- **T::m x;** If **T** is a concept parameter and **T::m** is a type then this introduces a variable **x** without applying initialization rules, just as in **T x**.
- **42** introduces a value of type **int** (and equivalently for other integer types). Any other **int** literal can be use equivalently.
- **3.14** introduces a floating-point value of type **double** (and equivalently for other floating-point types). Any other **double** literal can be used equivalently.
- **T::m** introduces a member **m** of the type **T**. Note that when **T** is a concept argument, the actual definition of **m** can be in the concept, rather in the type. As ever, we have to use the prefix **typename** if m is to be a type.

In general, every expression is taken as just an example of its type. For example, **f(1)<2** is not evaluated yielding a **bool**; rather, it is checked that a function **f()** that can accept an **int** argument and yield a result that can be compared to an **int**. If we want evaluation, we use a **where** clause.

7 Implementation model

The big question is

> "if a call of a template function compiles and (separately) the definition of that template function compiles, can the program fail to link?"

We must simultaneously try to avoid three problems:

1. The call and the definition both compile, but the program don't link (because of an instantiation failure)
2. The call or definition fails to compile, but the program would have compiled and linked had concepts not been used.
3. The call and the definition both compile, but the program has a different meaning from what it would have had using templates without concepts.

It is impossible to avoid all three problems simultaneously. If we add the constraint that the specification of concepts should be relatively simple we have an interesting set of tradeoffs.

We choose to solve (1) and (2) at the cost of accepting (3). Our rationale is that "some programs ought not to work and if they do then they should not work in such surprising ways". That is, even though (3) is a problem for some users and potentially for compiler implementers, there can also be benefits for both users and implementers.

7.1 Abstract signatures

In [Stroustrup, 2003a] we introduced the notion of "abstract signatures" as an alternative notation for concept requirements ([Siek, 2005] calls those "pseudo signatures"). One way of understanding concepts (however expressed) is to consider how a compiler might convert them to the most primitive form of abstract signatures. Consider

```
concept LessThanComparable<class T> {
        T a, b;
        bool b1 = a<b;          // the result of a<b can be used as a bool
};
```

That is, two **T**s can be compared using **<** and the result of such a comparison can be converted to (be used as) a **bool**. As discussed in [Stroustrup, 2003a], this does not mean that there must exist a function **bool T::operator<(const T&);** rather, one of the many ways of implementing **a<b** must have been used (a non-member <, a built-in <, a < involving conversions, etc.). We can express the requirement by introducing three additional types **A1**, **A2**, and **R**:

```
T -> A1
T -> A2
Operator<(A1,A2) -> R
R -> bool
```

The first three abstract signatures say that there must exist types **A1** and **A2** for which there exists a **<** operation returning some type **R**; **T** must be (implicitly) convertible to **A1** and **A2**. Obviously, in the simplest case, both **A1** and **A2** are **T**. The fourth abstract signature says that the type **R** must be convertible to **bool.** **A1** and **A2** are potential intermediate types used to invoke **<** and **R** is a potential intermediate type for the result. The arrow, **->**, means "converts to".

As ever, only the specified operations must be used for a type. In particular, we cannot apply any operations not mentioned for **A1, A2**, and **R**. We consider such simple abstract signatures an "assembly code" for usage patterns and assume that they, or an equivalent abstract syntax representation, will be used internally to compilers.

In C++ just about every operation involves conversions that can be described using such abstract signatures. Most operations potentially require two auxiliary types (the operand type and the result type). The complexity of explicitly introducing these auxiliary types is a major reason for using usage patterns to define concepts.

7.2 An interesting example

[Siek, 2005] presents this example:

```
template<LessThanComparable T>
bool foo(T x, T y)
{
        return x<y && random()%3;
}
```

Assume that **LessThanComparable** is defined in the minimal way:

```
concept LessThanComparable<class T> {
        T a, b;
        bool b1 = a<b;          // the result of a<b can be used as a bool
};
```

The writer of **foo()** probably assumed that **x<y** yields a **bool** and that the built-in **&&** is used on that **bool** and the result of the **int** result of **random()%3**. That is, the resolution is most likely assumed to be

```
return bool(x<y) && random()%3;
```

However, consider this:

```
class Y {
        operator bool();
        bool operator&&(int);
};
```

```
class X {
        Y operator<(X);
};

X x1, x2;
bool b = foo(x1,x2);
```

Without use of concepts, the following would happen: **Y**'s **&&** is an exact match and will be chosen over the conversion of **x<y** to **bool**. That is, the resolution would be

```
return Y::operator&&(x<y, random()%3);
```

However, we did use a concept: **foo()** was defined using **LessThanComparable** and the only operations specified for (and therefore the only operations allowed on) the argument type **T** (in our example **X**) was to compare it and to use its result as a **bool**. Nothing further was said about or can be assumed about the result of **x<y**. Thus, the instantiation **foo<X>** may not use **Y::operator&&()**; it must use **bool(x<y)**. This resolution differs from [Siek, 2005] Section 3.2-3 where the example was used to argue for the use of exact types, rather than "convertible".

So, this example is a case of "problem (3)" above; that is, a program that would compile using templates without concepts, but differently when using concepts. However, we consider this an example that in an ideal world shouldn't have worked in the first place because dramatically different resolutions (some surprising) were used for different argument types.

7.3 The three-operand problem

The example above is an instance of a general problem where two operators and three operands are involved. In [Stroustrup, 2003b], this problem was discussed in terms of how to handle intermediate results, but we can now state a solution more simply. Consider

```
template<Arithmetic T> void f(T x, T y, T z)
{
        // …
        x*y+z;
        // …
}
```

What should the concept **Arithmetic** look like? Our **Arithmetic** should of course be able to handle the built-in arithmetic types and also other types where the type of **x*y** is different from that of the type of **x**.

Let's first define **Arithmetic** like this:

```
concept Arithmetic <Value_type T> {
        T a, b;
        a = a+b;
        a = a-b;
        a = a*b;
        a = a/b;
};
```

To describe this in terms of abstract signatures requires 12 auxiliary types:

```
T -> A1
T -> A11
Operator+(A1,A11)->R1
R1->T

T -> A2
T -> A21
Operator-(A2,A21)->R2
R2->T

T -> A3
T -> A31
Operator*(A3,A31)->R3
R3->T

T -> A4
T -> A41
Operator/(A4,A41)->R4
R4->T
```

This is fully general and makes no assumptions about the types used to hold intermediate results.

7.4 The “same type” problem

How would we express the more constrained concept where intermediate results have to be of the same type as the arguments? In other words, how do we express the notion of “same type”? Basically, there is no simple way in C++ to express the idea that two expressions have the same type. Equality involves possible conversions. Initialization involves possible conversions and the rvalue/lvalue distinction. What we want is something like this:

```
concept Arithmetic <Value_type T> {
        T a, b;
        a = a+b;
```

```
        a = a-b;
        a = a*b;
        a = a/b;
        same_type(a+b,a-b);
        same_type(a-b,a*b);
        same_type(a*b,a/b);
};
```

We could provide **same_type()** as a primitive operation, possibly related to **decltype**. However, this would probably do:

```
template<class T, class U> concept Same_type {
        T t;
        U u;
        T* r1 = &u;
        U* r2 = &t;
}

template<class T, class U> void same_type(T x, U y)
        where Same_type<T,U>
{ }
```

On the other hand, this might be seen as the kind of "clever" template programming, we'd like to minimize. Obviously, there is an art to writing concepts that we still have to develop. Like the art of writing templates, we can't expect to know it all before we have implementations to work on, but as in the case of templates, we must provide general mechanisms to allow programmers to express solutions beyond our immediate imagination.

An alternative solution is to introduce the intermediate type explicitly:

```
concept Arithmetic <Value_type T, Value_type W = T> {
        T a, b;
        W(a+b);
        W(a-b);
        W(a*b);
        W(a/b);
};
```

We could use this last Arithmetic like this:

```
template<Arithmetic T>
void calc1(const vector<T>& a1, const<vector<T>& a2, vector<T>& res);

template<class T, class W> where Arithmetic<T,W>
```

```
void calc2(const vector<T>& a1, const<vector<T>& a2, vector<W>& res);
```

7.5 The "arrow" problem

The standard iterator requirements include the requirement that **p->m** should be valid if **(*p).m** is valid. This is curious because it is a conditional requirement: **p** may be a pointer to a type that doesn't have any members, such as an **int** for **int***. There is no simple piece of code than we can write to determine whether an iterator type **Iter** actually points to something with a member. We can of course easily check whether a type supports a particular member, but that's a different problem.

Consider a trivial template definition:

```
template<Forward_iterator Iter> void f(Iter p)
{
        int x = p->m;
}
```

There are basically one right way and three wrong ways to call **f()**:

```
struct S1 { int m; };          // what f() likes
struct S2 { void* m; };        // f() doesn't like the type of m
struct S3 { int n; };          // no member m for f() to use
S1* p1;
S2* p2;
S3* p3;
int* p;

f(p1);          // ok
f(p2);          // type error: can't convert void* to int
f(p3);          // type error: S3 has no member m
f(p);           // type error: you can't use -> in an int*
```

We would like to make these type errors – detected only during instantiation – into concept errors – caught at the point of call and/or the point of **f()**'s definition. A complete solution is to specify **f()**'s requirement on the **value_type**:

- The **value_type** must be a class
- The class must have a member called **m**
- the member **m** must be convertible to **int**

In other words:

```
concept Has_m<class Ptr> {
        Ptr p;
```

```
        int i = p->m;
};

template<Forward_iterator Iter>
        where Has_m<Iter>
void f(Iter p)
{
        int x = p->m;
}
```

Unfortunately, the mechanism of usage patterns offers no way of just stating that -> is required. There couldn't be because C++ offers no general syntactically valid way of using -> without a member name. Thus, if we want an **Arrow** predicate, we must provide it through simple, but special-purpose compiler magic. We propose

```
concept Arrow<class P> {
        P p;
        // we can apply -> to p if we can apply . to *p
};

concept Forward_iterator<Input_iterator Iter>
        where Arrow<Iter>
{
        // ...
};
```

However, before adopting this special-purpose "intrinsic predicates" we should consider if a modification of the iterator requirements would be a better choice. We use **Arrow** rather than **Has_arrow** because of its curious conditional nature.

7.6 The function call syntax problem

Our concepts definitions do not care whether a required function is implemented as a built-in, a member function, or as a non-member function. Ideally, we would provide the same freedom when specifying function calls: **p->f()** or **f(p)**? Why should we care? We care because when we make the distinction between **p->f()** and **f(p)**, we either double the number of concepts or force the users of a concept to conform to the concept's preferred notation. Either way, the resulting templates will be less flexible and less easy to use. Consider a concept that requires the use of both calling syntaxes:

```
concept FG<class T> {
        T a;
        f(a);
        a.g();
};
```

We also happen to have a class that similarly uses both calling syntaxes, but "unfortunately" in exactly the opposite cases:

```
class X {
        virtual void f();
};

void g(const X&);
```

If we can handle **FG<X>**, we can handle any call syntax mappings, and more. As for member types, we can specify the mapping in an explicit assert of the relation between the concept and the type:

```
static_assert FG<X> {
        void f(X& a) { a.f(); }         // non-member to member mapping
        void X::g() { g(*this); }       // member to non-member mapping
};
```

Consider

```
X x;
// …
template<FG T> void ff(T a)
{
        f(a);
        a.g();
}
// …
ff(x);
```

When we instantiate **ff()** for an **X**, we look at **f(a)**. That will call the **f(X&)** defined in the assert, which in turn will call **X::f()** for **a**. Similarly, **a.g()** will call the "**X::g()**" defined in the assert, which in turn will call **g(a)**. Note that there is absolutely nothing "abstract" about the functions and types defined in an assert; they are simply ordinary definitions in a scope that encloses the argument type's scope.

Why is this better than simply defining an extra class to map conventions and let the programmer use that? For example:

```
struct XX {     // map X to FG's requirements
        X* p;
        XX(X& a) :p(&a) { }
        void g() { g(p); }                      // member to non-member notation
        static void f(XX& pp) { pp.p->f(); }// non-member to member notation
};
```

Apart from the possible performance implications, the problem is that we now have to explicitly mention **XX** every time an **X** is passed to something requiring an **FG**. Consider again:

```
X x;

template<FG T> void ff(T a)
{
        f(a);
        a.g();
}

ff(x);
```

As written, this is clearly wrong. We can add the **static_assert FG<X>** or replace **ff(x)** with **ff(XX(x))**. At best, the latter is inelegant and the exact ways of mapping the conversions will vary from class to class.

8 Standard-library concepts

One of the first and most important tasks, given concepts, is to provide concepts for the standard library. That will not only give an easier to use and better specified library, but also provide a set of standard-library concepts for programmers to learn from and use. The aim is to remove as much as possible from the requirements tables and present them as concepts instead. The ideal would be to eliminate the requirements tables altogether.

The standard library defines and uses several relational notions: comparisons with equality, relative orderings:

```
concept Equality_comparable<class T, class U = T> {
        T a;
        U b;
        bool eq = (a == b);
        bool neq = (a!=b)
};

 concept Less_comparable<class T, class U = T> {
        T a;
        U b;
        bool lt = (a < b);
 };

concept Less_equal_comparable<class T, class U = T>
        where Less_comparable<T, U> {
```

```
        T a;
        U b;
        bool le = (a <= b);
};

concept Greater_comparable<class T, class U = T> {
        T a;
        U b;
        bool gt = (a > b);
};

concept Greater_equal_comparable<class T, class U = T>
        where Greater_comparable<T, U> {
        T a;
        U b;
        Bool ge = (a >= b);
};

concept Total_order<class T, class U = T>
        where Less_comparable<T, U>
        && Less_equal_comparable<T, U>
        && Greater_comparable<T, U>
        && Greater_equal_comparable<T, U>
        && Eeual_comparable<T, U>  { };
```

Total_order is not directly required by the standard library, but is a property provided for all standard sequences, provided the sequence's **value_type** is also a **Total_order**.
The standard containers use a fairly elaborate notion of copyable object types: value types are assumed to support the operation of explicitly calling their destructors, taking the address of an object of such type is assumed to yield an expression convertible to plain pointer types:

```
concept Copy_constructible<class T> {
        T t;
        const T u;
        T(t);           // direct-initialization from plain T
        T(u);           // direct-initialization from const T
        t.~T();         // destructible
        T* p = &t;      // addressable
        const T* q = &u;
};
```

Note that the "core language" notion of copy-constructible does not involve, in itself, the ability to take the address of an object; only the container requirements make it so.

Therefore it would have made sense to take that property out of **Copy_constructible** and make it a separate, independent concept **Addressable**:

```
concept Addressable<class T> {
        T t;
        const T u;
        T* p  = &t;             // address is convertible to T*
        const T* q = &u;        // address is convertible to const T*
};
```

In some cases, the standard library requires some types to be default constructible, or assignable.

```
concept Default_constructible<class T> {
        T();
};

concept Assignable<class T> {
        T t;
        T u;
        const T v;
        t = u;          // can assign both a T
        t = v;          // and const T
};

 concept Value_type<Copy_constructible T>
        where Default_construtible<T>
           && Assignable<T> {
};

concept Trivial_iterator<Arrow Iter>
        where Copyable<Iter> {
        typename Iter::value_type;
        typename Iter::difference_type;
        // not required but present  in the iterator<> traits:
        typename Iter::reference;
        typename Iter::pointer;
 };

concept Input_iterator<Trivial_iterator Iter>
        where Equality_comparable<Iter> {
        Iter  p, q;             // variable p and q
        Iter& i = (p = q);      // must be assignable
                                // the result must be usable as an Iter&
        Iter::value_type v = *p;    // dereferencing converts to value_type
        Iter& r1 = ++p;
```

```
        p++;                    // we don't know the result type of post-increment
        Iter::value_type v2 = *p++;
};
```

The **Input_iterator** requirements – as mandated by the standard library -- may appear "overly restrictive" as they rule out pointers to non-copyable types (e.g. **ofstream***) from being considered input iterators (let alone random access iterators).

```
concept Output_iterator<Trivial_iterator Iter> {
        // not equal comparable in standard
        // The standard requires that both value_type and difference_type
        // be defined as void, for iterator<>.  Was that intentional?
        Iter p;
        Iter::value_type t;
        *p = t;    // we don't know the type of the result
        Iter& q3 = ++p;
        const Iter& q4 = p++;
        *p++ = t;
 };

concept Forward_iterator<Input_iterator Iter>
        where Output_iterator<Iter>
          && Default_constructible<Iter>
           && Assignable<Iter> {
        Iter p, q;
        Iter& r = (p = q);
        Iter::value_type& t = *p;
        Iter& q2 = ++p;
          const Iter& q3 = p++;
          Iter::value_type& t2 = *p++;
 };
```

The "Forward iterator" concept as expressed above actually stated the assumptions on mutating iterators

```
concept Bidirectional_iterator<Forward_iterator Iter> {
        Iter p;
        Iter& q = --p;
        const Iter& r = p--;
        Iter::value_type t = *p--;
};

concept Random_access_iterator<Bidirectional_iterator Iter>
        where Less_equal_comparable<Iter>
          && Greater_equal_comparable<Iter> {
        Integer_type Iter::difference_type;          // Integer_type is a concept
```

```
        Iter::difference_type n;
        Iter p;
        Iter& q1 = (p += n);
        Iter q2 = p + n;
        Iter q3 = n + p;
        Iter& q4 = (p -= n);
        Iter q5 = p - n;
        Iter::difference_type m = p - q1;
        Iter::value_type t = p[n];
 };
```

The **Random_access_iterator** requirements impose mutable **value_type**, as inherited from the requirements for **Forward_iterator**. In fact, that "mutability" requirement is a property of the associated **value_type** and should probably be taken out and stated separately for those algorithms that need it.

```
static_assert template<Value_type T> Random_access_iterator<T*> {
        typedef T* Iter;
        using Iter::value_type = T;
        using Iter::difference_type = ptrdiff_t;
        using Iter::reference = T&;
         using Iter::pointer = T*;
};

 static_assert template<Value_type T> Random_access_iterator<const T*> {
        typedef const T* Iter;
        using Iter::value_type = const T;
        using Iter::difference_type = ptrdiff_t;
        using Iter::reference = const T&;
        using Iter::pointer = const T*;
 };
```

While the standard library requirements state that the associated **value_type** of a **const T*** should be **T**, it is debatable whether that is consistent with the fact that the associated **reference** is **const T&**.

Standard containers are assumed to provide iterators that have at least the "forward iterator" properties:

```
concept Container<class C>
        where Copy_constructible<C> && Assignable<C> {
        Value_type C::value_type;  // needs value_type that matches Value_type
        Reference C::reference;       // must be an "lvalue type"
        Reference C::const_reference; // must be a non-modifible "lvalue type"
        Forward_iterator C::iterator;
        Signed_integral_type C::difference_type
```

```
            where Same<difference_type, iterator::difference_type>;
        Unsigned_integral_type C::size_type;

        C c1;
        const C c2;
        C::iterator p1 = c1.begin();
        C::iterator q1 = c1.end();
        C::const_iterator p2 = c2.begin();
        C::const_iterator q2 = c2.end();
        c1.swap(c2);
        C::size_type s1 = c1.size();
        C::size_type s2 = c1.max_type();
        bool b1 = c1.empty();
};
```

Clearly the specification of reference and const_reference needs work. It is unclear whether we should allow for "proxy references" (class that overload the dot operator). Several relational operators on containers's type are lifted to the containers themselves, whenever they are available:

```
static_assert template<Container C>
        where Less_comparable<C::value_type> Less_comparable<C>;
// and so on for other relational notions.
```

Containers that provide bidirectional iterators can be reversed:

```
concept Reversible_container<Container C> {
        Bidirectional_iterator C::reverse_iterator;
        Bidirectional_iterator C::const_reverse_iterator;
        C c1;
        const C c2;

        C::reverse_iterator r1b = c1.rbegin();
        C::reverse_iterator r1e = c1.rend();
        C::const_reverse_iterator r2b = c2.rbegin();
        C::const_reverse_iterator r2e = c2.rend();
        C::iterator i;          // variable i
        C::reverse_iterator ri(i);      // can convert from C::iterator
        C::const_reverse_iterator cri(i);
};
```

The notion of **Sequence** refines that of **Container**, providing several construction (e.g. "fill" constructors and "range" constructors).

```
concept Sequence<Container C> {
        C::size_type n;
```

```
        C::value_type t;
        C::iterator p;
        C::Input_iterator Iter;
        Iter i, j;
        C(n, t);                    // fill the container with n copies of t
        C c(i, j);                  // construct from the range [i, j)
        C::iterator q = c.insert(p, t);
        c.insert(p, n, t);
        c.insert(p, i, j);
        C::iterator r = c.erase(p);
        C::iterator s = c.erase(p, q);
        c.clear();
};
```

Notice that although the STL provides a **Sequence** view out of associative containers, they differ from sequences in many ways; one of them being that they do not support "fill" constructors.

9 Future work

This paper presents a design for concepts. Future work should aim at expressing as much as possible of the standard library requirements tables in terms of concepts, rephrase warts (e.g. "do the right thing" rule), etc. Given the ability to overload and specialize functions on concepts, one should try to rephrase functions like **sort()** or **find()** in terms of the properties of their iterators arguments. For example, we should be able to specify **find()** to effectively and efficiently supports iterators from ordered containers and that **list::sort()** no longer need to be a member. This will improve both the concept design and the standard library. For example, we expect many (most?) traits classes will become redundant.

It will be important to try out concepts on an application domain that is not the STL, or STL derived. We plan to use classical math, in particular algebra.

One important side-effect of this work with concepts will be a better understanding of the programming techniques required to use them well.

Finally we need a more formal specification of concepts and text suitable for the standard.

10 Summary

The design presented here is not assumed to be perfect; it is a best effort to address competing concerns between improved type checking, effective programming styles, and implementation complexity. We hope for further improvements.

Our major conclusions are

- We can achieve perfect separate checking based on typed template arguments (using concepts) and predicates specifying requirements on combinations of template arguments.
- Haskell-style schemes based on same-type checking (rather than conversions) and overriding (type hierarchies, rather than overloading) are infeasible for C++
- Concept/type asserts ("models statements") are necessary (for flexibility) and must be optional (not to over constrain and complicate code)

The main points of the design is

- Perfect separate checking of template uses and definitions
- Near perfect backwards compatibility, incl. integration of "old" template arguments with "new" type checked template arguments (using concepts)
- No requirements to impose hierarchical order on template argument types
- No restrictions on conversions or overloading compared to existing code
- No performance penalty in time or space
- New code can be simpler than old code not using concepts (through selection based on concepts and elimination of traits and helper classes)
- Concepts are relatively easy to define and many can be defined once as part of a library
- The STL can be described using concepts (and in places improved)

11 Acknowledgements

The work leading to concepts goes back to the earliest days of templates [Stroustrup, 1994], but builds most directly on ideas of Alex Stepanov. Matt Austern, Jeremy Siek Jaakko Järvi, and many others also contributed. Much of the work on revision 1 was prompted by questions, comments, and examples at the Lillehammer standards meeting, notably by David Abrahams, Doug Gregor, and Jeremy Siek.

12 References

- [Siek, 2005] J. Siek, et al: *Concepts for C++0x*. C++ standard committee. Paper N1758. March 2005.
- [Stroustrup, 1994] B. Stroustrup: The Design and Evolution of C++. Addison-Wesley.
- [Stroustrup,2003a] B. Stroustrup: "Concept checking - A more abstract complement to type checking". C++ standard committee. Paper N1510. 2003.
- [Stroustrup,2003b] B. Stroustrup and G. Dos Reis: "Concepts - Design choices for template argument checking" C++ standard committee. Paper N1522. 2003.
- [Stroustrup,2003c] B. Stroustrup and G. Dos Reis: "Concepts – syntax and composition" C++ standard committee. Paper N1536. 2003.

Appendix A: Why asserts must be optional

Why don't we require that a user always explicitly state that a type is of a concept? That is, why don't we require an assert (Section 4) for each (type,concept) combination? We need an explicit mechanism anyway, and there are at least three arguments for asserts:

1. Where a type requires a template instantiation, compilers can have a hard time recovering from a failed attempt to instantiate, so it is better to have a place where a failure to instantiate is an error. (The alternative is a language rule that require all compilers to be able to recover from any failed instantiation).
2. We need a place to map abstract requirements, such as "we should be able to refer to a type **value_type** in a template definition" to the details of a specific type, such as "for an **int***, **value_type** is **int**". (The alternative is a language mechanism providing named members of built-in types [Stroustrup, 2003b]).
3. A type could accidentally have the properties required by a concept, but a different semantics.

We consider (3) unlikely and infrequent in real code, but others are less optimistic about this. An optional assert mechanism (**static_assert**) will allow people who worry to protect themselves. One reason this is feasible is that most "accidental matches" are not really accidental, but the result of deliberate designs that leaves two concepts or two type very closely related. An example of that is a non-mutable **Forward_iterator** that differs from a **Input_iterator** only in the protocol for the use of the dereference and increment operators (Section 6.6).

Problem (2) only applies when we need to map properties of a type to the requirements of a type. So a mapping mechanism has to be available, but is irrelevant to the many types where no mapping is needed. Thus, and optional assert mechanism suffices.

Note that the designer of a concept can explicitly assert that for that concept a **static_assert** required for a type to match (Section4.1).

Here, we present an argument that asserts cannot be compulsory. We first demonstrate that consistent use of asserts would be tedious to the point of poor engineering yielding unmaintainable code. Then, we present a worst case example where asserts are not logically possible and argue that this example is not an unimportant corner case. Please remember that asserts are clearly essential in some cases (4.1and 6.6) and very useful in others (??? and ???). Thus, asserts must be optional. Furthermore, the programmer can enforce the use of asserts for specific concepts where such asserts are logically necessary (???).

Implementation problems

The implementation problem (1) is the critical one. Our initial design discussions [Stroustrup, 2003a-c] did not require (compulsory) asserts, but after discussions with implementers, we decided that a compulsory assert would solve many real implementation problems and the absence of an explicit assert for some types would force implementers to use "speculative instantiation" that doesn't fit well into all current compilers. Thus, unless we require asserts in all cases (or at least in all cases that can lead to speculative instantiation), we cause implementation problems.

Consider how simple the implementation of concepts is if we have compulsory asserts: When a compiler sees a use of a predicate, either that predicate has been seen in an assert (and it is true) or it is false. No evaluation is ever required outside the asserts. The implementation of predicate evaluation degenerates into a table lookup. An assert states something about the relationship of a type and a concept. However, to say anything about a type, it has to exist, so an assert implies a request for instantiation. If an assert fails, it is a compilation error; no code will ever be generated for that translation unit and the quality subsequent error messages may be affected by the user-error.

If asserts are optional, the implementation becomes: when we see **C<T>** we look up in the table to see if we have already computed the answer; otherwise, we compute the answer and enter it into the table. The snag is that in evaluating **C<T>**, the compiler might have to instantiate the type **T** and that instantiation may fail. Unfortunately, instantiation may cause changes to the compiler state (such as symbol tables) that some compilers can't easily undo.

From an implementation point of view, we would prefer to require asserts. Unfortunately, the effect on programming style would be major (see below). In fact, we are convinced that requiring asserts would render concepts useless for their main intended purpose: to better support the current template-based programming techniques and to provide a base for significantly increased mainstream use of those techniques.

Programming problems

Consider why generic programming took off with C++ and not with Ada. A key reason is that Ada requires explicit definition of every instantiation (using Ada's instantiation operator **new**). In C++, we can create a type and use it without even mentioning it. For example, think of **make_pair()** or a **vector**'s iterator type returned by **begin()**. With explicit instantiation (or rather without implicit instantiation), the programmer would have to figure out what the types of the arguments of **make_pair()** were and explicitly instantiate the appropriate **Pair** type. Similarly, before using **v.begin()** we would have to look at the type of **v**, see what its iterator type is, and instantiate that. For example:

```
vector<My_type> v;
// …
copy(v.begin(), v.end(), somewhere);
some_function(make_pair(string("some name"),v));
```

Compulsory asserts would closely mirror Ada's explicit instantiation requirement, and we'd have to write something like:

```
static_assert Element_type<My_type>;
static_assert Forward_iterator<vector<My_type>::const_iterator>;
static_assert Value_type<string>;
static_assert Value_type<vector<My_type>>;
static_assert Value_type<Pair<string,vector<My_type>>>;
```

Clearly, we will at least have to "automate" some of these asserts and make many implicit.

Who would write asserts? When we implement a template, we can place requirements on the template arguments; that is, we can use concepts. However, we don't know the types with which our template will be used, so only the user can make the asserts. On the other hand, only we (and not our users) know which "helper templates" we use in our implementation. Without violating information hiding, our users cannot make assertions about those "helper templates"; some may even have access restrictions that make them inaccessible to users. Consider:

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

template<Value_type T, Allocator A = allocator<T> > class vector {
        // …
        typedef Checked_RA_iter<T> iterator;
        // …
};

// written by vector implementer:
static_assert template<Value_type T> Allocator<allocator<T>>;
static_assert template<Value_type T>
        Random_access_iterator<Checked_RA_iter<T>>;

// …

template<class T1, class T2>
        where Assignable<T1, T2>
void some_fct(T1& a, T2& b);

class My_type { /* … */ };

// written by My_type implementer:
static_assert Value_type<My_type>;
```

```
vector<My_type> vec;
My_type x;
int y;
// …

// written by some_fct() user:
static_assert Assignable<My_type,int>;
static_assert Assignable<int, My_type>;

some_fct(x,y);
some_fct(y,x);
```

This would be tedious (many have decided it to be too tedious). However, this example is radically simplified by the pervasive use of **Value_type** and the general (templated) asserts for **Allocator** and **Random_access_iterator** provided by the writer of **vector**.

The templated assert works like this: the first time the compiler sees a specific example, such as

Random_access_iterator<Checked_RA_iter<My_type>>

for **Random_access_iterator<Checked_RA_iter<T>>**, it acts as if it has seen

static_assert Random_access_iterator<Checked_RA_iter<My_type>>;

That way, an error occurs if the assertion doesn't hold (as opposed to the predicate just being **false**).

The situation is worse still when we consider types that weren't meant for the user to know about – aren't part of the explicit interface of the abstraction – or are indented to be for advanced users only. Think of **trait**s, **allocator**s, and **tuple**s in the implementations of **smart_pointer**s, **container**s, **lambda**s, etc. They are part of the implementation of an abstraction, yet depend on user type. If concept matching must be explicitly asserted, these types must be known to the end-user and correctly used (combined with user types in asserts).

The need to make assertions would become a portability nightmare unless such "implementation types" were standardized. If I used **My_helper** and you used **Your_helper** in the implementation of, say, **multimap**, the end-user wouldn't be able to write simple portable code. An **#ifdef** would be needed to choose between asserts that user's types meet the requirements of **My_helper** and **Your_helper**.

As described, this is clearly a violation of abstraction, of data hiding, and a serious maintenance problem. This is equivalent to the problems building and maintaining systems based exclusively on explicit mechanisms, such as macro-based generic programming.

"Encapsulated" assertions?

Could assertions about helper classes and implementation classes be hidden in the implementation of the templates that the user explicitly uses? For example

```
template<class T> class vector {
        // …
        typedef T* iterator;
        static_assert Random_access_iterator<iterator> {
                typedef T value_type;
        };
        // …
};
```

That's plausible, but would lead to repetition of asserts (in similar classes requiring similar asserts, e.g. the definition of **deque** would have to repeat assertions made in **vector**) and would work only if the template arguments of such "encapsulated asserts" were well known concepts. Furthermore, the compiler would have to instantiate the enclosing class template before discovering the assertion – thus defeating the crucial aim of using assertions to avoid "speculative instantiation".

Even if such encapsulation is considered acceptable for classes, it wouldn't suffice for function templates: A function has no place in its declaration that could be said to "encapsulate/hide" an assertion.

The problems with explicit asserts are not restricted to the single-argument predicates used to control individual arguments, we would also need to be explicit about the predicates used in **where** clauses. Consider:

```
template<Forward_iterator For, Value_type V>
        where EqualComparable<For::value_type,V>
For find(For first, For last, const V& v);
```

To use this for a type **My_type** and an iterator **My_iter**, we would need:

```
static_assert Forward_iterator<My_iter>;
static_assert Value_type<My_type>;
static_assert EqualComparable<My_iter::value_type,My_type>;

void f(My_iter p, My_iter q, My_type x)
{
        My_iter y = find(p,q,x);
        // …
}
```

We are convinced that compulsory explicit assertion of concepts is unmanageable. The use of explicit asserts simply doesn't scale – even using techniques such as assertions of templated predicates (Section 4.1).

A worst case example

In some cases, explicit assertions are simply impossible. Consider a variant of the example from Section ???:

```
// a helper concept:
concept Small<class T, int N> { where sizeof(T) <= N; }

// some implementation defined constant:
const int max = 200;

// helper function f() overloaded on the size of T
template<class T> where Small<T, max> void f(const T&);
template<class T> where !Small<T, max> void f(const T&);

template<class T> void foo(const T& t)
{
        // ...
        f(t);       // use f() as implementation detail
        // ...
}
```

Here, the function **f()** is used as an implementation detail, relying on overloading on concepts (we can express that idea in today C++ with "SFINAE techniques"). Note:

- The user of **foo()** shouldn't have to know about the "implementation detail" **f()**.
- The implementer of **foo()** shouldn't have to know about the implementation of **f()** (such as **f()**'s use of **max** and **Small**).
- The implementers of **foo()** and **f()** have no idea about the user type argument for **T**.

Since the implementer of **foo()** does not know the properties of **T** he cannot do a **static_assert** for **Small<T,N>** or for **!Small<T,N>** (one will be **true** and the other **false**, but the implementer of **foo()** has no idea which). Making **static_assert** compulsory bans such current techniques because there would be no reliable way to simply and elegantly express those with concepts.

We could try to make **f()**'s requirements on its argument part of **foo()**'s requirement. This is a common technique, but it doesn't work for the "Small example". Consider exposing **f()**'s requirements in **foo()**'s interface:

```
template<class T> void foo(const T& t)
```

```
        where Small<T, max> || !Small<T, max>
{
        // ...
        f(t);       // use f() as implementation detail
        // …
}
```

This exposes the implementation details so that the user of **foo()** can write an assert before calling **foo()**. However, in this case the exposure of the implementation details (**Small** and **max**) is useless and absurd: **Small<T, max> || !Small<T, max>** is always **true** so we didn't actually add anything useful. However, to proceed the caller of f**oo()** would have to assert something like

```
static_assert Small<My_type, max> || !Small<My_type, max>;
```

This would evaluate **Small<My_type, max>** which would fail whenever we have a large object. In general, a negative assertion after an "or" is absurd and should probably be prohibited to minimize confusion. For example, the assert above would mean:

> "check **Small<My_type, max>** and have the compilation fail if it is **false.** Then, if it was **false** (and the compilation failed) assert that **Small<Mytype, max>** is to be considered **false** from now on and never evaluated."

Thus, we cannot use that function **f()** in a system where explicit asserts are compulsory. That might be acceptable if the example was a mere curiosity, but it is simple a particularly short example of an important kind of code: a function with an implementation that relies on a call to a set of functions overloaded on concepts. Consider:

```
template<C1 T> void helper(T x);
template<C2 T> void helper(T x);
template<C3 T> void helper(T x);

template<class T> void foo(T x)
{
        // …
        helper(x);
        // …
}
```

How would we specify **foo()**'s **T** (with a concept or a **where** clause) to make **foo()**'s requirements explicit to its callers? In addition to whatever else **foo()** needs, it requires

```
C1<T> || C2<T> || C3<T>
```

For example:

```
template<C0 T> void foo(T x)
        where C1<T> || C2<T> || C3<T>;
{
        // …
        helper(x);
        // …
}
```

This can be handled (implicitly) by the compiler if the overloading is really specialization (some users of iterators, such as **advance()**, falls into this category, see ???). However, the general case is severely troublesome if explicit asserts are compulsory. The user of **foo()** shouldn't know about **foo()**'s dependence on **helper()** even if it could be clearly expressed. The call of **helper()** will succeed only if exactly one of **C1<T>, C2<T>**, and **C3<T>** holds (or if we have specialization), so to write an assert we would have to know which **helper()** function to call. In general, we cannot know that – and evaluating all the conditions would lead to the speculative instantiation that we were trying to avoid through compulsory explicit asserts.

Also, the set of overloads is open: We can always add a fourth **helper()** somewhere before the declaration of **foo()**, and that would imply changes to **foo()** or (worse) to users of **foo()**, writing asserts to use **foo()**. In general, the problem of implementation details "leaking out" of functions is made severe by a requirement of compulsory asserts. Here, we have only discussed one level of implementation. More realistic examples would involve more levels (e.g. a function calling functions, calling functions, calling functions with a few helper classes and constants thrown in for good measure). The requirements of each level would leak though to the next so that the "end user" will be faced with an incomprehensible mess of requirements. This exposure of implementation details would be exactly equivalent to the exposure currently seen in error messages related to template implementations.

Conclusions

We conclude that asserts cannot be compulsory. We also conclude that consistent use of asserts even in the theoretically feasible cases would be tedious to the point of poor engineering (???). One the other hand, asserts are clearly essential in some cases (??? and ???) and very useful in others (??? and ???). Thus, asserts must be optional.

Appendix B: Haskell type classes

It has been repeatedly suggested that the notion of concepts envisioned for C++ is nothing but a rephrasing of "type classes" as found in the Haskell[Peyton Jones, 2003] programming language. It is often further implicitly assumed that "what's good enough for Haskell is good enough for C++". Consequently, assertion (or "model declaration") should be compulsory as are the "corresponding" "instance declarations" of Haskell.

In fact, the notion of "concept" presented in this paper is radically different in spirit, meaning and purpose from Haskell's type classes. For one thing, type classes [Wadler, 1989] were designed for Haskell to constraint its Hindley-Milner-based type system to support a form of overloading (conventionally called ad-hoc polymorphism to emphasize its non-parametric nature). Haskell does not actually support overloading in the C++ sense. For example, the following fragment is illegal in Haskell because the symbol **areSame** is not allowed to be declared twice (with different types):

```
areSame :: Int -> Int -> Bool
areSame x y = primEqInt x y          -- use built-in equality comparator

areSame :: Int -> Int -> (Int -> Int -> Bool) -> Bool
areSame :: x y p = p x y             -- use custom comparator
```

What type classes provide is a form of "overriding" of virtual functions, as we will explain below. Of course, this illustration is not 100% right but, we hope that that will expose the fundamental ideas of how it works, the programming style it supports and why the design of concepts for C++ can not just adopt a variant of Haskell's type class design.

Type classes in Haskell are user-defined constraints in the type system, designed to guide the Hindley-Milner-Damas type inference style to support a variant of overloading. For example, Haskell's **Prelude** facility defines a type class named **Eq** that makes it possible to use the symbols == and /= with a variety of arguments, whose actual implementation vary from one type to another (i.e. they are *not* derived from instantiations of a unique parametric function). It is declared as

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    -- default implementations, that can be overridden in instance
    -- declarations of specific types.
    x /= y = not (x == y)
    x == y = not (x /= y)
```

What the above declaration says is that a type **a** belongs to the class **Eq** if there are known implementations for == and /= that take arguments of type **a**, or that type is declared to be **deriving** from class **Eq** (in which case the compiler generates implementations based on the structure of the datatype). The symbols == and /= are called method of the class **Eq**. Notice that the automatic generation of methods – based on explicit derivation -- is limited to a fixed set of Haskell standard type classes. Consequently, we will focuse on the more general facility of user-defined operations. Then, a type belongs to a class only through explicit declaration called *instance declaration*.

A type class can have default implementations, as illustrated in the **Eq** case. If the instance declaration does not mention an implementation of a method then, the default implementation (if any) is used, otherwise it is **undefined** (no compile-time error). In the case of **Eq**, if one method is implemented in an instance declaration then the default can be used for the other. Of course, if none is implemented then a call to either method will result in infinite loop.
Instance declarations supply method implementations and assert membership. For example, the type **Int** belongs to the class **Eq** and there is an instance declaration resembling

```
instance Eq Int where
        (==) = primEqInt
```

which says that the implementation of the method **==** is provided by the built-in function **primEqInt**. So, instance declarations are mapping from abstract interfaces to concrete implementations. A close translation to C++, from both conceptual and popular implementation point of views [Peterson 1993, Hall 1996] based on dictionary passing, is to regard a type class as a C++ polymorphic class template. So, assuming the trivial mapping of **Bool** to **bool** and **Int** to **int**, the above type class **Eq** declaration would correspond to

```
template<typename a>
struct Eq {
        // default implementation for == and !=. Can be overridden in
        // instance declarations.
        virtual bool eq_impl(a x, a y) const { return !neq_impl(x, y); }
        virtual bool neq_impl(a x, a y) const { return !eq_impl(x, y); }
};

template<typename a>
bool operator==(a x, a y)
{
        // lookup the real implementation in the instance declaration.
        const typename Instance<Eq, a>::Decl inst_decl;
        return inst_decl.eq_impl(x, y);
}

template<typename a>
bool operator!=(a x, a y)
{
        const typename Instance<Eq, a>::Decl inst_decl;
        return inst_decl.neq_impl(x, y);
}
```

Here, we use the traits **Instance<>** to map a Haskell instance declaration "**C a**" to the corresponding C++ implementation class **Instance<C, a>::Decl**, as follows

```
// This traits maps a type "a" to the dictionary that implements an instance
// declaration "C a".
// Absence of instance declaration is indicated by the type "void".
template<template<typename> class C, typename a>
struct Instance {
        typedef void Decl;
};
```

Now, assume we have a datatype **Position** defined as

```
data Position = Pos Int
```

and we declare it to be a member of the type class **Eq**

```
instance Eq Position where
        (Pos x) == (Pos y) = x == y
```

The corresponding C++ declarations would be

```
struct Position {
        const int value;
        explicit Position(int v) : value(v) { }
};

struct Eq_Position_instance : Eq<Position> {
        bool eq_impl(Position x, Position y) const
                { return x.value == y.value; }
};

// Indicate that Eq_Position_instance implements the required methods:
template<>
struct Instance<Eq, Position> {
        typedef Eq_Position_instance Decl;
};
```

In general, type classes support inheritance – translating directly to C++ inheritance – and instance declarations map to overriding. Using Haskell type classes leads to a traditional object-oriented style, where everything must fit a given hierarchy before use. That style is already supported by C++.

Note that the C++ program is not a perfect translation of the Haskell equivalent; we did not think it would be a good idea to obscure the main points by dealing with minor details. However, the translation scheme captures the general idea and the style of programming it supports.

References

[Hall, 1996] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, Philip Wadler: *Type Classes in Haskell*. ACM Transaction on Programming Languages and System, 1996.
[Peterson, 1993] John Peterson, Mark Jones: *Implementing Type Classes*. PLDI 1993.
[Peyton Jones, 2003] Simon Peyton Jones: *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press 2003.
[Wadler, 1989] Philip Wadler, Stephen Blott: *How to Make Ad-Hoc Polymorphism Less Ad Hoc*. Conference Record of the 16th Annual ACM.

Appendix C: Mixing concepts and unconstrained parameters

When all template parameters have been declared with concepts, we can separately check template definitions and template uses. That leaves two questions:

- What can we do if some, but not all, template parameters have concepts?
- Are there important programming styles that require that some, but not all, template parameters have concepts?

Our basic answers are "check what can be concept checked early and leave the rest until instantiation time" and "yes".

Mixed parameter templates

Consider

```
template<C T1, class T2> void f(T1 t1, T2 t2)
{
        ++t1;
        ++t2;
        t1+t2;
}
```

Clearly, we could check **++t1** in the absence of any further information. The concept **C** determines the complete set of valid operations for **t1**.

Equally clearly, no checking is possible **++t2** until an actual type **T2** is known.

A little thought makes it clear that no early checking of **t1+t2** is possible: Even if **C** provided for a **+** operation, a **T2** might "highjack the operation with a better matching **+**. Conversely, if C doesn't provide a + operation, a **T2** might provide one for which a **T1** is a match.

The obvious conclusions follow:

- Any part of a template definition that depends on a "plain class" template parameter cannot be checked until instantiation time.

- Any part of a template definition that depends only on concept parameters can be checked immediately.

We will consider non-type template parameters (e.g. int, function, and template template parameters).

???

So are mixed parameter templates ever needed? Are the ever useful?

<<To be written>>

<<fill() again, apply()>>

Appendix D: Semantic properties of concepts

Here, we have only considered properties of types and values when stating assumptions that are part of concepts. However, semantics properties of functions play central roles in generic programming and several transformations could be done by a compiler when it has those information. As a general rule, those are run-time properties, therefore would need special annotations. Consequently, this concept design does not propose to extend "predicates on types" to "predicates on functions".

This appendix is simply a note pointing out this area of future (probably post C++0x) exploration. A sorting algorithm, for example, may be idempotent; that is, given a sequence **s** the expression **sort(sort(s))** is the same as **sort(s)**, therefore the outer call to **sort()** could be removed as unnecessary. Similarly, given a sequence **seq** of **bigint** and knowing that the operation **operator+(const bigint&, const bigint&)** is associative, the expressions **accumulate(seq.rbegin(), seq.rend(), bigint())** can be transformed into **accumulate(seq.begin(), seq.end(), bigint())**. That is, summing forwards or backwards are equivalent. Similar transformations are done internally by optimizing compilers on built-in types. Similar or equal support for user-defined types would undoubtedly need some annotations on functions (not just on types); however, we need more work and experience in that area.