C++0x initialization: lists

Bjarne Stroustrup

Texas A&M University

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

C++0x initialization: lists

- A case study
  - Details matter
  - Details are hard
  - Compatibility requirements are really tricky
- We have not forgotten the big picture
  - But that’s another talk

Overview

- What we want
- Obstacles
- Initializer lists
- Generalization to all initialization
- A way of eliminating narrowing conversions
- Summary

Initialization – what do we want?

- Initializer lists for containers
  - as for arrays (and structs)
- Uniform initialization syntax and semantics
  - One syntax and one semantics for all uses of that syntax
  - In every context
    - Global / namespace
    - Free-store
    - Local
    - Member and base
    - Const and non-const
  - No implicit conversion surprises
  - No element list vs. constructor argument ambiguity surprises
  - Compatibility: Don’t break my code!
  - No verbosity (compare to what we have)

It’s a tricky puzzle

- C provided
  - **X a = { v };** initialization for structs, arrays, and non-aggregates
  - **X a = v;** initialization for non-aggregates
- C++ added
  - **new X(v);**
  - **X a(v);** for classes with constructors and non-aggregates
  - **X(v)** temporaries and “function style” casts
  - Explicit and “ordinary” constructors
  - Private copy constructors
- Parenthesized lists are heterogeneous but can look homogeneous
  - **pair<string,int>(“Hello”,10);**
  - **vector<int>(10,2);** // 10 elements each with the value 2
- Curly-brace lists can be homogeneous or heterogeneous
  - **struct S { int x, char* p; } s = { 10, 0 };**
  - **int a[] = { 10, 0 };**
- Some of these syntactic differences reflects semantic differences
  - most do not

Really basic examples

- Intialization of variables:

```
vector<int> seq = { 1, 2, 3, 5, 8, 13 };
vector<string> loc = {
    “Lillehammer”,
    “Kona”,
    “Oxford”,
    “Portland”
};
```

- Initialization in argument passing:

```
template<class T> sum(const vector<T>&);
int x = sum(seq);
int y = sum( { 1, 2, 3, 5, 8, 13 } );
```

Why?

- Fix violation of one of C++'s basic design rules
  - "provide as good support for user-defined types as for built-in types"
    - Note: **int a[ ] = { 1, 2, 3, 5, 8 };**
- Uniformity is essential for generic programming
  - We should know how to initialize a type **X** with a value **v** for every **X** and for every **v**
    - Without studying the details of every **X** and **v**

An example

- Four different syntaxes:
  - **X t1 = v;** // “copy initialization” possibly copy construction
  - **X t2(v);** // direct initialization
  - **X t3 = { v };** // initialize using initializer list
  - **X t4 = X(v);** // make an X from v and copy it to t4
- All have their uses and their fans
  - It’s a mess
  - We can define **X** so that for some **v**,
    - 0, 1, 2, 3, or 4 of these definitions compiles
    - the values of some of the 4 variables differ
- Sometimes, we only have one syntax alternative
  - **new X(v);** // free-store allocation
  - **X(v);** // temporary of type X

An example: X is a scalar

- **double v = 7.2;**
- **typedef int X;**

- **X t1 = v;** // ok (yuck! Narrowing conversion)
- **X t2(v);** // ok (yuck! Narrowing conversion)
- **X t3 = { v };** // ok; see standard 8.5; equivalent to “double t3 = v;”
- **X t4 = X(v);** // ok (explicit conversion)

An example: X is a container

- **int v = 7;**
- **typedef vector<int> X;**

- **X t1 = v;** // error: vector's constructor for int is explicit
- **X t2(v);** // ok
- **X t3 = { v };** // error: vector<int> is not an aggregate
- **X t4 = X(v);** // ok (make an X from v and copy it to t4)
  // (possibly/probably optimized)

An example: X is a C-style struct

- **int v = 7;**
- **typedef struct { int x; int y; } X;**

- **X t1 = v;** // error
- **X t2(v);** // error
- **X t3 = { v };** // ok: X is an aggregate
  // (“extra members” are default initialized)
- **X t4 = X(v);** // error: we can’t cast an int to a struct

An example: X is a pointer

- **int v = 7;**
- **typedef int* X;**

- **X t1 = v;** // error
- **X t2(v);** // error
- **X t3 = { v };** // error
- **X t4 = X(v);** // ok: explicitly convert an int to an int*; yuck!

Is this a real problem?

- Yes!
  - A major source of confusion and bugs
- Can it be solved by restriction?
  - No
    - No existing syntax can be used in all cases

      int a [] = { 1,2,3 }

      new int(4);
    - No existing syntax has the same semantics in all cases

      typedef char* Pchar;

      Pchar p(7); // error (good!)

      Pchar(7); // fine (ouch!)

Aggregate initializer lists

- A nice C and C++ feature, but
  - it can be used only in as an initializer of array and struct variables
  - It can be used only in a few contexts

- **X v = {1, 2, 3.14};** // as initializer (ok)

- **void f1(X);**
  f1({1, 2, 3.14}); // as argument (error)

C++0x initializer lists

- Initializer lists can be used for all initialization

- **X v = {1, 2, 3.14};** // as initializer (ok)
- **void f1(X);**
  f1({1, 2, 3.14}); // as argument (error)
- **X g() { return {1, 2, 3.14}; }** // as return value (error)
- **X* p = new X{1, 2, 3.14};** // make an X on free store X (error)
- **class D : public X {**
  X m;
  D()
  : X{1, 2, 3.14}, // base initializer (error)
  m{1, 2, 3.14} // member initializer (error)
  { }
  };

Idea

- Allow the designer of a class to define a constructor to deal with initializer lists
  - A “sequence constructor”
- Allow initializer lists for every initialization

- See all the gory details
  - Bjarne Stroustrup and Gabriel Dos Reis: *Initializer lists (Rev. 3)*. WG21 N2215=07-0075
  - Gabriel Dos Reis and Bjarne Stroustrup: *Initializer Lists for Standard Containers*. WG21 N2220=07-0080

Basic rule for initializer lists

- If a constructor is declared
  - If there is a sequence constructor that can be called for the initializer list
    - If there is a unique best sequence constructor, use it
    - Otherwise, it's an error
  - Otherwise, if there is a constructor (excluding sequence constructors)
    - If there is a unique best constructor, use it
    - Otherwise, it's an error
  - Otherwise, it's an error
- Otherwise
  - If we can do traditional aggregate or built-in type initialization, do it
  - Otherwise, it's an error

What should a sequence constructor look like?

- This turned into a very contentious issue (syntax always does):
  - **template<Forward_iterator For> C<E>::C(For first, For last);**
  - **template<int N> C<E>::C(E(&)[N]);**
  - **C<E>::C(const E*, const E*);**
  - **C<E>::C{}(const E* first, const E* last);**
  - **C<E>::C(E … seq);**
  - **C<E>::C(... E seq);**
  - **C<E>::C(... initializer_list<T> seq);**
  - **C<E>::C(... E* seq);**
  - **C<E>::C ({}<E> seq);**
  - **C<E>::C(E{} seq);**
  - **C<E>::C(E seq{});**
  - **C<E>::C(E[*] seq);** // use sizeof to get number of elements
  - **C<E>::C(E seq[*]);**
  - **C<E>::C(const E (&)[N]);** // N becomes the number of elements
  - **C<E>::C(initializer_list<T> seq);**
  - **C<E>::C(E [N]);**
  - **C<E>::C({E});**
  - …

What should a sequence constructor look like?

- And the answer is:

```
template<class E> class vector {
    E* elem;
public:
    vector (std::initializer_list<E> s) // sequence constructor
    {
      reserve(s.size());
      uninitialized_fill(s.begin(), s.end(), elem);
    }
    // … as before …
};

std::vector<double> v = {1, 2, 3.14};
```

Semantics

- Compiler lays down array and sequence constructor copies
  - For example

```
std::vector<double> v = { 1, 2, 3.14 };
```

  - Implemented as

```
double temp[] = { double(1), double(2), 3.14 } ;
initializer_list<double> tmp(temp,sizeof(temp)/sizeof(double));
vector<double> v(tmp);
```

Initializer_list<T> definition

```
template<class E> class initializer_list {
    // representation (probably a pair of pointers or a pointer plus a length)
    // constructed by compiler
    // implementation defined constructor
public:
    // allow uses: [first,last) and [first, first+length)

    // default copy constructor and copy assignment
    // no destructor (or the default destructor, which would mean the same)

    constexpr int size() const; // number of elements

    const E* begin() const; // first element
    const E* end() const; // one-past-the-last element
};
```

So, what about uniformity?

- Can we generalize initializer syntax and semantics to cover all cases?
  - Yes!
    - But
    - But
    - But now we have to deal with the really messy details
      - See B. Stroustrup and G. Dos Reis: “Initializer lists” (Rev 3.) N2215=07-0075
      - Ambiguities
      - Syntax
      - Narrowing conversions
      - C99
      - Header files
      - Template deduction
      - …

Syntax

- Every form of initialization can accept the { … } syntax
- The = can be optionally added where it is currently allowed

```
X x1 = X{1,2};
X x2 = {1,2};       // the = is optional and not significant
X x3{1,2};
X* p2 = new X{1,2};

struct D : X {
   D(int x, int y) :X{x,y} { /* … */ };
};

struct S {
   int a[3];
   S(int x, int y, int z) :a{x,y,z} { /* … */ }; // solution to old problem
};
```

Aesthetics

- Do you like this notation?

```
X x1 = { 1, 2 };
X x2{1,2};
f(X{1,2});
X* p2 = new X{1,2};
```

- Why? / Why not?

- People’s reactions vary dramatically
- People’s rationales vary dramatically
- Give it a chance
- Think about it

Arrays and structs

- Initializer lists do double duty:

```
struct S { int x, y; };
S s = { 1,2 };          // or structs

int a[] = { 1,2 };      // for arrays
```

- We can't change that
  - People like it
  - C compatibility
  - C++ compatibility
- This comes back to haunt us

```
vector<int> v1(1,2);    // one element with value 2
                        // "ordinary constructor"
vector<int> v2{1,2};    // two elements with values 1 and 2
                        // assuming a sequence constructor
```

{ … } for ordinary constructors

- To achieve uniform notation, we must allow { … } initialization for “ordinary constructors”:
  - It is allowed for structs and arrays
  - It is allowed for scalars
- Current irregularity
  - **double d = { 2.3 };** // ok
  - **complex<double> z = { 2.3 };** // error in C++98
  - **struct Dpair { double re, im; };**
    Dpair dp = { 2.3 }; // ok

{ … } for ordinary constructors

- An ordinary constructor can be invoked with the {…} syntax (as long as there is no sequence constructor):

```
complex<double> z1(1,2);      // ok as always
complex<double> z2{1,2};      // ok
```

- The uniformity happens to solve an old problem:

```
complex<double> z3;           // default initialization (0,0)
complex<double> z4();         // oops! A function
complex<double> z3{};         // default initialization (0,0)
```

Disambiguation

- To get “the old semantics” we use “the old syntax”

```
vector<int> v1(1,2);      // one element with value 2
vector<int> v2{1,2};      // two elements with values 1 and 2
                          // assuming a sequence constructor
```

- The initializer list notation gives precedence to the sequence constructor if one exists
  - This is not ideal
    - Because it breaks the use of uniform syntax
  - This is “almost necessary”
    - We can’t ban the old syntax anyway
    - Examples follow
  - This is relatively rare
    - You need a constructor of a container with elements of a type that are also used as arguments to other constructors to get this problem

Disambiguation

- Sequence constructors take precedence

```
vector<int> v0 { };          // no elements
vector<int> v1 { 3 };        // one element
vector<int> v2 { 1, 2 };     // two elements
vector<int> v3 { 1, 2 , 3 }; // three elements

vector<int*> vp0 { };              // no elements
vector<int*> vp1 { &i1 };          // one element
vector<int*> vp2 { &i1, &i2 };     // two elements
vector<int*> vp3 { &i1 , &i2, &i3 }; // three elements
```

Disambiguation

- Why not simple overload resolution?
  - Would give far too many “false alarms”
  - This would be awful

```
vector<int> v0 { };     // ambiguous:
                        //default constructor or empty initializer?
vector<int> v1 { 3 }; // ambiguous:
                        // three elements (with default values) or one element?
vector<int> v2 { 1, 2 };          // ambiguous (with count+value iterator)
vector<int> v3 { 1, 2 , 3 };      // ok (three elements with values 1, 2, 3)

vector<int*> vp1 { };             // ambiguous
vector<int*> vp1 { &i1 };         // ok (one element)
vector<int*> vp1 { &i1, &i2 };            // ambiguous (with iterator initializer)
vector<int*> vp1 { &i1 , &i2, &i3 };    // ok (three elements)
```

Disambiguation

- What if we want an initializer list of a specified type?
  - Use **X{ ... }**
  - For example:

```
void f(X);
void f(Y);
f( {1,2} );          // could be ambiguous
f(X{1,2});           // call f(X)
f(Y{1,2});           // call f(Y)
```

Disambiguation

- We tried a lot of crazy and not-so-crazy alternatives
  - See paper

Semantics

- { …} initialization is direct initialization
- For example

```
vector<string> vs = { “CPL”, “BCPL”,“C”, “C++” };

vector<string> verbose = {
    string(“CPL”),
    string(“BCPL”),
    string(“C”),
    string(“C++”)
};
```

Semantics

- { … } initialization doesn't narrow

  vector<int> vi = { 1, 2.3, 4, 5.6 }; // error: double to int narrowing

- This is still allowed (and compatible)

  char a[] = { 'a', 'b', 'c', 0 }; // error: 0 is an int

  We allow this case because we can prove that it's not really narrowing!

- Potentially the most controversial issue
  - After surveying a lot of code we find that the problem mostly affect literals (and constant expressions) where compilers already detect narrowing and can verify that the conversion actually doesn't narrow.

Why mess with narrowing?

- Casting!
- Function-style cast looks innocent, but isn't:

```
typedef char* Pchar;
int i;
// …
Pchar p = Pchar(i); // no obviously nasty reinterpret_cast
```

- There is no general syntax for construction in generic code:

```
template<class T, class V> void f(T t, V v)
{
    T x;
    // …
    x = T(v); // construct (but for some types it casts)
    // …
}
```

Why mess with narrowing?

- Solution:

```
template<class T, class V> void f(T t, V v)
{
    T x;
    // …
    x = T{v};  // constructs; no nasty casting (or narrowing)
}
```

- Consider the uniformity requirement:
  - **T{v}**
  - **T x{v};**
  - **T y = {v};**
  - **T a[] = {v};**
  - **p = new T{v}**
- The values of **T{v}**, **x**, **y**, **a[0]**, and ***p** must be identical.

It is worth while

- **vector<int> v = { 1, 2, 3, 4 };**
- **map<string,int> m = { {“ardwark”,91}, {“bison”, 43} };**
- **f(vector<int>&);** … **f({ 1.2, 4.5, 8.9});**
- **double d = 2.3;** … **int x = { d };** // error: narrowing

Will it happen?

- Initializer lists
- Ban on narrowing conversions in {…} initialization

- I hope so
- I think so
  - based on “evolution working group” votes and feedback
- Nothing is certain until the votes are in
  - I hope for next week!