N2906=09-0096
2009-06-21
Bjarne Stroustrup

Simplifying the use of concepts

Abstract

This proposal to simplify the use of concepts by making concept maps rare. It provides "explicit refinement" as a more specific remedy for the problems that otherwise required similar concepts to be explicit to avoid errors. It further proposes to make all concepts implicit/automatic, to make calls of similarly constrained functions from within a constrained function legal, and (consequently) to make all standard-library concepts implicit. Furthermore it provides a mechanism to allow either a member or a free-standing function match an associated function requirement, making many explicit empty concept maps redundant.

To motivate these changes, a few problems with the usability of concepts as currently defined are presented. I argue that changes are necessary for concepts to succeed outside a small group of experts.

The current definition of concepts and requirements for use drowns the programmer in complexities of a magnitude not warranted by the need to express type-checked 9constrained) generic programming.

Introduction

This note is a follow-up on the long "Are concepts required of Joe Coder?" thread. That thread started when Howard Hinnant asked that question in the context of a design (of a utility) that could be done in two ways: One would require quite a lot of users – not necessarily expert users – to write concept maps. The other design – arguably less elegant – would avoid concept maps (and concepts) so as not to require users to understand anything significant about concepts. From there, the discussion branched out into several related directions, incl. how the new range-for should be specified, how to ensure that a newly written type match a concept, and whether explicit or implicit concepts should be the recommended style and/or the default for concepts.

I will argue that "average programmers" should write concepts and (less frequently) concept maps, that it is good for C++ that they do so, and that they will only do so if they see benefits from doing so. Most C++ programmers should have a sufficient understanding of concepts to write one as an experiment or for occasional use. Who is "Joe Coder?" asked Peter Gottschling. Great question, I answered:

> I think *most* C++ programmers are "Joe Coder" (I again register my opposition to that term). I'm Joe Coder most of the time and with most libraries. I expect to remain so as

> long as I keep learning new techniques and libraries. Yet, I want to use concepts (and, when I must, concept maps). I want the "doctrine of use" radically simpler than the subtle expert-only use of facilities we have now.

The alternative is for most programmers to discard concepts and libraries built using them out of fear of the unknown and complex. Since concepts is one of the most prominent features of C++0x, people avoiding them would be very bad (adding to C++'s reputation for bloat and complexity). I will argue that

1. the proper ideal and proper language design is for concept maps to be implicit except where a map is clearly needed to add information.
2. if in the eyes of a programmer, a type "obviously match" a concept, the language rules should (if at all possible) not put language-technical obstacles in the way of a match.

Please note that I'm not saying that "concept maps are bad" (there are many cases where a concept map is obviously needed) or that "it ok to define concepts that differ only semantically and also exclusively rely on syntactic matching" (there are concepts that must be explicitly distinguished from each other because they cannot reasonably be distinguished syntactically). I'm arguing that the ideal is "automatic concepts that just work" and that this ideal can be achieved in many cases while still retaining effective, compile-time, protection against known problems (known from traditional unconstrained templates and from what I think of as first-generation constrained templates).

My aim is to articulate a set of guidelines for the use of concepts and concept maps and propose adjustments to the language mechanisms to reflect such guidelines.

The rest of this paper is organized like this:

1. A bit of language philosophy
2. The purpose of concept maps
3. Leaking implementation details
4. Viral concept maps
5. Suggestions
6. Which standard-library functions should be explicit
7. Matching types to concepts
8. Proposal text

I started writing this proposal and the reflector discussions related to it with less radical aims, but the complexity and subtlety that met me at every turn convinced me that only a decrease of the complexity of the language itself would do. A friend who silently followed the reflector exchanges emailed me this reminder:

> "How do we convince people that in programming simplicity and clarity -- in short: what mathematicians call 'elegance' -- are not a dispensable luxury, but a crucial matter that decides between success and failure?"
>
> -- Edsger W. Dijkstra

A bit of language philosophy

Types and type checking come in many flavors. Thousands of thick books and articles have been written on the topic (most using more Greek letters than I prefer) and every professional programmer will have noticed practical differences between languages (e.g. C and Python) and even between facilities within a language (e.g. C++ classes and templates). We can think of a spectrum of languages from languages where two object are of the same types iff they have the same name (e.g. C++ classes without inheritance or typedef) to languages where two objects are of the same type iff they have the same "structure" (e.g. a purely tuple-based type system). The former are called "nominal" (they are name-based), the latter "structural" (they are based on some form of structure, such as layout or member function names). To simplify, we can talk about type systems being more or less nominal even though the design choices do not exactly fit a single line from purely nominal to purely structural. C++ classes (like C/C++ built-in types) fit towards the nominal end of the spectrum (notions of inheritance and compatible types keep them away from the extreme). "Duck typing" (popular in dynamically-typed languages) and C++ templates are closer to the structural end of the spectrum, with "the structures" most frequently matched (to determine type equivalence) being function and type names.

At a first approximation, we can say that structural type systems help the programmer by making "things" easy to say and maximizing interoperability ("if they look the same they can be used in the same way"), whereas nominal type systems help the programmer by forcing "things" to be explicitly expressed and catching errors ("unless you explicitly said so, it ain't so"). For example, in a purely dynamic language we can have something like this

```
f(x) { print x+1; }
f("asdf");      // print asdf1
f(2);           // print 3
f(2.3);         // print 3.3
```

whereas for a language without overloading you'd have to write three separate functions. With functions calling functions, you may need an exponential explosion of the number of functions. Conversely, in a strictly nominal statically typed language we must be specific about type

```
int f(int x) { print x+1; }     // more verbose: int specified and specified twice
f("asdf");      // error "asdf" is not a string
f(2);           // print 3
f(2.3);         // error: 2.3 is not an int
```

C++ with templates and implicit conversions approximate the dynamically-typed languages – even to the (almost certain) semantic error of adding an int to a string.

I conjecture that a major reason that generic programming succeeded in C++ where it failed in languages using object-oriented programming with explicitly specified interfaces (as in C++ and Java – a nominal type system) is that templates are essentially structural (and similar to many dynamically-typed languages where OO techniques have succeeded in supporting variants of generic programming). The added freedom of expression and flexibility provided by templates over class hierarchies with explicit interfaces has been a major advantage to C++ programmers, arguably the key to modern C++. That freedom has also been a significant source of problems (especially poor compile-time error handling).

I see concepts as a way of compensating for the weaknesses of templates stemming from their extreme structural nature. The major design challenge is to do this without pushing the type system so far over into the nominal camp that we reintroduce the problems of rigidity, verbosity (notational overhead) and overspecification (explicitly specifying details that could be deduced) found with class hierarchies and object-oriented programming. The way I see it, catching the errors we see with template arguments is easy: we just use an equivalent of nominal typing (such as classes). The challenge is to do just enough of this without going so far as to damage generic programming, performance, etc. As my old advisor used to say (in the context of security) “protection is easy, it’s granting access that’s hard.”

Concepts were meant to make generic programming easier as well as safer. It is part of a whole collection of features aimed at simplifying GP, together with template aliases, auto, decltype, lambdas, etc. However, “concepts” is a complex mechanism and its language-technical complexity seems to be leaking into user code. By “language-technical complexity” I mean complexity arising from the need of compiler/linker technology rather than complexity from the solution to a problem itself (the algorithm).

My particular concern is that in the case of concept maps, in the name of safety we have made templates harder to use. We require programmers to name many entities that would better be left unnamed, cause excess rigidity in code and encourage a mindset in programmers that will lead to either a decrease in generic programming (in favor of less appropriate techniques) or to concepts not being used (where they would be useful). We have overreacted to the problems of structural typing.

Concept maps

Concept maps play a key role in the mapping between requirements (concepts) and types. Without concept maps, a type would have to *exactly* match a requirement (either structurally or nominally). For example, if I have a type that I’d like to pass to your algorithm, my type would have to have the name you expected (if your type was expressed as a non-template function) or

have the structure you expected (the right functions, operators, etc. if your type was expressed as a template function). However, there are plenty of types that *almost* match a set of requirements. In the absence of concept maps, I have to use workarounds:

1. For a non-template function, I'd have to change my type, rename my type, or somehow create a new type (or a synonym) with the name your function expects. Deriving from a base class used to specify the interface is a classical solution, which unfortunately is intrusive.
2. For a template function, I'd have to change my type or somehow create a new type with associated functions, types, etc. that can be used as expected by your function. There is a fair amount of freedom of choice in exact argument types, member vs. free standing operations, etc.

I see concept maps as a mechanism to make such adaption simpler and more systematic. That's all. In particular, we could use concepts without concept maps by relying on the conventional adaption techniques (described above) developed for templates and ordinary functions (we would just prefer not to).

So, what are concept maps good for? Assume that we don't want to modify types that we want to use as template arguments, or wrap them in other types, etc. then a **concept_map** is needed

1. if information needs to be added for a type to be usable for a concept (e.g., a **concept_map Iterator<int*>** to add a member type **value_type** to **int***)
2. if two concepts in a derivation/refinement hierarchy differ semantically (e.g., **ForwardIterator** and **InputIterator**), but not (or only slightly) syntactically, we must disable automatic matching of at least one and a concept map is needed to specify which concept – if either – a type matches.
3. to prevent a type with unrelated semantics, but identical member names, from implicitly and accidentally match a concept (e.g., a **Cowboy** class with a **draw()** function might accidentally be accepted by a function requiring a **Shape** concept with a **draw()** rendering function).

Use #1 is to my mind the primary use of concept maps: to add information to non-intrusively fit a type into a framework specified through concepts. Use #2 is essential in a few cases (e.g., **ForwardIterator** and **InputIterator**). However, we have lived happily with class hierarchies and templates for decades without use #3 reaching anyone's top 100 list of traps and pitfalls, so it is not on my top 100 list of C++ problems needing solution. One reason "accidental match" hasn't been a major problem is that we rely on simultaneous matches of both names and types. For example, only if the **Cowboy**'s **draw()** has the same argument and return types as **Shape**'s **draw()** and the same holds for every other function in the concept/type can the problem slip past the compiler (to be caught be the simplest testing). Also, unrelated types tends to get muddled

only when they are used for similarly named functions, so that overload resolution often catches the mistakes as ambiguities.

My conjecture is that most real-world types do not fall into any of those three categories. The obvious conclusion is that even though concept maps are essential (for reasons #1 and #2) they should be used sparingly. If they are frequent, the reason must be that we are using concept maps for some other reason and/or that there are technical problems in the rules for concept maps.

There is an obvious "other use" for concept maps: To ensure early error detection for concept/type combinations: By using a concept map, even an empty concept map, we can guarantee early detection of errors – exactly as we get from a nominal type system. For example:

```
class Foo : public Bar { … };          // Foo is a Bar
```

This guarantees that a **Foo** is a **Bar** (the OO "is a"). Similarly,

```
class Foo { … };

concept_map Bar<Foo> { };// Foo can be used as a Bar
```

This guarantees that a **Foo** can be used as a **Bar**. Without the (empty) concept map, errors would be found only at the first use of a **Foo** as a **Bar**.

Incidentally, I consider the **concept_map** variant superior to conventional inheritance (in respect to type checking) because it is non-intrusive.

Systematic use of **concept_map**s in this way (as would happen if every concept was explicit) would give us the benefits of interfaces in OOP at the cost of some of the inflexibility of OOP and a slight added increase of verbosity compared to OO. I consider that a serious problem/danger in the context of a language feature aimed at improving generic programming. If you want OO-style type checking in C++, you know where to find it.

My ideal is an expression of GP that is less verbose than and as flexible as what we have with unconstrained templates, but with vastly improved error checking, error reporting, and overload resolution. I think we *almost* have that with concepts, but that a few details of the concept definition and some major flaws in how we think of their use could move the C++ community far from those ideals.

I have come to think of extensive use of explicit concepts as a serious mistake and a departure from the ideals of generip programming as embodied in the STL. I observe that Alex Stepanov's latest and greatest book "Elements of Programming" do not use any equivalent of explicit concepts.

Problems

What concrete problems am I trying to address?

- In which ways can constrained templates be less flexible and more verbose that unconstrained ones?
- How and when can overuse of explicit concepts add to the problems of inflexibility and verbosity?
- How can the language rules be modified to alleviate these problems (by increasing flexibility and minimizing code complexity) without causing type-safety problems?

My claim is

- that there is a lack of flexibility stemming primarily from the use of better specified interfaces,
- that some of that inflexibility is good (even if users initially don't appreciate it),
- (but) that an overemphasis on interface names (nominal checking of names of concepts) and explicit concepts unnecessarily increase such problems.

The next sections present examples of problems and suggest remedies. The remedies simplify programming, shorten code without compromising type safety and also simplify the language itself.

The debug example

Consider a simple example of a traditional unconstrained template:

```
template<class T> f(T& t) { store(t); }
```

now take an equivalent constrained template

```
template<ST T> cf(T& t) { store(t); }
```

where ST is a simple concept that (just) allows a value to be "stored" using store(). Now I want to do a bit of debugging using cerr:

```
template<class T> f(T& t) { cerr<< "storing " << t; store(t); }

template<ST T> cf(T& t) { cerr<< "storing " << t; store(t); }          // error
```

This **f()** still works (assuming **ostream** can handle a **T**) but **cf()** does not. We can make **cf()** work only by modifying its interface. Thus, the unconstrained design is more flexible than the constrained one. By the way, this is not a random example, the equivalent the **cf()**'s problem happens all the time in Haskell.

I'm not arguing against the use of a concept such as **ST**. On the contrary, I'm strongly in favor of explicitly expressing interfaces (**cf()** promises to use its argument only as an **ST**) and really **f()** is relying on something unstated (**cerr** can handle a **T**) which is obviously not universally true. Forcing **cf()** to use its argument only as an **ST** saves us from problems related to undisciplined

use of arguments – in particular, this is key to compiler checking of template bodies as opposed to trying for exhaustive checking with a variety of template argument types "to see if they work" (the most popular example is an algorithm using **p+1** on a **p** that is only guaranteed to be a forward iterator – we want to catch such errors). However, constrained templates are indisputably less flexible than constrained ones and undoubtedly someone will complain loudly against that. Unless we limit this inflexibility to cases where it is necessary and beneficial the complaints will be valid.

What is the general case (or cases) of this little debug example? Fundamentally, we tried to make a change to the implementation of a template and found that we had to modify its interface in a way that would surprise someone coming from an unconstrained template background (or from a language with duck typing). In this case, I think we must accept that to use an **ostream** for a **T** we must modify the interface, find a way of saying "print **T** only if you can", or use some hack. The "print **T** only if you can" could be this clever technique (due to Dave Abrahams):

```
struct debuglog {
        debuglog(ostream& os) : os(os) {}
        ostream& os;

        // Identity adds no constraints, but causes this to be a constrained template:
        template <typename T>
                requires Identity<T>
        debuglog operator<<(T const&) const { os<<"<unprintable>"; return *this; }

        template <typename T>
                requires Identity<T> && OutputStreamable<T>
        debuglog operator<<(T const& x) const { os<<x; return *this; }
};
```

Unfortunately, this postpones the error message "**<unprintable>**" to runtime. That might be acceptable for the specific task of debugging, but it is not a general solution.

The hack could be a **late_check** (a hack because it violated the spirit of interface based checking). In general, I prefer not to require such cleverness or to make **late_check** an idiomatic part of concept-based programming. A **late_check** pushed the error message to link time.

I don't actually suggest a remedy for this example. It simply demonstrates that programming using well-specified and enforced interfaces carries a cost. This particular cost I'm willing to pay. However, similar examples are more bothersome.

This example points to a serious danger: programmers may choose very wide (general) interfaces to simplify changes to the implementation and to keep interfaces stable. This would be

unfortunate because comprehensibility, error detection, and generic programming depend on narrow (specific) interfaces.

Subsets

The problem with the debug example, which caused the need for cleverness, hacks, or interface changes, was to use of a facility not provided by the declared interface (concept). But what if the compiler rejected an implementation that did not in fact use facilities not specified in its interface (concept)? Such examples would cause legitimate complaints of C++ becoming a "discipline and bondage language." Such examples exist. Will they become common? Consider:

```
concept AB<typename T> {
        void a(T&);
        void b(T&);
};

concept A<typename T> {
        void a(T&);
};
```

Obviously, every type that's an **AB** is also an **A**, so we could reasonably write:

```
template<A T> void g(T);

template<AB T> void f(T t)
{
        g(t);   // valid call?
}
```

Any non-expert would answer "yes!" Obviously, an **AB** has an **a()** as required by **A**. Obviously, **A** is a subset of **AB**. Someone with some knowledge of concepts might say "no" and suggest that **AB** needs to be derived from (a refinement of) **A** for the compiler to notice the obvious subset relationship. However, in real code we may not be able to modify the definition of **AB**. Requiring derivation/refinement is intrusive and removes an advantage of GP by moving it closer to OOP.

Before trying to resolve this example, consider a related (but simpler, example:

```
concept ABx<typename T> {
        void a(T&);
        void b(T&);
};
```

```
concept Ax<typename T> {
        void a(T&);
};
```

Obviously, every type that's an **ABx** is also an **Ax**, so:

```
template<Ax T> void f(T);
template<ABx T> void f(T t);

void h(X x)     // X is a type for which a(x) is valid
{
        f(x);   // ambiguous
}
```

In other words, in general, we have to protect against **ACx**'s **a()** being different from **Ax**'s **a()**. If these two **a()**s can be different we cannot accept the call **g(t)** above because it would call "the wrong **a()**."

Reluctantly, I accept that in general, we must require a statement that an AB and an A may be considered equivalent:

```
template<AB T> concept_map A<T> { }    // every AB is an A
```

In other words, we (non-intrusively) say that every **AB** is an **A**. Or "for every type **T** that is an **AB**, please check that it is also an **A**". Try explaining the need for saying that explicitly and separately to a novice. Unconstrained templates resolve such cases all the time (rather late) and **auto** concepts does that all the time (on first use). Unfortunately, that doesn't even work (as far as I read the WP). First we try to place that concept map in **f()**; after all, we need it as part of **f()**'s implementation:

```
template<AB T> void f(T t)
{
        template<AB T> concept_map A<T> { }    // every AB is an A

        g(t);   // valid call?
}
```

That's a bit verbose, and it does not work: a concept map cannot be local (see grammar). So we try to move it outside **f()**, "leaking" an implementation detail:

```
template<AB T> concept_map A<T> { }
```

```
template<AB T> void f(T t)
{
        g(t);   // valid call?
}
```

Still no luck! I find it hard to understand the WP text but James Widman (thanks) assures me that it does not. It seems that there are scopes problems (e.g. see 3.3.9). There is an issue on this (CWG issue 870).

Whether this resolution is sufficient remains to be seen. In particular, I wonder if allowing the local concept map would be needed to avoid implementation leakage. My work on "intermediate results" (below) may answer that question "real soon now."

Aside: is it allowed to define a concept map **C<X>** twice? If not managing to have only a single map for **C<X>** in a scope could be challenging and if so we have a maintenance problem (how do we ensure that every **C<X>** defines the same map?).

Type of intermediate results

How to specify, constrain, and/or deduce an intermediate result in an algorithm has been a problem since the earliest discussions of concepts (2002). The simple **void f(T t) { g(t); }** example above does not use visible intermediate types. However, in most realistic application domains, we generate intermediate results from expressions such as **f(x,g(),h())** and **a+b*c**. Often the type of such intermediate results is non-trivial and most important for the algorithms; think: expression templates, **pair**, **tuple**, **matrix**, etc. The practical difficulties in managing explicit concept maps for such cases are non-trivial. Some of the students here (who has written lots of concept code in domains not usually discussed) claim that the "intermediate type problem" is completely unmanageable for real code.

Unfortunately, I don't have the time to write a paper on this for the pre-Frankfurt meeting, but I hope to be able to provide more information before the Frankfurt meeting.

When are automatic/implicit concepts insufficient?

We cannot manage with just automatic/implicit concepts. To remind ourselves and summarize, consider two examples **ForwardIterator** / **InputIterator** (from N????) and **ContiguousIterator** / **RandomAccessIterator** from (Doug Gregor in the reflector discussions). These examples demonstrate that implicit concepts by themselves can easily be (mis)used in unsafe ways:

```
auto concept ContiguousIterator<typename Iter>
        : RandomAccessIterator<Iter> {
        requires LvalueReference<reference> &&
LvalueReference<subscript_reference>;
}
```

The idea (implicit except for the name the concept) is that a **ContiguousIterator** is an iterator to a contiguously allocated sequence of elements. Knowing that elements are contiguously allocated opens the possibility for significant optimizations. For example:

```
template<ContiguousIterator InIter, ContiguousIterator OutIter>
        requires SameType<InIter::value_type, OutIter::value_type>
        && POD<InIter::value_type>
OutIter copy(InIter first, InIter last, OutIter out) {
        if (first != last)
                memmove(&*out, *&first, (last - first) * sizeof(InIter::value_type));
        return out + (last - first);
}
```

Now there is no (reasonable and general) way that a compiler can know whether a given container provides iterators that are **ContiguousIterator**s. Syntactically, a **ContiguousIterator** is identical to a **RandomAccessIterator**. This can lead to the **ContiguousIterator** version of copy to be invoked for a "plain RandomAccessIterator", such as **deque::iterator**, with disastrous results. The "conventional solution" is to declare both **ContiguousIterator** and **RandomAccessIterator** as explicit concepts and let their users write concept maps to say which are which. This is tedious (though some argue "not too tedious"). However, my observation is that the **ContiguousIterator**/**RandomAccessIterator** design is fundamentally flawed. Derivation/refinement says that the refined (most derived) version of an operation will be used when there is a choice. This language rule is fundamental and reasonable (think **advance()**). However, this kind of substitution requires that the derived/refined operation has the same semantics as the less refined one; in other words that it is a pure optimization. This is not the case in the **ContiguousIterator**/**RandomAccessIterator** example. The writer of the optimized **copy()** assumed (erroneously) that it would be applied only to **ContiguousIterator**s but the refinement rules ensures that we can get the "optimized **copy()**" invoked for the random access iterators provided by **deque**. The optimization provided by **copy()** is a good and important one if the sequence really is contiguously allocated, but a disaster otherwise.

I consider the **ContiguousIterator**/**RandomAccessIterator** example roughly equivalent to overriding a virtual function with a version with different semantics: Substitutability is sacrificed even though it is assumed by language rules and conventions of use.

What would be a good solution to this problem? The writer of **ContiguousIterator** knows (or at least can know) that there is a problem with the refinement from **RandomAccessIterator** in some cases, so he solves it by requiring *every* user of **ContiguousIterator** to take an action to avoid it (even though only uses of operations using **ContinuousIterators** that are not pure optimizations of equivalent operations using **RandomAccessIterators** are affected). This being

burdensome, he then may provide a remedy in the form of one or more concept maps that he writes himself.

I see that as a patch upon a patch arising from the lack of a specific remedy. And that "specific remedy" is a statement that says that you cannot implicitly distinguish between a **ContiguousIterator** and a **RandomAccessIterator**. In particular, this does not mean that every use of **ContiguousIterator** or **RandomAccessIterator** requires a concept map, just that we must avoid treating a **ContiguousIterator** as a specialization of **RandomAccessIterator**.

What would be a better, more specific solution, to this class of problem? We should make sure that the burden of ensuring that a specialized version of a more generalized version is used only when appropriate is placed (exclusively) on the provider of the specialized version; that is, not on the provider of the general version (who cannot know if a specialized version will ever exist) and not on the user (who does not in general know that there are two versions).

Let's see if we can do that. To do so, we have to break this sequence of events:

1. Programmer **A** defines concept **CA**
2. Programmer **B** defines concept **CB** derived from **CA**, but syntactically very similar yet semantically different
3. Programmer **U** manages to use a type **T** somehow meant to be **CA** as a **CB**

Note:

- **A** does not know about **B** or **U**.
- **B** knows about **CB** and **CA** (but may not be able to modify **CA**).
- **U** may only know about **CA** or **CB** and would rather know as little as possible.

Basically, the problem boils down to:

1. What can **B** do to protect **U**?
2. What can we – as language designers – do to "remind **B** to protect **U**" and to help **U** if **B** forgets?

Thus, this is *not* a question of explicit vs. implicit concepts. It is an issue of derivation (refinement): Can we identify the cases of derivation that may cause problems? I think so: We must be able to move "up" a concept hierarchy to get optimizations (e.g. for **advance()**). We (implicitly) move "up" because we assume that identical functions (name plus signature) in a hierarchy have the same semantics (just like for virtual functions). If that's not the case, the designer of the derivation/refinement should say so, forcing a move "up" to a derived concept to be explicit. That would take care of the "inappropriate optimization" cases.

I conclude that we need

1. A way to disable implicit conversion/selection “up” a concept hierarchy, to be applied by the definer of a (derived/refined) concept providing semantically different (“potentially unsafe”) versions of base concept operations.
2. An explicit way of enabling such a conversion/selection "up" a concept hierarchy for a particular set of types.

Consider:

concept ForwardIterator<class T>

: explicit InputIterator<T> { ... };

This says that the derivation/refinement **ForwardIterator : InputIterator** is not (also) a specialization so that if we call an algorithms (e.g. **advance()**) for **InputIterator** with a type that also matches **ForwardIterator** we do not convert/select “up” to the **ForwardIterator** version. This eliminates the errors for the **ForwardIterator** / **InputIterator** example and the **ContiguousIterato**r / **RandomAccessIterator** example can be handled in the same way.

However, it does so at the cost of eliminating the optimizations, so we'll re-enable those where appropriate. For example:

concept_map ForwardIterator<int*> {}

This says that we may consider an **int*** a **ForwardIterator** even though it is also an **InputIterator** and we don't in general allow such movement “up” to **ForwardIterator**.

Obviously I hijacked **explicit** and **concept_map** to achieve a familiar syntax. I don’t think I did violence to the semantics.

This simple mechanism eliminates the errors for the **ForwardIterator**/**InputIterator** and **ContiguousIterator**/**RandomAccessIterator** examples even if all of those concepts were (as I would like them to be) implicit/automatic.

Let’s consider how that would work for our hardest case, the standard iterator hierarchy augmented with Doug’s **ContiguousIterator** to illustrate extensibility. I hope the abbreviations are obvious:

CI -> RAI -> BI -> FI -> II -> I

Currently (N2857), **I**, **II**, **FI**, **BI**, **RAI**, and (in this discussion) **CI** are explicit. That solution is to push the decision on which types have which hierarchical relations onto the type designers (and then lessening the burden by using templatized concept maps). Basically, this makes a mockery of the concept hierarchy: we don't use it except as a prop for the concept maps.

The alternative solution is to make all iterator concepts implicit/**auto**, but explicitly make **CI** not a specialization of **RAI** and **FI** not a specialization of **II**:

```
concept FI<typename T> : explicit II<T> { … }

concept CI<typename T> : explicit RAI<T> { … }
```

So what concept maps do we need? (please don't nitpick technical details not relevant to the main argument).

```
template<class T> concept_map I<T*> { typedef T* value_type; } // add value_type
                                                               // to pointers

template<class T>
        concept_map FI<vector<T>::iterator> { };   // but not for istream_iterator
template<class T>
        concept_map FI<list<T>::iterator> { };   // but not for istream_iterator

// what we don't need is a concept map saying that list and deque are BI and vector is RAI

template<class T>
        concept_map CI<vector<T>::iterator> { };   // but not for deque
template<class T> concept_map CI<T*> { };
```

So, consider

```
template<FI I> void advance(I p,int n);
template<RAI I> void advance(I p, int n);
template<CI I> void advance(I p, int n);

template<FI I> algo(I p)
{
        advance(p,4);
}

input_iterator<int> pii;
int* pi;
list<int> li;
deque<int> di;

algo(pii);      // error:: pii is not FI
```

```
algo(pi);      // ok: use CI advance
algo(li);      // ok: use FI advance; no [] or +
algo(di);      // ok: use RAI advance; di has [], +, etc. so it is a RAI, but not a CI
```

It seems to me that using explicit refinements rather than explicit concepts, we save people from writing redundant concept maps, teach people to directly address the semantic problems, and not to unnecessarily fear automatic concepts.

Concept ambiguities

How should we handle two identical concepts not related by refinement? Consider:

```
concept A<typename T> { void f(T&); }
concept B<typename T> { void f(T&); }

template<A T> f(T&);
template<B T> f(T&);

class X { };

X x;
f(x);
```

The call **f(x)** is ambiguous (of course), but how do we get to call one of the **f()**s? If **A** and **B** are explicit concepts, we simply give a concept map for the one we want (and not also for the other). That's at best brittle. If **A** and **B** are implicit concepts, we are simply stuck.

I don't propose to solve this problem. I don't think that it's all that important, but if it turns out to be, we can add some casting/resolution mechanism to apply at the point of call (e.g. **B<X>::f(x)**).

Note that this kind of ambiguity is the kind we resolve within a refinement hierarchy for concepts that differ only semantically. For example

```
concept RAI<typename T> { … }
concept CI<typename T> : explicit RAI<T> { }

template<RAI T> f(T&);
template<CI T> f(T&);

class X { … }; // syntactically matches RAI and CI
```

```
X x;
f(x);    // ok: RAI's f()
```

We could not – without further (explicit information) assume that an **X** could be used as the more refined concept **CI**. If X is a CI we have to say so

```
concept_map CI<X> { };
f(x);    // ok: CI's f()
```

Explicit concepts are viral

Given N2857, someone will define a concept to be implicit and a user thinks that it would be better if it was explicit and occasionally, someone will define a concept to be explicit and a user thinks that it would be better if it was implicit. What can we do in such cases?

I argue that implicit concepts (structural matching) is the ideal (compared to explicit concepts (nominal matching)) in the case of constrained templates. Furthermore, it is not unimportant whether there is a default (implicit/explicit) or what it is. A default of explicit leads to a proliferation of concept maps – and a mindset that goes with them. A default of implicit leads to the need for (far fewer) explicit refinements.

Say that someone defines an explicit concept and the resulting need to write concept maps bothers me, so I try to build an implicit concept from it:

```
concept Foo<typename T> { ... };   // explicit
template<Foo T> void f(T);

auto concept<typename T> Afoo : Foo<T> {  };   // implicit
template<Afool> void g(T);

X x;    // atype that matches Foo without any need for mapping

f(x);   // error no Foo<X> concept map  (I can write one if I want to)
g(x);   // ok: X matches Afoo
```

However, I would probably want to use some of the functions written requiring **Foo**, such as **f()**. If that works, we could write

```
template<Afoo T> inline void g(T t) { f(t); }
```

But that wouldn't work because **Foo** is explicit, so I try

```
template<Afoo T> inline void g(T t)
{
        concept_map Foo<T> { }

        f(t);
}
```

but that doesn't work, so I'm back to

```
concept_map Foo<X> { }

template<Afoo T> inline void g(T t)
{

        f(t);
}
```

But that was the concept map that I was trying to avoid in the first place – and it doesn't work either.

I conclude that explicit concepts should at best be used.

Here is another workaround, due to Peter Gottschling:

```
concept CExplicit<typename T> { ... }

template <CExplicit T>
void f(const T& x, const char* xc)
{
   T y(x);
   std::cout << "In f with x = " << xc << "\n";
}

// Faking concept_map:
auto concept CAuto<typename T> { }
template <CAuto T> concept_map CExplicit<T> {}
```

But aren't implicit concepts also viral? Yes, once a concept is implicit

1. We can always write concept maps for implicit concepts to ensure early checking
2. We can easily build an explicit concept from an implicit one

What we cannot do (without a language extension) is to disable matching of a type to an implicit concept.

The language complexity and the many clever solutions to the problem of switching back and forth between implicit and explicit concepts proposed in the reflector discussion ere truly scary. This is expert-only territory. We seem to have created a language with two more or less viral notions competing in ways that force users to choose between them and to switch between them. This is unacceptable. At best code written by several people will become unreadable (the same text in different places in a program will have different rules for correctness) and much energy and cleverness will have to be expended managing concepts and concept maps. I suspect most people will simply give up and revert to other language features and other languages.

I reluctantly conclude that there can be only one kind of concepts (and thus not problems switching among many). That "kind of concepts" must be what we currently call implicit/**auto**.

Note that people who prefer explicit concept maps can still write them; they just can't force others to do so except where necessary to distinguish semantically differing concepts in a hierarchy.

Negative asserts

For a variety of reasons, several people have suggested "negative assertions or "negative concept maps" to say. This type does not match that concept. For example:

```
!concept_map ForwardIterator<isream_iterator>; // don't use an istream_iterator
                                               // as a ForwardIterator
```

I'm not fundamentally opposed to this idea, but I'm not sure I understand all the implications and (given explicit refinement) I don't have sufficient evidence for a need.

Integrated concept maps

When a type is defined, it is often defined to meet a (pre-defined) specific concept. For example:

```
class X { … };
concept_map SomeConcept<X> { };
```

Immediately writing that concept map checks that our design aim is met (accidental error are caught). Several people (incl., Dave Abrahams, Beman Dawes, and Alisdair Meridith) have observed that it might be convenient to combine those two declarations. For example

```
class X : Someconcept { … };
```

or

```
SomeConcept X { … };                    // credit: Dave Abrahams
```

or

```
class X<SomeConcept> { … };
```

or

class X requires SomeConcept { … }; // credit : Beeman Dawes

I am not convinced that there is a real need for such simplified syntax, but as long as there are no technical problems (e.g. grammar problems) I'm not fundamentally against such a notation. However, I'm not proposing one partially because we already have the non-intrusive, so all we would do would be to save a few keystrokes, and partly because we might create myth that types should be designed primarily to match specific named concepts, which could lead to further emphasis on names f concepts (as opposed to properties of concepts. Of the suggestions above, I prefer the one that explicitly uses **requires** and which can easily be extended to deal with multiple concepts.

Which Library components should be implicit?

Years ago, the effort to "conceptualize the standard library started out with the ideal that "all or most should be explicit. However, by the reasoning above, we should look for standard library concepts that would actually best be explicit. The result of that exercise was interesting.

First, I will ignore the concepts "known to the compiler" since the auto/explicit distinction is irrelevant to those. Of the rest, several of the explicit ones are "magic" in that they receive compiler support and have rules against user-supplied concept maps:

- **True**
- **LvalueReference**
- **RvalueReference**
- **TriviallyDestructible**
- **HasVirtualDestructor**
- **TriviallyCopyConstructible<typename T>**
- **TriviallyCopyAssignable**

Of the rest, it seems that about two thirds are already **auto**.

I find it hard to see more that really need to be explicit. Beman Dawes listed these as currently explicit (I added **InputIterator** and **ForwardIterator**):

- **IntegralLike**
- **ArithmeticLike**
- **Allocator**
- **Container**
- **FrontInsertionContainer**
- **BackInsertionContainer**
- **StackLikeContainer**
- **QueueLikeContainer**

- **InsertionContainer**
- **RangeInsertionContainer**
- **FrontEmplacementContainer**
- **BackEmplacementContainer**
- **EmplacementContainer**
- **Iterator**
- **InputIterator**
- **ForwardIterator**
- **BidirectionalIterator**
- **RandomAccessIterator**
- **Range**

At a glance, I don't see any that are likely to get accidentally matched. All seems to have distinguishing operations. If there are other reasons for one of these concepts to be explicit, we should consider if the reason is fundamental or language technical. If the reason is not fundamental, we must consider the language rule allowing explicit concepts problematic.

So, I propose that all standard library concepts to be implicit and to have **ForwardIterator** explicitly derived from **InputIterator** to avoid the classical mismatch bug.

Type/Concept Matching

Once upon a time, an associated function could match either a member function or a free standing function. That's no longer so. I don't know exactly why this was changed, but I can think of several good technical reasons. However, consider:

```
struct Traverser {
        typedef int* iterator;
        iterator begin();
        iterator end();
};

Traverser trav;
// ... attach trav to a data source ...
for (auto x : trav) ...
```

To my surprise this does not work. Why not? The range-for requires a Range:

```
concept Range<typename T> {
        InputIterator iterator;
        iterator begin(T&);
        iterator end(T&);
}
```

That looks ok: **Range** requires an object of a type with a pair **begin()** and **end()** functions returning an appropriate input iterator and my **Traverser** class does exactly that (just like **std::vector**). The snag is that (despite appearances) Range requires free-standing functions **begin()** and **end()** rather than members. I could rewrite, **Traverser** like this:

```
struct Traverser {
        typedef int* iterator;
};

Traverser::iterator begin(Traverser);
Traverser::iterator end(Traverser);
```

For reasons that will appear quite incomprehensible to an ordinary and reasonable user, we have converted a nice, idiomatic example into a more verbose version with potential overloading problems. This is a black art we do not need.

So can we overcome the language-technical problems and make this **Traverser**/**Range** example work as originally (and naively) written? I think we must or we will have lots of otherwise redundant concept maps needed to overcome this problem. What makes this nasty is that the problem is fundamentally one of language design rooted in (scope and overloading rules) rather than of fundamental needs of programmers.

The original rules for type matching were based on scope (lookup). Let's instead consider what it would take to make the example work whichever of the two ways above was used. That is, what would it take to map the types into the concept? To simplify that discussion let me simplify the notation:

```
concept Range<typename T> {
        InputIterator iterator;
        iterator begin(T&);
        iterator end(T&);
}

struct T1 {
        typedef int* iterator;
        iterator begin();
        iterator end();
};

struct T2 {
        typedef int* iterator;
```

```
};

T2::iterator begin(T2);
T2::iterator end(T2);
```

What would it take to make both **T1** and **T2** match **Range**? For the moment, ignore issues of **T** vs. **T&** arguments in concepts. Given that imagine the following rules:

T1 matches **Range** because

- It has a member type **iterator** as required by **Range**
- It has a (member) function **begin()** that takes an **T1** and returns an **iterator**.
- It has a (member) function **end()** that takes an **T1** and returns an **iterator**.

T2 matches **Range** because

- It has a member type **iterator** as required by **Range**
- There exist a (free-standing) function **begin()** that takes a **T2** and returns an **iterator**.
- There exist a (free-standing) function **end()** that takes a **T2** and returns an **iterator**.

In other words, when trying to match a type to a concept, we consider a type's member function equivalent to a free-standing function with an added first argument. This is a variant of ideas (repeatedly floated by Francis Glassborow and me in the EWG) for unifying function call syntax.

The alternative (status quo) leaves us with a large class of surprising lack of type/concept matches and forces us to write many otherwise unnecessary concept maps.

Conclusions

We must minimize the explicit use of concept maps to make concepts usable by "ordinary programmers." In particular, concept maps must be implicit, classes that "obviously match" concepts must match (rather than forcing people to write concept maps for purely language technical reasons), and the standard library can and must set a good example by using explicit concept maps only as appropriate.

The use of concepts is supposed to help people write and use a wide range of templates. The current definition of concept maps and the philosophy that seems to go with them makes it harder.

Addressing this is important. I suspect that the alternative is widespread disuse of concepts and libraries using concepts. I would consider that a major failure of C++0x.

Summary of proposals

Language proposal (Please note my use of the singular. I consider this one proposal to address a serious problem, not a set of unrelated proposals to address as variety of weakly related minor problems):

1. Allow refinement to be **explicit**
2. Make all concepts implicit (i.e. remove explicit concepts from C++0x)
3. Allow a concept to match a constrained template argument
4. Allow both free-standing and member functions to match a concept

Standard-library proposal:

1. All standard library concepts should be implicit
2. The following standard library concepts are explicitly refined:
   a. **ForwardIterator** is explicitly refined from **InputIterator**
   b. **TriviallyDefaultConstructible** is explicitly refined from **DefaultConstructible**
3. Remove concept maps made redundant the member function matching rule (1 above)

Acknowledgements

Thanks to the many contributors in the "Joe Coder" thread and to the many similar discussions over the years and especially lately.

Proposal text

14.10.1 Concept definitions [concept.def]

In [1] remove **auto**opt from the grammar

Remove [4] which defines auto concept

14.10.2 Concept maps [concept.map]

In [11] replace "an **auto** concept" with "a concept"

In [13] replace "A concept map or concept map template shall be defined" with "A concept map or concept map template shall (explicitly or implicitly ([concept.map][11]) be defined"

Globally replace "**auto concept**" with "**concept**"

14.10.3 Concept refinement [concept.refine]

In [1] replace

refinement-specifier:
*concept-instance-alias-def*opt ::opt *nested-name-specifier*opt *concept-id*

With

refinement-specifier:
: explicit$_{opt}$ *concept-instance-alias-def*$_{opt}$ ::$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*

Add a paragraph [6]

If a refinement is declared explicit an operation from the refined concept may not be substituted for the equivalent operation for the less refined concept and an explicit concept map is required for a type to match the more refined concept. A type X that matches both concepts is considered to have matched only the less refined concept [Comment the more refined concept is assumed to have more semantic constraints – end Comment] unless an explicit concept map has been defined for X and the more refined concept (in which case X is considered a match for the more refined concept only).

[Example:

```
concept C1<typename T> { void f(T&);  }

concept C2<typename T> : explicit C<T> { void f(T&); }

template<C1 T> void algo(T& t)

{

        f(t);       // will never use C2's f()

}

struct S { … };

void f(S&);

S s;

f(s);      // ok: S matches C1

concept_map C2<X> { };

f(s);      // ok: S matches C2
```

- End example]

I do not have specific wording for the "implementation leakage" issue, but somewhere (e.g. as part of CWG issue 870) resolve the rules for concept maps to allow a use to say nonintrusively:

```
concept AB<typename T> {
        void a(T&);
        void b(T&);
};

concept A<typename T> {
        void a(T&);
```

```
};

template<AB T> concept_map A<T> { }  // every AB is an A

template<A T> void g(T);

template<AB T> void f(T t)
{
        g(t);   // ok
}
```

- end example]

14.11.4 Instantiation of constrained templates [temp.constrained.inst]

Replace the first bullet item of [3] by

- If the seed is a non-member function, the instantiated form is a call to the associated function candidate set.

[*Example:*

```
concept F<typename T> {
        T::T();
        void f(T const&);
}
template<typename T> requires F<T>

void g(T const& x) {
        f(x);       // calls F<T>::f. When instantiated with T=X, calls #1
        f(T());     // calls F<T>::f. When instantiated with T=X, calls #2
}
struct X {};
void f(X const&);  // #1
void f(X&&);       // #2
concept_map F<X> { }      // associated function candidate set for
                          // f(X const&) contains #1 and #2, seed is #1
void h(X const& x) {
        g(x);
}
```

—*end example*]

- If the seed is a member function, the instantiated form is a call to the associated function candidate set. The member function may be invoked either using the functional notation (f(x)) notation or the member function notation (x.f()) – a call using the functional notation is mapped to a member function call by using its first argument as the object; for example f(x,y,z) is interpreted as x.f(x,y).

[*Example:*

```
concept F<typename T> {
        T::T();
```

```
        void f();
    }
    template<typename T> requires F<T>

    void g(T const& x) {
        f(x);       // calls F<T>::f. When instantiated with T=X, calls #1
        f(T());     // calls F<T>::f. When instantiated with T=X, calls #2
        x.f();      // calls F<T>::f. When instantiated with T=X, calls #1
    }
    struct X {
        void f(X const&);  // #1
        void f(X&&);       // #2
    }

    concept_map F<X> { }    // associated function candidate set for
                            // f(X const&) contains #1 and #2, seed is #1
    void h(X const& x) {
        g(x);
    }
```

—*end example*]

Remove **18.9.3 Initializer list concept maps [support.initlist.concept**] (it has become redundant)

Remove from **23.2.6 Container concepts [container.concepts]** [1] (it has become redundant):

```
template<Container C> concept_map Range<C> see below;
template<Container C> concept_map Range<const C> see below;
```

Remove from **23.2.6.3 Container concept maps [container.concepts.maps]** (it has become redundant)**:**

```
template<Container C>
concept_map Range<C> {
typedef C::iterator iterator;
iterator begin(C& c) { return Container<C>::begin(c); }
iterator end(C& c) { return Container<C>::end(c); }
}
template<Container C>
concept_map Range<const C> {
typedef C::const_iterator iterator;
iterator begin(const C& c) { return Container<C>::begin(c); }
iterator end(const C& c) { return Container<C>::end(c); }
}
```

13 *Note:* these concept_maps adapt any type that meets the requirements of Container to the Range concept.