Concepts for C++1y: The Challenge

(Why C++0x Concepts failed and what to do about that)

Bjarne Stroustrup

Texas A&M University

http://www.research.att.com/~bs

Abstract

- One of the major goals of the language part of C++0x is to provide better support for generic programming. I see that as a key to both better library development and to better support of "C++ novices" of all backgrounds.
- I presents the fundamental mechanisms and basic idioms of C++ generic programming using C++98, then I present the support for template argument requirement specification for template arguments designed for C++0x. I critique that design based on experience gained from its specification, implementation, and use leading to some ideas for a redesign.

Overview

- The problem
  - Generic programming in C++98
- Constraints on a solution
  - The real world is messy
- C++0x concepts
  - Getting too close to type classes
- Ideas for C++1y concepts
  - A concept is primarily a predicate

Initial aims

- User-defined parameterized containers
  - **vector<T>**, **list<T>**, **map<K,V>**, **array<T,int>**, etc.
- User-defined operations on parameterized containers
  - **sort(c)**, **sort(c,cmp)**, **find(c,v)**, etc.
- Outcompete built-in arrays
  - Static type safety (incl. optional(?) range checking)
  - Simpler use
  - Absolute performance: Not a byte and not a cycle more
- Don’t break C++’s separate compilation model

- 1980-1988: macro tricks
- 1988-: unconstrained templates
  - thoughts and experiments with constraints (e.g. D&E)

Templates

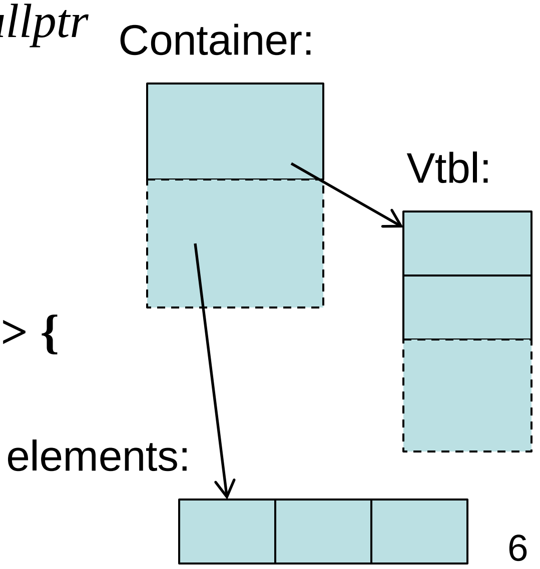
- C++98 can parameterize functions and classes with types and values (typically integers or operations):
  - **template<typename T>** means "for all types **T**"
  - **template<int N>** means "for all integer values **N**"
  - For historical reasons **template<class T>** is a synonym for **template<typename T>**
- For example

```
template<typename T, int N>
class Buffer {
public:
    // interface
private:
    T a[N];     // represented as an array of N Ts
};
Buffer<char,1024> buf;          // a buffer of 1024 characters
```

Statically typed conventional OO

- Use virtual functions
  - Provides separate compilation and a simple ABI
    - indirect function call for iteration and access (far too slow)
    - Containers must be on free store (close to unmanageable without GC)

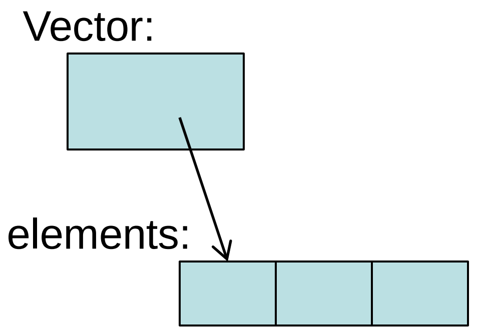
```
template<class T> class Container {          // just an interface
public:
    virtual T* begin();   // pointer to first element or nullptr
    virtual T& operator[](int);     // subscripting
    // …
};

template<class T> class Vector: public Container<T> {
    // representation (data)
    // functions (override begin() and operator[]() …)
};
```



Statically typed; not conventional OO

- Containers and iterators are concrete types
  - Fast and compact:
    - Containers are mostly on stack or statically allocated (elements usually on heap)
    - Operations are easily inlined

```
template<class T> class vector {
public:
    vector();    // constructors acquire memory and if necessary initializes elements
    ~vector();   // destructor releases memory and if necessary destroy elements
    iterator<T> begin();     // iterator to first element
    iterator<T> end();       // iterator to last element
    T& operator[](int);      // subscripting
    // …
private:
    // representation
};
```



Find an element that equals a value

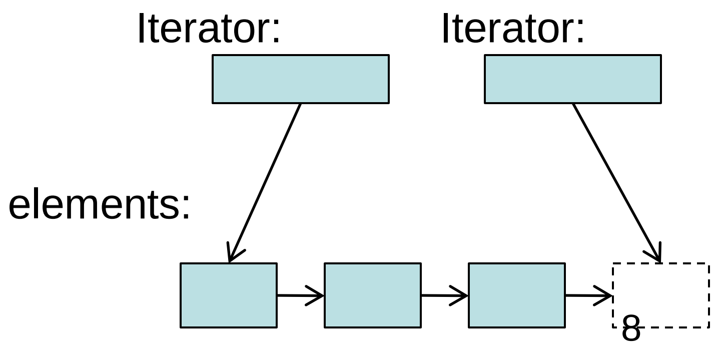
- Abstract over different kinds of containers using iterators

```
template<typename Iter, typename Val>
Iter find(Iter p, Iter q, Val v)      // find v in [p:q)
{
    while (p!=q && !(*p==v)) ++p;
    return p;
}

vector<int> v = { 1,2,3,4,5,6,7};
list<string> lst = {"Strachey", "Richards", "Ritchie"};

auto p = find(v.begin(), v.end(), 99);
if (p!=v.end())  // found  *p==99

auto q = find(lst.begin(), lst.end(), "Wirth");
if (q!=lst.end())  // found  *q=="Wirth"
```

Operations on Iter

Operation on Val

Operations dependent on Iter and Val



Find an element that matches a predicate

```
template<typename Iter, typename Cmp>
Iter find_if(Iter p, Iter q, Cmp cmp)        // find cmp(element) in [p:q)
{
    while (p!=q && !cmp(*p)) ++p;                          Operations on Iter
    return p;
}                                            Operations dependent on Iter and Val

vector<int> v = { 1,2,3,4,5,6,7};
list<string> lst = {"Strachey", "Richards", "Ritchie"};

auto p = find_if(v.begin(), v.end(), Less_than(42));   // function object: compare to 42
if (p!=v.end())  // found  *p<42

auto q = find_if(lst.begin(), lst.end(),
                 [](string s) { return s<="Wirth"; });   // lambda expression:
                                                         // compare element to "Wirth"
if (q!=lst.end())  // found  *q<="Wirth"
```

Parasol
Smarter computing.
Texas A&M University

Coping with irregularity

- How do you dispatch (select an operation) based on type?
  - Template instantiation
  - Overloading
- How do you make separately-developed types appear identical to a single piece of code?
  - Traits
- How do you make a single interface to differing types
  - Specialization

Overloading and instantiation

- Names can be overloaded freely
  - As long as we can disambiguate by syntax or types

```
template<typename T>
    T operator*(T a ,T b) { return a*=b; }  // multiply (binary *)

template<typename S>
    complex<S> operator*(complex<S>, complex<S>);    // definition elsewhere

template<typename S>
    complex<S> operator*(S, complex<S>);    // definition elsewhere

template<typename Iter>
    typename Iter::value_type& operator*(Iter);      // dereference (unary *)
```

Associated types

- Many generic algorithms need “associated types”
  - E.g. what is the type of an element pointed to by an iterator?

```
template<typename FwdIter>               // sort a list
void fast_sort(FwdIter first, FwdIter last) {
    using Elem = typename FwdIter::value_type;  // requires member type
    vector<Elem> v(first, last);
    sort(v.begin(), v.end());
    copy(v.begin(), v.end(), first);
}
```

Traits

- But what if a type doesn’t have a required member type (“associate type”)?
  - E.g., **List<T>::iterator** may be a **Node*** (a pointer) which does not have a **value_type**
  - Use a “trait” (in use since 1993 and earlier internally)

```
template<typename FwdIter>     // sort a list
void fast_sort(FwdIter first, FwdIter last) {
    using Elem = typename iterator_traits<FwdIter>::value_type;
    vector<Elem> v(first, last);
    sort(v.begin(), v.end());
    copy(v.begin(), v.end(), first);
}
```

Traits

- How to non-intrusively add properties to types
  - Add/change properties after the definition of a type (or algorithm)

```
template<class Iterator> struct iterator_traits;  // general template (never used)

template<class T> struct iterator_traits<T*> {  // specialization for all pointers
    using difference_type = ptrdiff_t;
    using value_type  = T;              // the value_type of a T* is T
    using pointer = T*;
    using reference = T&;
    using iterator_category = random_access_iterator_tag;
};
```

Tag dispatch

- Compile-time selection based on trait

```
template<class Iter> void reverse(Iter first, Iter last)
{
    reverse_helper(first, last, iterator_traits<Iter>::iterator_catagory);
}

template<class For> void reverse_helper(Iter first, Iter last, forward_iterator_tag)
{ /* use only forward traversal */ }

template<class R> void reverse_helper(R first, R last, random_access_iterator_tag)
{
    if (last-first>1) {
            swap(*first,*--last);
            reverse_helper(--first, last, random_access_iterator_tag);
    }
}
```

Parasol
Smarter computing.
Texas A&M University

A problem

```
template<typename Iter, class Val>
Iter find(Iter p, Iter q, Val v)       // find v in [p:q)
{
    while (p!=q && !(*p==v)) ++p;
    return p;
}

auto p = find(0,1000, 99);                      //  silly error: int has no prefix *

int a[] = { 1,2,3,4,5,6,7,8 };
auto q = find_if(a, a+8, less<int>());          // error: less is a binary operation
```

- These problems are found,
  - But far too late (at link time)
  - Error messages are spectacularly bad

Summary

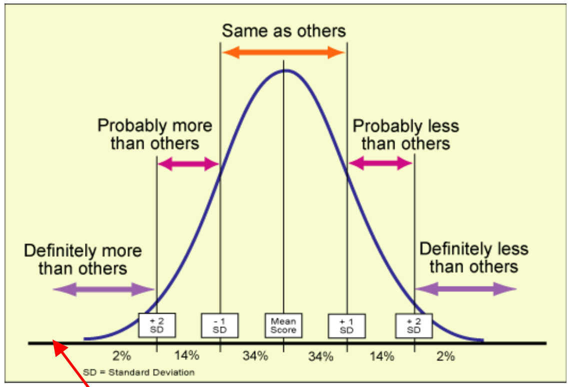
- Templates are flexible, general, type safe, and efficient
- Templates are widely used
  - Incl. high-end computing and embedded systems
- BUT
  - Templates are fully checked only at instantiation time
    - That’s far too late
  - There is no reasonable and general way of expressing constraints on a set of template arguments
    - Brittle: spectacularly bad error messages
    - Poor overloading – leading to verbosity
    - Much undisciplined hacking
    - Much spectacularly obscure code
    - Verbose in places
  - There is no separate compilation of templates
    - No templates in ABI

Constraints on any solution

- Billions of lines of C++
- Millions of C++ Programmers

- Don’t break their code
- Make it attractive for them to learn and use

- Only about 50% of the code and about 50% of the programmers are above industry averages

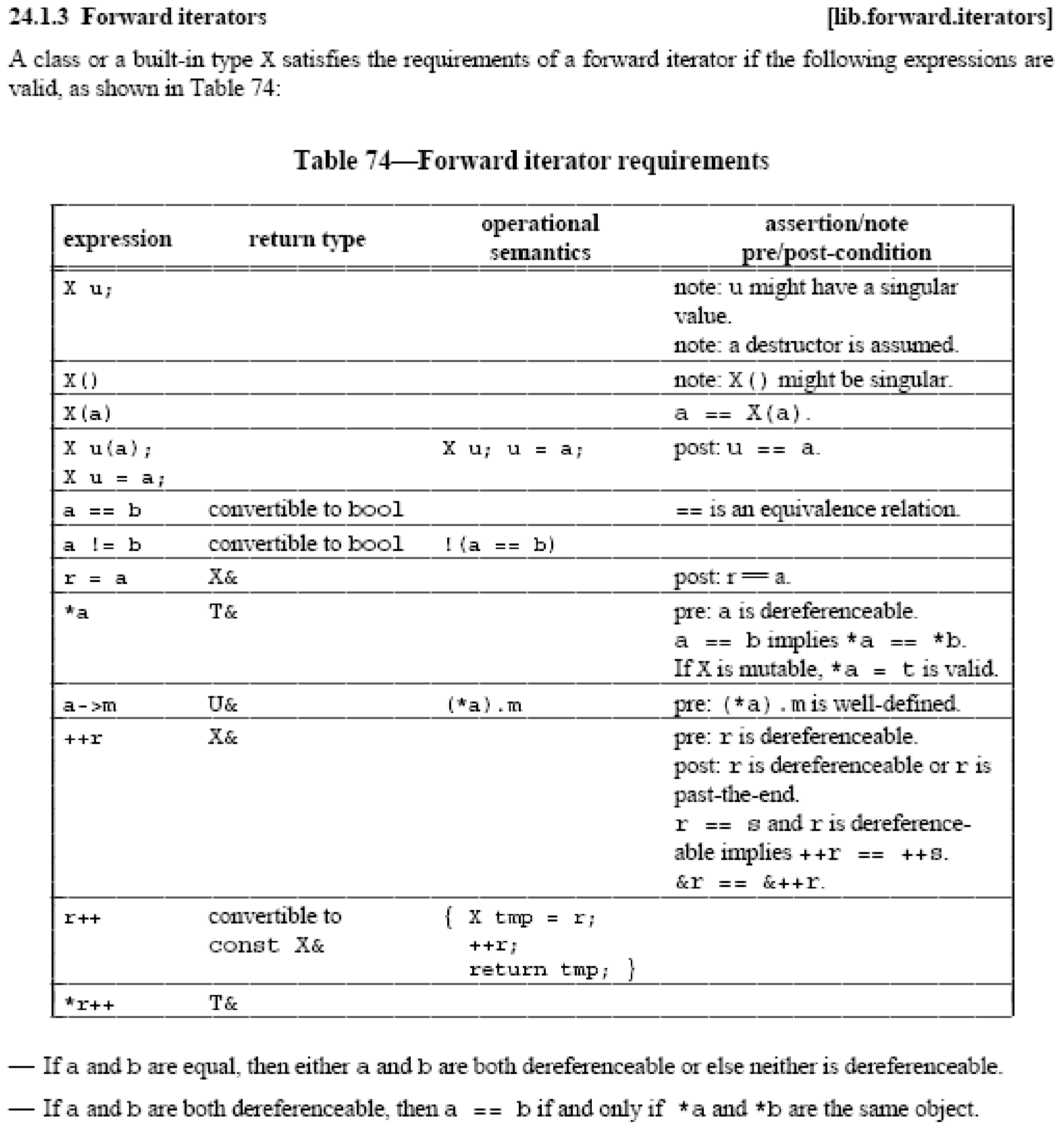


You are here

Aims for a solution
("industry demands")

- "Seemless" replacement of standard library use
  - i.e. recompile with concept-enabled standard library
    - we did that for C++0x for significant amounts of C++98 code
- Max 20% increase in compile time
  - My ideal is a decrease in compile time due to simplification and separate compilation
- No significant increase in run time
  - We did that for C++0x concepts (<5% but dues to heroic efforts)
- No significant increase in code size
  - Source and object code
  - Beware of increases in header files
- No significant re-training of programmers
  - E.g. don't require understanding of type theory

What we have in C++98



24.1.3 Forward iterators [lib.forward.iterators]

A class or a built-in type X satisfies the requirements of a forward iterator if the following expressions are valid, as shown in Table 74:

Table 74—Forward iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X u;`			note: u might have a singular value. note: a destructor is assumed.
`X()`			note: `X()` might be singular.
`X(a)`			`a == X(a)`.

`X u(a);`
`X u = a;`
		`X u; u = a;`	post: `u == a`.

`a == b`	convertible to `bool`		`==` is an equivalence relation.
`a != b`	convertible to `bool`	`!(a == b)`	
`r = a`	`X&`		post: r == a.
`*a`	`T&`		pre: `a` is dereferenceable. `a == b` implies `*a == *b`. If X is mutable, `*a = t` is valid.
`a->m`	`U&`	`(*a).m`	pre: `(*a).m` is well-defined.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. `r == s` and `r` is dereference-able implies `++r == ++s`. `&r == &++r`.

`r++`	convertible to `const X&`	`{ X tmp = r;`
		`++r;`
		`return tmp; }`	

`*r++`	`T&`		

— If `a` and `b` are equal, then either `a` and `b` are both dereferenceable or else neither is dereferenceable.

— If `a` and `b` are both dereferenceable, then `a == b` if and only if `*a` and `*b` are the same object.

C++0x Concepts

- "a type system for C++ types"
  - and for relationships among types
  - and for integers, operations, etc.
- Based on
  - Search for solutions from 1985 onwards
    - Stroustrup (see D&E)
  - Lobbying and ideas for language support by Alex Stepanov
  - Analysis of design alternatives
    - 2003 papers (Stroustrup, Dos Reis)
  - Designs by Dos Reis, Gregor, Siek, Stroustrup, …
    - Many WG21 documents
  - Academic papers:
    - POPL 2006 paper, OOPSLA 2006 paper
  - Experimental implementations (Gregor, Dos Reis)
  - Experimental versions of libraries (Gregor, Siek, …)

Checking of uses

- The checking of use happens immediately at the call site and uses only the declaration

```
template<Forward_iterator For, Value_type V>
    requires Assignable<For::value_type,V>
void fill(For first, For last, const V& v);    // <<< just a declaration, not definition

int i = 0;
int j = 9;
fill(i, j, 99);    // error: int is not a Forward_iterator (no unary *)

int* p= &v[0];
int* q = &v[9];
fill(p, q, 99);   // ok: int* is a Forward_iterator
```

Checking of definitions

- Checking at the point of definition happens immediately at the definition site and involves only the definition

```
template<Forward_iterator For, Value_type V>
    requires Assignable<For::value_type,V>
void fill(For first, For last, const V& v)
{
    while (first!=last) {
      *first = v;
      first=first+1;          // error: + not defined for Forward_iterator
                              // (instead: use ++first)
    }
}
```

Concept maps ("models")

- How can we non-intrusively add a property to a type

```
template<Value_type T>
concept_map Forward_iterator<T*> {            // T*'s value_type is T
    using value_type = T;
};
```

- When we use **T*** as a **Forward_Iterator,** the **value_type** of **T*** is **T**
- **value_type** is an associated type of **Forward_iterator**

- Concept maps can be seen as
  - A more general and elegant version of traits
  - Explicit modeling statements

Expressiveness

- Simplify notation through overloading:
  - Currently, this requires a mess of helper functions and traits

```
template<Container C> void sort(Container&);
template<Random_access_iterator R> void sort(R,R);
template<Container C, Comparator<C::value_type> Cmp>
    void sort(Container&, Cmp);
…
```

```
void f(vector<int>& vi, list<int>& lst, Fct f)
{
        sort(vi);                       // sort container (vector)
        sort(vi, f);                    // sort container (vector) using f
        sort(lst);                      // sort container (list)
        sort(lst, f);                   // sort container (list) using f
        sort(vi.begin(), vi.end());     // sort sequence
        sort(vi.begin(), vi.end(), f);  // sort sequence using f
}
```

Parasol
Smarter computing.
Texas A&M University

Algorithmic performance

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

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

template<Forward_iterator  Iter>
        // note: no mention of Random_access_iterator, no “traits trickery”
    void mumble(Iter p, int n)
    {
        // …
        advance(p, n / 2);
        // …
    }

    vector<int> v = { 904, 47, 364, 652, 589, 5, 35, 124 };
    mumble(v.begin(), 4);    // invoke RandomAccess’ advance()
```

- Necessary? (This destroys modular type checking)
  - Widely claimed essential for STL (and other) performance

But isn’t this “constrained genericity”?

- As in Eiffel, Java, or C#?
- Or Haskell type classes?

- No, we want something that
  - does not require a hierarchy
    - Is not glorified abstract classes
    - Does not require indirect function calls to implement
    - Handles built-in types perfectly
      - Does not wrap/box values of built-in types
    - Does not introduce extra indirections (pointers/references)
  - handles values and operations as arguments
    - Template meta-programming, generative programming
  - simplifies non-trivial compile-time computation

Problems with C++0x concepts

- Explicit vs. implicit (automatic) concepts (“modeling”)
  - What happens if two a type matches two concepts?
    - Detection and disambiguation
  - Is widespread/universal explicit modeling good for usability?
    - My answer: no
    - The standard library ended up with 80% implicit modeling (when rewritten by firm believers in explicit modeling) only two purely disambiguating concept maps are necessary)
    - Does **draw()** paint on the screen or pull a gun?
- How to handle conversions and intermediate types
  - E.g. **x=++*p** and **a=b+c*d**
- Complexity
  - Specification, implementation, and use
    - 71 page standards text
    - 150 standard-library concepts

What we want to make obsolete

- Traits
  - Helper/dispatch functions
- SFINAE function overloading
  - If you know what that means, you have learned too much ☹

So, what are concepts?

- Compile-time predicates on
  - types
    - C1<T>
    - For expressing "type of type" as needed for template argument passing
  - combinations of types
    - C2<T1,T2,T3>
    - For expressing relationships among types as needed by algorithms
  - combinations of types, integers, operations, etc.
    - C3<T1,Plus,0>
    - For expressing relationships as by general mathematical notions and to support "advanced template uses"
- Specify requirements on types
- Can specify semantic properties

“Strong signatures”

- Consider how to handle ***++p**

```
concept Input_iterator<typename Iter> {  // simplified
    typename value_type;
    typename reference;
    typename increment_result;
    requires Same_type<   // too strong
                Has_dereference<increment_result>::result_type,
                const value_type&
    >;
    reference operator*();
    increment_result operator ++();
}
```

- Far too strong, what about
  - Conversions
  - Proxies

Parasol
Smarter computing.
Texas A&M University

“Strong signatures”

- Consider how to handle *++p

```
concept Input_iterator<typename Iter> {  // simplified
    typename value_type;
    typename reference;
    typename increment_result;
    requires Convertible<
                Has_dereference<increment_result>::result_type,
                const value_type&
    >;
    reference operator*();
    increment_result operator ++();
}
```

- But you still have to specify all the types somewhere
- Too complex – unmanageable

Constraints

- Instead of specifying types that have to be defined elsewhere by users, leave intermediate types for the compiler to deduce

```
typename value_type;
typename reference;
typename increment_result;
```

- That means that you cannot build a table of fully typed parametric operations to represent a concept without introducing “artificial functions”

```
reference operator*();
increment_result operator ++();
```

- We must see the set of “signatures” as a set of equations with (some of) their argument and result types as variables
  - A concept is a predicate; we must solve its set of constraints to evaluate it (does the set of constraints have a unique solution for the subset of types and values specified by the programmer?)

Weak signatures or use-patterns

- An alternate notation (POPL'06)

```
concept Input_iterator<typename Iter> {  // simplified
    Expr<Iter> p;                        // notation to introduce a name
    Iter::value_type x = *++p;           // a use pattern
}
```

- Every expression defines a set of constraints
- The use-pattern notation and conventional signature notations can be logically equivalent
  - I think
  - Which notation is easier for "ordinary programmers" to use in large code bases?

Concepts combine with &&, ||, and !

- Move or copy:

```
template<Input_iterator In, Output_iterator Out>
    requires Copyable<In::value_type, Out<value_type>
                    || Movable<In::value_type, Out<value_type>
Out copy(In first, In last, Out res);
```

- Alternatively:

```
concept Assignable<typename T>
    requires Copyable<In::value_type, Out<value_type>
                    || Movable<In::value_type, Out<value_type>
{}

template<Input_iterator In, Output_iterator Out>
    requires Assignable<In::value_type, Out<value_type>
Out copy(In first, In last, Out res);
```

Concepts references

- http://www2.research.att.com/~bs/papers.html
- B. Stroustrup: The C++0x "Remove Concepts" Decision. Dr.Dobb's Journal. 2009.
- B. Stroustrup: Evolving a language in and for the real world: C++ 1991-2006. HOPL-III. 2007.
- D. Gregor, J. Jarvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. OOPSLA'06,
- Gabriel Dos Reis and Bjarne Stroustrup: Specifying C++ Concepts. POPL'06.
- G. Dos Reis, B. Stroustrup, A. Meredith: Axioms: Semantics Aspects of C++ Concepts. N2887. 2009 .
- B. Stroustrup Simplifying the use of concepts. N2906. 2009. The paper that caused concepts to be postponed.
- B. Stroustrup and G. Dos Reis: An analysis of concept intersection. N2221. 2007.
- B. Stroustrup: Abstraction and the C++ machine model. Proc. ICESS'04.

The original design criteria discussion

- Bjarne Stroustrup and Gabriel Dos Reis: Concepts -- Syntax and composition. October 2003. An early discussion of how to express concept checking in C++.
- Bjarne Stroustrup and Gabriel Dos Reis: Concepts -- Design choices for template argument checking. October 2003. An early discussion of design criteria for concepts for C++.
- Bjarne Stroustrup: Concept checking -- A more abstract complement to type checking. October 2003. A discussion of models of concept checking.

Concepts and type classes

- IMO concepts are not type classes
  - Even though they serve similar purposes
  - They have different strength, weaknesses, and idiomatic uses
- http://portal.acm.org/citation.cfm?id=1411324
- http://wiki.portal.chalmers.se/cse/pmwiki.php/FP/ConceptsTypeClasses
- http://bartoszmilewski.wordpress.com/2010/11/29/understanding-c-concepts-through-haskell-type-classes/
- http://haskell.org/haskellwiki/Simonpj/Talk:OutsideIn GDR: this paper (draft) is related to the kind of constraints I'm solving with Liz when everything is translated in an intermediate representation inside the compiler. That is the conclusion Simon and I came to. Of course, the devil is in the details;
- http://www.haskell.org/haskellwiki/OOP_vs_type_classes