Better support for generic programming

Bjarne Stroustrup
Texas A&M University
(and AT&T Labs – Research)
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. The focus of the talk will be the proposals currently under discussion in the standards committee, such as concepts, generalized initialization, auto, template aliases. The aim is to consider such features as a whole supporting programming techniques, rather that as just a collection of neat features to be considered one after another.

2

Overview

- Problems, aims, and limitations
- Minor improvements
  - – Auto, decltype, and template alias
- Concepts
- Initialization
  - – Sequence, forwarding, inheriting, literal, rvalue, …

Problems

- Generic programming has been so successful it outstripped its language support: The language is straining
  - – Late checking
    - At template instantiation time
  - – Poor error messages
    - Amazingly so
      - – Pages!
  - – Too many clever tricks and workarounds
    - Works beautifully for correct code
      - – Uncompromising performance is usually achieved
        - » After much effort
    - Users are often totally baffled by simple errors
    - The notation can be very verbose
      - – Pages for things that's logically simple

What's wrong?

- Poor separation between template definition and template arguments
  - But that's essential for optimal code
  - But that's essential for flexible composition
  - So we must **improve separation** as much as possible without breaking what's essential
  - A source of poor build times – not addressed here
- We have to say too much (explicitly)
  - So we must find ways to **abbreviate and make implicit**
- The template name lookup rules are too complex
  - But we can't break masses of existing code
  - So find ways of saying things that **avoid the complex rules**

5

What's right?

- Parameterization doesn't require hierarchy
  - Less foresight required
    - Handles separately developed code
  - Handles built-in types beautifully
- Parameterization with non-types
  - Notably integers
- Uncompromised efficiency
  - Near-perfect inlining
- Compile-time evaluation
  - Template instantiation is Turing complete

We try to strengthen and enhance what works well

6

Aims

- Support generic programming
- Support programming using templates in general
- Not add a bunch of unrelated “neat features” one by one

7

Minor improvements

(all approved in principle)

- **auto**
  - To avoid repetition (verbosity)
  - To avoid redundant explicit mention of type names
    - Mentioning things that need not be mentioned can be a maintenance hazard
- **decltype**
  - To solve part of the forwarding problem
- Template aliases
  - To name partially bound templates (Currying)

8

Auto

- Deduce the type of a variable from its initializer

```
template<class T> void f(const vector<T>& v)
{
   for (auto p = v.begin(); p!=v.end(); ++p)
            do_something(p);
}
```

Saves mentioning vector<T>::const_iterator

```
template<class T, class U> void f(T t, U u)
{
   auto x = t*u;
   // ...
}
```

Saves naming deduced type

9

Decltype

- A forwarding problem

```
template<class F, class A> result_of<F(A)> trace(F f, A a)
{
        cout << "call f\n";
        return f(a);
};
```

- We can't define **result_of**<T> perfectly generally and simply (approximations abound)

10

Decltype

- A partial solution

```
template<class F, class A> decltype(f(a)) trace(F f, A a)
{
        cout << "call f\n";
        return f(a);
};
```

- **decltype(expr)** means “the type declared for **expr**”
  - Note this preserves references (**auto** doesn’t and mustn’t)
- But how can we use f and a before they are declared?
  - we can’t

11

Decltype

- A solution
  - Move the return type after the arguments so that we can use the arguments to express it
  - Use **auto** to mean “type follows (deduced from arguments)”

```
template<class F, class A>
auto trace(F f, A a) -> decltype(f(a))
{
        cout << "call f\n";
        return f(a);
}
```

12

Template Aliases
(current workarounds)

- Aliases for templates are widely needed and used
  - but “odd”, verbose, and not general

```
template<class T> struct MyAllocator {
      // ...
};

template<class T> struct Vec {
      typedef std::vector<T, MyAllocator<T> > type; // alias
};

Vec<int>::type v(29);       // slightly verbose, but works

template<class T> void f (typename Vec<T>::type& v) { /* ... */ }

Vec<int>::type v;
 f(v);   // oops: doesn’t work
```

Template Aliases

- Formerly called template typedefs:

```
template<class T>
   using Vec = std::vector<T, MyAllocator<T> >;

Vec<int> v(29);

template<class T> void f(Vec<T>& v) { /* ... */ }

f(v);              // works nicely
```

Concepts

(still in the design stage)

- Aims and limitations
- Use of concepts in code
- Implementation of (library) facilities using concepts
- Implementation of concepts in compilers

- Based on
  - Primarily: analysis of design alternatives and design by Stroustrup & Dos Reis ("Texas proposal")
    - Three 2003 papers, April 2005 paper
  - Secondarily: Siek, et al ("Indiana proposal")
    - February 2005 paper

15

Aims

- Perfect separate checking of template definitions and template uses
  - Implying radically better error messages
- Simplify all major current template programming techniques
  - Can any part of template meta-programming be better supported?
- Simple tasks are expressed simply
  - close to a logical minimum
- No performance degradation compared to C++ that doesn't use concepts
- No loss of expressiveness compared to current template programming techniques
- Relatively easy implementation within the current implementation frameworks

16

Aims

- A type system for C++ types
  - Note that C++ (like C) use “comparable to”, “assignable to”, etc. far more than “same type”
  - Note that C++ in general and generic programming styles in particular rely heavily on overloading (not just “object-oriented overriding”)
  - Classical type systems (Hindley-Milner-Damas) as found in Haskell, ML, etc. do not handle overloading
- Put to sleep the old “C++ templates are just macros” canard
- Improve the standard library (and all code like it)
  - Important the STL is not the only test case (Algebra is a useful second)
  - By moving informal requirements into code
- Not to fall into the old type hierarchy trap
  - Many uses of generic programming succeeds compared to object-oriented approaches because they don’t require pre-definition of hierarchies
    - If you want abstract classes, you know where to find them
- Preserve all existing template code as correct

17

Example

```
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; }
}

int i = 0;
int j = 9;
fill(i, j, 9.9);        // error: int is not a Forward_iterator

int* p= &v[0];
int* q = &v[9];
fill(p, q, 9.9);        // ok
```

18

Example

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

```
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);        // error: int is not a Forward_iterator

int* p= &v[0];
int* q = &v[9];
fill(p, q, 9.9);       // ok
```

19

Alternate (explicit predicate) notation

A “concepts” is a predicate on one or more types
(or types and integer values)

```
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; }
}
```

template<class T> means “for all types **T**”

template<C T> means “for all types **T,** such that **C<T>**”

20

Example

- A template definition can be checked in isolation from its uses
- In a template definition you can use only the operations defined for the concept in the way they are specified in the concept

```
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 = first+1;     // error: + not defined for Forward_iterator
    }
}
```

21

Ever wanted to do something like this?

(this doesn't mean what it appears to say)

```
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);
```

22

Overloading based on concepts

```
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
   }
```

- Currently, this requires a mess of helper functions and traits
  - In this case some of the traits must be explicit (user visible)

23

Overloading on concepts

```
template<Randon_access_iterator Iter>
   void sort(Iter first, Iter last);
template<Randon_access_iterator Iter, Compare Comp>
         where Assignable<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 Assignable<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 Assignable<Comp::argument_type, Cont::element_type>
   void sort(Cont c, Comp comp);
```

24

Another example

(preserve performance)

```
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)
{
    advance(ld.begin(), 7);        // Forward_iterator version
    advance(vi.begin(), 3);        // Random_iterator version
}
```

25

Yet another example

(fix standard library hack)

```
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> v1(100,1);     // call 1st constructor
int* p = …
int* q = …
vector<int> v2(p,q);       // call 2nd constructor
```

- Important principle (currently violated):
- the C++ standard library should be written in C++
  - and preferably reasonably obvious and good C++ because people do read it and copy its style

26

Defining concepts

(my favorite alternative: usage patterns)

```
concept Forward_iterator<class Iter>{
    Iter p;                    // uninitialized (new meaning)
    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(p==q);                // equality comparisons,  result convertible to bool
    bool(p!=q);

    Value_type Iter::value_type;   // Iter has a member type value_type,
                                   // which is a Value_type
    Iter::value_type = *p;         // *p is an lvalue of Iter's value type
    *p = v;
};
```

27

Concepts and abstract signatures

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

Internally handled by introducing three additional types A1, A2, and R:

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

28

Defining concepts

(an alternative: abstract/pseudo signatures)

```
concept LessThanComparable<class T> {
    bool operator<(T,T);  // Note: not a function declaration (new syntax)
                          // means < can accept two Ts
                          // (somehow: conversions, const, references etc.)
                          // and the result can be converted to bool
};
```

Could be internally handled by introducing two additional types A and R:

```
T -> A
Operator<(A,A) -> R
R -> bool
```

Using a type (obvious match of concept)

```
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);

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

Static asserts are allowed but not required

- I can assert that **Ptr_to_int** has the syntactic properties required by **Forward_iterator** and that it's semantics is correct

```
static_assert Forward_iterator<Ptr_to_int>;          // optional
```

- This is a critical and difficult point
- The Indiana group proposes to make such asserts compulsory

31

Using a type (not so obvious match of concept)

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

- Obviously, we want an **int*** to be a **Forward_iterator**
  - But what about the member type **value_type**?

32

Explicit concept asserts

- when used as an argument for a **Forward_iterator** concept parameter, **value_type** should be considered a member of **T*** with the "value" **T.**
- We need some syntax for that:

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


// clearer, but would involve syntax extensions
static_assert template<Value_type T>  Forward_iterator<T*> {
    using T*::value_type = T;
};
```

33

We can't require explicit asserts

(major topic under discussion)

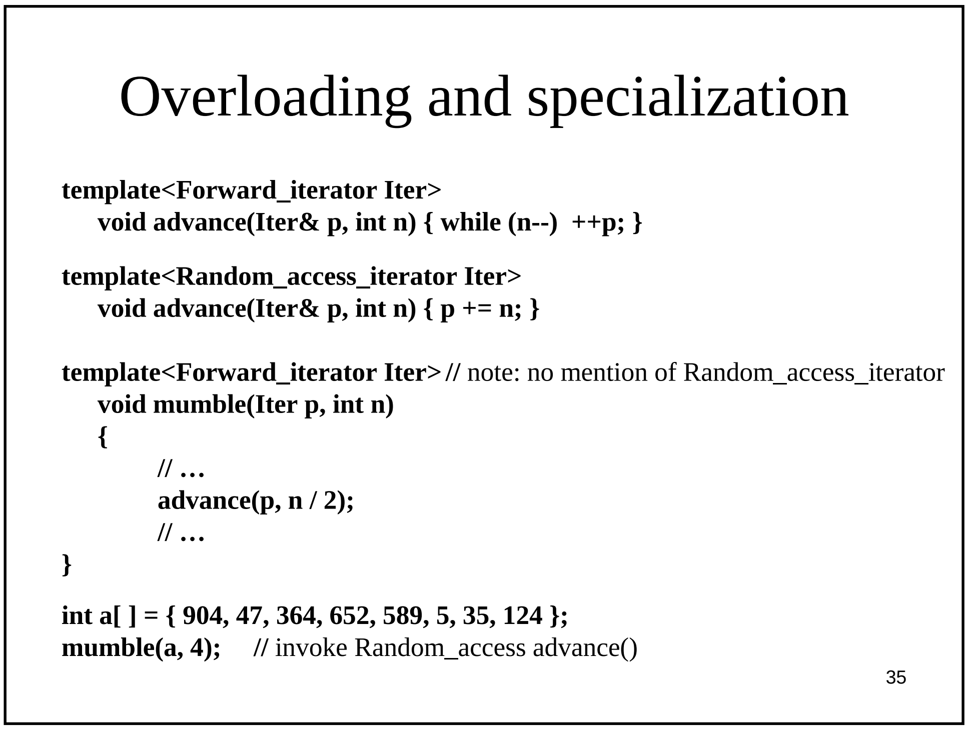
```
concept Small<class T, int N> { sizeof(T) <= N; }

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

- Now:
  - The writer of **foo()** cannot write an assert for **Small<X,max>**
    - he doesn't know **X**
  - How does a user of **foo()** do a static assert for **Small<X>** for **foo(x)**?
    - he should not know about **f()** or **max**
  - If the user writes an assert he must choose between **Small<X,max>** or **!Small<X,max>**

34



Overloading and specialization

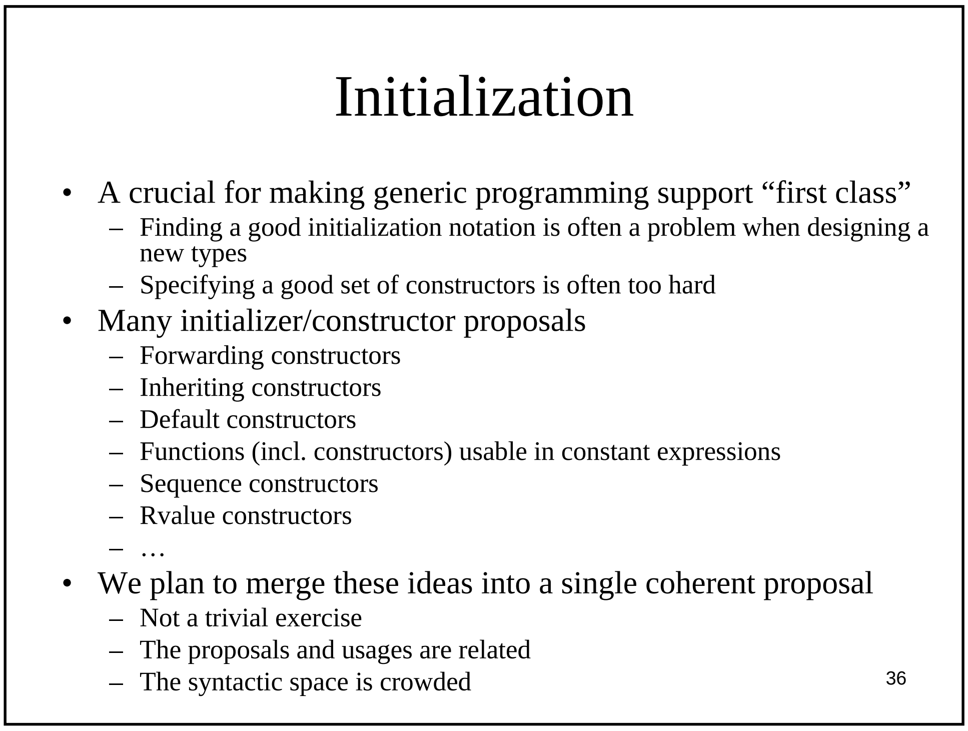
```
template<Forward_iterator Iter>
   void advance(Iter& p, int n) { while (n--)  ++p; }

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

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

int a[ ] = { 904, 47, 364, 652, 589, 5, 35, 124 };
mumble(a, 4);     // invoke Random_access advance()
```

35



Initialization

- A crucial for making generic programming support “first class”
  - Finding a good initialization notation is often a problem when designing a new types
  - Specifying a good set of constructors is often too hard
- Many initializer/constructor proposals
  - Forwarding constructors
  - Inheriting constructors
  - Default constructors
  - Functions (incl. constructors) usable in constant expressions
  - Sequence constructors
  - Rvalue constructors
  - …
- We plan to merge these ideas into a single coherent proposal
  - Not a trivial exercise
  - The proposals and usages are related
  - The syntactic space is crowded

36

Sequence constructors

- The problem: we support arrays better than user-defined containers
  - Leads to verbose, unsafe code
  - Violates one of C++’s design principles
    - Support user-defined types at least as well as built-in types

```
double a[ ] = { 3.4, 4.5, 5.6 };
vector<double> v(a, a+sizeof(a)/sizeof(double));
```

37

Sequence constructors

- A solution: let the user define a constructor taking an initializer list

```
template<Value_type T>
Class vector {
    // …
    vector(Sequence<T>);            // possible syntax for sequence constructor
                                    // a Sequence<T> specifies the start and end
                                    // of the sequence defined by an initializer list
    // …
};

vector<double> v = { 3.4, 4.5, 5.6 };
```

38

Move semantics

- Allow objects to be “moved” rather than copied when the old value is no longer used

```
class X {
   // …
   X(const X&);        // all “normal copy”
   X(X&&);             // “copy” X that is an rvalue (i.e. unique)
                       // can use destructive copy “move”
   // …
};

X x = f();      // just move the result, f can’t use it anymore
X y = x;        // ordinary copy
```

39

Literal constructors

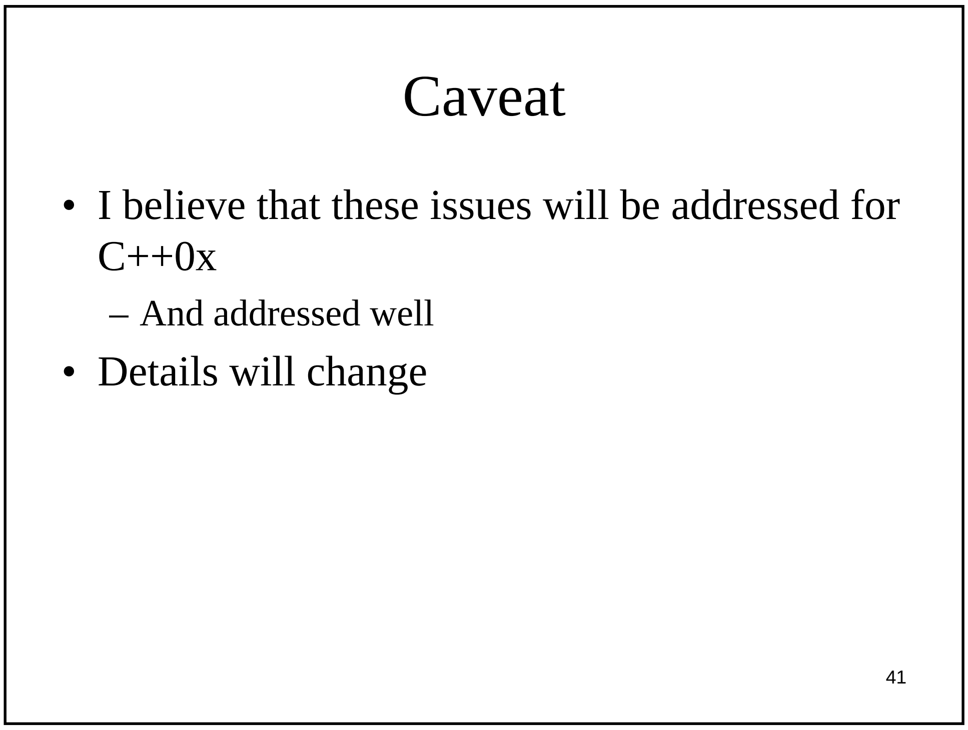
- How can we ensure that a constructor is run at compile time?

```
template<class Scalar> class complex {
   Scalar re, im;
   literal complex(Scalar r, Scalar i) : re(r), in(i) { }
   // …
};

complex<int> x(1.2, 5.6);       // pick literal constructor
complex<int> xx(i,j);           // pick ordinary constructor
```

- Closely related to constant expressions and ROMable

40



Caveat

- I believe that these issues will be addressed for C++0x
  - And addressed well
- Details will change

41