Doc No: SC22/WG21/N1890
J16/05-0150
Date: September 22, 2005
Project: JTC1.22.32
Reply to: Bjarne Stroustrup
Computer Science Dept.
Texas A&M University, TAMU 3112
College Station TX USA 77843-3112
Fax: +1-979-458-0718
Email: bs@cs.tamu.edu

Initialization and initializers

Bjarne Stroustrup and Gabriel Dos Reis
Texas A&M University

Abstract

This paper presents a way through the maze of proposals related to initialization, constructors, and related issues. The aim is to provide a synthesis of many proposals that can guide further work. It is not the aim to present every detail for final approval.

The discussion is based on the earlier papers and on discussions in the evolutions group.

1 What's the problem?

There is not one problem; there are several interrelated problems. Each individual problem can be solved relatively easily. The real problem is to provide a coherent solution to all (or most) of the problems. Here is a list of problems and suggested improvements we consider potentially related so that they must be considered together:

- General use of initializer lists (Dos Reis & Stroustrup N1509, Gutson N1493, Meredith N1806, Meridith N1824, Glassborow N1701)
- There are four different syntaxes for initializations (Glassborow N1584, Glassborow N1701)
- C99 aggregate initialization (C99 standard)
- Literal constructors (Stroustrup N1511)

- Forwarding constructors (Glassborow ???)
- More general constant expressions (Dos Reis 1521)
- Inherited constructors (Glassborow ???)
- Type safe variable length argument lists (C++/CLI)
- Default constructors (and other operations) (Glassborow N1582)
- Overloading "new style" casts
- Making **T(v)** construction rather than conversion (casting)
- Rvalue constructors (Hinnant ???)

In each case, the person and paper referred to is just one example of a discussion, suggestion, or proposal. In many cases there are already several suggested solutions. This is not even a complete list: initialization is one of the most fertile sources of ideas for minor improvements to C++. Quite likely, the potential impact on the programmer of sum of those suggestions is not minor. In addition to the listed sources, we are influenced by years of suggestions in email, newsgroups, etc.

We go into some detail to illustrate the solutions we propose. However, please remember that the aim of this paper is not to present a complete and detailed solution of a specific problem or even to exhaustively explain the specific problems. Our primary aim is to present an overview that allows us to work out the details of a set of related problems without fear that the solution to one problem precludes the solution of another.

The paper discusses the issues roughly in the order presented above.

2 Four ways of providing an initializer

Initialization of objects is an important aspect of C++ programming. Consequently, a variety of facilities for initialization are offered and the rules for initialization has become complex. Can we simplify them? Consider

```
X t1 = v;       // "assignment initialization" possibly copy construction
X t2(v);        // direct initialization
X t3 = { v };   // initialize using initializer list
X t4 = X(v);    // make an X out of v and copy it to t4
```

We can define **X** so that for some **v**, 0, 1, 2, 3, or 4 of these definitions compile. For example:

```
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
```

and

```
int v = 7;
typedef int X;
X t1 = v;       // ok
X t2(v);        // ok
X t3 = { v };   // ok; see standard 8.5; equivalent to "int t3 = v;"
X t4 = X(v);    // ok
```

and

```
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
X t4 = X(v);    // error: we can't cast an int to a struct
```

and

```
int v = 7;
typedef  int* X;
X t1 = v;       // error
X t2(v);        // error
X t3 = { v };   // error
X t4 = X(v);    // ok: unfortunately this converts an int to an int* (see §8)
```

1.1 Can we eliminate the different forms of initialization?

It would be nice if we didn't need four different ways of writing an initialization. Francis Glassborow explains this in greater detail in N1701. Unfortunately, we loose something if we eliminate the distinctions. Consider:

```
vector<int> v = 7;     // error: the constructor is explicit
vector<int> v(7);      // ok
```

If the two versions were given the same meaning, either both would be correct (and we would be back in "the bad old days" where all constructors were used as implicit conversions) or both would fail (and every program using a vector or similar type would fail). We consider both alternatives unacceptable.

The equivalent problem for argument passing demonstrates that we cannot simplify by eliminating copy initializations or explicit constructors:

```
void f(vector<int> v);
f(7);    // error: the constructor is explicit
```

```
vector<int> v = { 1,2,3,4,5,6,7 };
f(v);    // copy
```

Also:

```
class X { /* … */ X(int); private: X(const X&); }; // no copy allowed
X x0 = X(1);   // error (copy)
X x1 = 1;      // error (copy)
X x2(1);       // ok (no copy)
```

To have a single rule here would require us to choose between breaking a lot of code (disallow all three cases) and requiring that copy not be considered (allow all three cases). We suspect we could live with the latter choice, but it would be a change making the language more permissive.

Consider finally the most explicit form of initialization:

```
vector<int> v = vector<int>(7);      // copy?
X e3 = X(1);                         // copy?
```

We cannot recommend that style for systematic use because it implies serious inefficiency unless compilers are guaranteed to eliminate the copy. It would also break reasonable expectations unless the access to the copy constructor was checked (to make the initialization of **e3** fail). If we special-cased this form of initialization (to make the examples legal and efficient), we would end up with a semantics that differed from that of argument and return value initialization. For example:

```
template<class T> void f(X v);
f(vector<int>(7));     // copy? Yes, we must copy
f(X(1));               // copy? Yes, we must copy and copy of X is disallowed
```

We conclude that we must live with different meanings for different initialization syntaxes. That implies that we can try to make the syntax and semantics more general and regular, but we cannot reach the ideal of a single simple rule.

3 Initializer lists

There seems to be a very widespread wish for wider use of initializer lists as a form of user-defined-type literal. The pressure for that comes not only from “native C++” wish for improvement but also from familiarity with similar facilities in languages such as C99, Java, and C#. Our aim is to allow initializer lists for every initialization. What you loose by consistently using initializer lists are the possibilities of ambiguities inherent in = assignment.

Consider a few plausible examples:

```
X v = {1, 2, 3.14};                    // as initializer
const X& r1 = {1, 2, 3.14};            // as initializer
X& r2 = {1, 2, 3.14};                  // as lvalue initializer

void f1(X); f1({1, 2, 3.14});          // as argument
void f2(const X&); f2({1, 2, 3.14});   // as argument
void f3(X&); f3({1, 2, 3.14});         // as lvalue argument

X g() { return {1, 2, 3.14}; }         // as return value

class D : public X {
        X m;
        D() : X({1, 2, 3.14}),         // base initializer
              m({1, 2, 3.14}) { }      // member initializer
};
X* p = new X({1, 2, 3.14});  // make an X on free store X
                             // initialize it with {1,2,3.14}

void g(X);
void g(Y);
g({1, 2, 3.14});          // (how) do we resolve overloading?
```

We must consider the cases where **X** is a scalar type, a class, a class without a constructor, a union, and an array. As a first idea, let's assume that all of the cases should be valid and see what that would imply and what would be needed to make it so.

Note that this provides a way of initializing arrays. We don't consider that particularly important, but there are occasional requests for a way of doing that.

1.2 The basic rule for initializer lists

The most general rule of the use of initializer lists is:

- Look for a sequence constructor and use it if we find a best one; if not
- Look for a constructor and use it if we find a best one; if not
- Look to see if we can do traditional aggregate or built-in type initialization; if not
- It's an error
-

We propose a slightly more restrictive rule "never use aggregate initialization if a constructor is declared". Without a restriction, we would not be able to enforce invariants by defining constructors. Consequently, we consider a restriction necessary and get this modified basic rule:

- If a constructor is declared
    - Look for a sequence construct and use it if we find a best one; if not
    - Look for a constructor and use it if we find a best one; if not

  - It's an error
- If no constructor is declared
  - look to see if we can do traditional aggregate or built-in type initialization; if not
  - It's an error
  -

This can (and should) be integrated into the overload resolution rules.

1.3 Sequence constructors

A sequence constructor is defined like this:

```
class C {
        C{}(const int* first, const int* last); // construct from a sequence of ints
        // …
};
```

The **{}** is syntax indicating that the constructor is a sequence constructor. A sequence constructor is called for an array of values indicated by a pointer to the first element and a pointer to the one-beyond-the last element. So a C can be initialized by an initializer lists that can be seen as an array of **int**s. Note that a sequence constructor is syntactically distinct from other constructors. Note also that the arguments to a sequence constructor are a pair of pointers to **const**. A sequence constructor cannot modify its input sequence.

So let's consider the examples above when **X** is **std::vector<double>**. That's easily done: **vector** has no sequence constructor, so we try **{1, 2, 3.14}** as a set of arguments to other constructors, that is, we try **vector(1,2,3.14)**. That fails, so all of the examples fail to compile when **X** is **std::vector**.

Let's try adding a sequence constructor to **vector**:

```
template<class E> class vector {
public:
        vector{}(const E* first, const E* last); // construct from a sequence of Es
        // … as before …
};
```

Now, some (but not all) of the examples work when **X** is **vector<double>**. In each case, **{1, 2, 3.14}** is interpreted as a temporary constructed like this:

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

That is, the compiler constructs an array containing the initializers converted to the desired type, initialized a temporary **vector** using the sequence constructor, and uses the resulting **vector** as the initializer instead of the initializer list. This implies that every use

of **{1, 2, 3.14}** in a place that requires an rvalue succeeds and the places that require an lvalue fails.

> **Discussion:** In the EWG there were strong support for the idea of the sequence constructor, but also serious disagreement about the syntax needed to express it. There was a strong preference for syntax to make the "special" nature of a sequence constructor explicit. This could be done by a special syntax (as suggested here) or a special (compiler recognized) argument type. For example:
>
> ```
> class X {
>     // …
>     X(Sequence<int>);    // construct from a initializer list of ints
>     // …
> };
> ```
>
> We prefer the **X{}(…)** design, because "syntax" seems to be the majority view and even more because it fits with the design of literal constructors (see §5).
>
> There is a choice between representing a sequence as the STL-style **(first,last)** or a **(first,length)** view. We don't think there is a fundamental reason to prefer the one over the other; after all, given one we can express the other: **(first,last-first)** and **(first,first+length)**. We chose the former to emphasize the standard library and because many classes will have an ordinary constructor for STL sequences in addition to the sequence constructor. There might be an argument for the **(first,length)** view because we can specialize a template on an integer constant, but we see no realistic use for that.

1.4 Initializer lists and ordinary constructors

When a class has both a sequence constructor and an "ordinary" constructor, a question can arise about which to choose. The resolution outlined in §3.2 is that the sequence constructor is chosen if the initializer list can be considered as an array of elements of the type required by the sequence constructor (possibly after conversions of elements). If not, we try the elements of the list as arguments to the "ordinary" constructors. The former ("use the sequence constructor") matches the traditional use of initializer lists for arrays. The latter ("use an ordinary constructor") mirrors the traditional use of initializer lists for **struct**s (initializing constructor arguments rather than **struct** members). Consider a few examples:

```
vector<double> v1({1,2});   // v1 has two elements (values: double(1),double(2))
                            // use sequence constructor
vector<double> v2({1});     // v2 has one element (value: double(1))
                            // use sequence constructor
```

Since we can convert 1 and 2 to the **double**s required by **vector<double>**'s sequence constructor, the sequence constructor is used for v1 and v2. If we don't want that, we must use another form of initialization:

```
vector<double> v11(1,2);    // v11 has one element (value: double(1))
                            // use ordinary constructor
vector<double> v22(1);      // v22 has one element (value: double(), i.e. 0.0)
                            // use ordinary constructor
```

If the type of the elements in the initializer list doesn't match what the sequence constructor requires, we use an ordinary constructor:

```
vector<double> v3({1,2,My_alloc}); // use ordinary constructor
                                   // can't convert My_alloc to double
vector<double> v4({v2.begin(),v2.end()}); // copy v2 into v4
                    // can't convert vector<double>::iterator to double
```

Discussion: Should the initialization of **v3** and **v4** simply be errors? That is, should we reject the use of initializer list when there is no sequence constructor? For aggregates, initializer lists serve two purposes:

- Initialize homogeneous sequences (i.e. arrays)
- Initialize heterogeneous sequences (i.e. structs)

To provide a general initializer mechanism, we must preserve this dual use. To support user-defined types as well as built-in types and to provide a uniform syntax for initialization, we must somehow ensure that initializer lists can be used for both sequences (to initialize containers) and "ordinary objects". We can have that support disjoint: "if you have a sequence constructor, you can't use initializer lists for other constructors, but if you don't have a sequence constructor you can" or we can have it with "sequence constructor has priority" as suggested above. If we choose the "either/or" rule, we will not be able to use **{}** initialization uniformly; another initialization syntax will have to be used for classes with sequence constructors; importantly, we would not be able to use **{ }** initialization for standard library containers. This would seriously weaken any effort to teach people to uniformly use a single initialization syntax (the **{}** notation). Furthermore, we would not be able to add a sequence constructor to a class that is already in use because that would break any **{}** initialization already used.

1.5 Initializer lists, aggregates, and built-in types

So what happens if a type has no constructors? We have three cases to consider: an array, a class without constructors, and non-composite built-in type (such as an **int**). First consider a type without constructors:

```
struct S { int a; double v; };
S s = { 1, 2.7 };
```

This has of course always worked and it still does. It meaning is unchanged: initialize the members of **s** in declaration order with the elements from the initializer list in order, etc.

Arrays can also be initialized as ever. For example:

```
int d[] = { 1, 2, 3, 5, 8 };
```

What happens if we use an initializer list for a non-aggregate? Consider:

```
int a = { 2 };          // ok: a==2
                        // (as currently: there is a single value in the initializer list)
int b = { 2, 3 };       // error: two values in the initializer list
int c = {};             // ok: default initialization: c==int()
```

In line with our ideal of allowing initializer lists just about everywhere – and following existing rules – we can initialize a non-aggregate with an initializer list with 0 or 1 element. The empty initializer list gives value initialization. The reason to extend the use of initializer lists in this direction is to get a uniform mechanism for initialization. In particular, we don't have to worry about whether a type is implemented as a built-in or a user-defined type and we don't have to depart from the direct initialization to avoid the unfortunate syntax clash between **()** initialization and function declaration. For example:

```
X a = { v };
X b = { };
```

This works for every type **X** that can be initialized by a **v** and has a default constructor. The alternatives have well known problems:

```
X a = v;        // not direct initialization (e.g. consider a private copy constructor)
X b;            // different syntax needed (with context sensitive semantics!)
X c = X();      // different syntax, repeating the type name

X a2(v);        // use direct initialization
X b2();         // oops!
```

It appears that **{}** initialization is not just more general than the previous forms, but also less error prone.

We do not propose that surplus initializers be allowed:

```
int a = { 1, 2 };       // error no second element
struct S { int a: };
S s = { 1,2 };          // error no second element
```

Allowing such constructs would simply open the way for unnecessary errors.

Discussion: Discussion: The standard currently says (12.6.1/2) that when an object is initialized with a brace-enclosed initializer list, elements are initialized through "copy-initialization" semantics. For uniformity and consistency of the initialization rules this should be changed to "direct-initialization" semantics. We think that will not change the semantics of current well-formed programs.

4 Initializer list technicalities

As the saying goes "the devil is in the details", so let's consider a few technical details to try to make sure that we are not blindsided.

1.6 Sequence constructors

Can a class have more than one sequence constructor? Yes. A initializer list that would be a valid for two (or more) sequence constructors is ambiguous.
Can a sequence constructor be a template? Yes. Note that a "yes" here implies that more than one sequence constructor is possible.

Can a sequence constructor be invoked for a sequence that isn't an initializer list? No.

1.7 What really is an initializer list?

The simplest model is an array of values placed in memory by the compiler. That would make an initializer list a modifiable lvalue. It would also require that every initializer list be placed in memory and that if an initializer list appears 10 times than 10 copies must be present. We therefore propose that all initializer lists be rvalues. That will enable two optimizations:

- Identical initializer lists need at most be store once (though of course that optimization isn't required).
- An initializer list need not be stored at all. For example, **z=complex{1,2}** may simply generate two assignments to **z**.

The second optimization would require a clever compiler or literal constructors (§5).

Note that an initializer list that is to be read by a sequence constructor must be placed in an array. The element type is determined by the sequence constructor. Sometimes, it will be necessary to apply constructors to construct that array.

Initializer lists that are used for aggregates and argument lists can be heterogeneous and need rarely be stored in memory.

Must initializer lists contain only constants? No, variables are allowed (as in currect initializer lists); we just use a lot of literals because that's the easiest in small examples.

Can we nest initializer lists? Yes (as in current initializer lists). For example:

```
vector<vector<int>> v = { {1,2,3}, {4,5,6}, {7,8,9} };        // a 3 by 3 matrix
```

1.8 Ambiguities and deduction

An initializer list is simply a sequence of values. If it is considered to have a type, it is the list of its element types. For example, the type of {**1,2.0}** would be **{int,double}**. This implies that we can easily create examples that are – or at least appears to be – ambiguous. Consider:

```
class X  {
        X{}(const int*, const int*);   // sequence constructor
        // …
};

class Y {
        Y{}(const int*, const int*);   // sequence constructor
        // …
};

class Z {
        Z(int,int);       // not a sequence constructor
        // …
};

void f(X);
void f(Y);

void g(Y);
void g(Z);

f({1,2,3});       // error: ambiguous (f(X) and f(Y)?)
g({1,2,3});       // ok: g(Y)
g({1,2});         // ok: g(Y)  (note: not g(Z));
g({1});           // ok
```

The overload resolution rules are basically unchanged: try to mach all functions in scope and pick the best mach if there is a best match. What is new is a need to specify conversions used for a legal call using an initializer list so that it can be compared with other successful matches.

> **Discussion:** We resolve the **g({1,2})** call by preferring the sequence constructor in one class over an ordinary constructor in another class. The alternative would be to have the resolution depend on the number of elements in the initializer list.

How do we resolve ambiguity errors? By saying what we mean; in other words by stating our intended type of the initializer list:

```
f(X{1,2,3});    // ok: f(X)
g(Z{1,2});      // ok: g(Z)
```

If we want to increase C99 compatibility we could also accept the more verbose version:

```
f((X){1,2,3});  // ok: f(X)
g((Z){1,2});    // ok: g(Z)
```

This is not something we recommend, though, and there is a danger that it might become popular in C++ just as "the abomination" **f(void)**. Also, there would be subtle incompatibilities between the C99 definition of such as construct and any consistent C++ view (see N1509).

Discussion: We do not propose to allow an "unqualified initializer list" to be used as an initializer for a variable declared auto or a template argument. For example:

```
auto x = {1, 2, 3.14};              // error
template<class T> void ff(T);
ff({1, 2, 3.14});                   // error
```

There is no strong reason not to allow this, but we don't want to propose a feature until we have a practical use in mind. If we wanted to allow this, we could simply "remember" the type of the initializer list and use it when the **auto** variable or template argument is used. In this case, the type of **x** would be **{int,int,double}** which can be converted into a named type when necessary. For example:

```
auto x = {1, 2, 3.14};          // remember x' is a {int,int,double}
vector<int> v = x;              // initialize v {1, 2, 3.14};
g(x);                           // as above
```

It's comforting to know that the concepts extend nicely even if we have no use for the extension.

1.9 Syntax

So far, we have used initializer lists after = in definitions (as always) and as function arguments. The aim is to allow an initializer list wherever an expression is allowed. In addition, we might consider leaving out the = in a declaration:

```
auto x1 = X{1,2};
X x2 = {1,2};
X x3{1,2};
```

```
X x4({1,2});
X x5(1,2);
```

These five declarations are equivalent (except for the name of the variables) and all variables get the same type (**X**) and value (**{1,2}**). Similarly, we can leave out the parentheses in an initializer after **new**:

```
X* p1 = new X({1,2});
X* p2 = new X{1,2};
```

It is never ideal to have several ways of saying something, but if we can't limit the syntactic diversity we can in this case at least reduce the semantics variation. We could eliminate these forms:

```
X x3{1,2};
X* p2 = new X{1,2};
```

However, since **X{1,2}** must exist as an expression, the absence of these two syntactic forms would cause confusion, and they are the least verbose forms. Note that **new X{1,2}** must be interpreted as "an **X** allocated on the free store initialized by **{1,2}**" rather than "**new** applied to the expression **X{1,2}**"

Note that if we add a sequence constructor to vector, each of these definitions will create a vector of one element with the value **7.0**:

```
vector<double> v1  = { 7 };
vector<double> v2 { 7 };
vector<double> v3 ({ 7 });

auto p1 = new vector<double>{ 7 };
auto p2 = new vector<double>({ 7 });
```

We don't propose a syntax for saying "this is a sequence: don't treat is as a constructor argument list". We don't see a need, because if you don't know anything about a type, you shouldn't try to tell it how to initialize itself. Similarly, we don't propose a syntax for saying "this is an aggregate initializer, don't use it for a class with constructors".

Discussion: we think that the most likely confusion and common error from the new syntax will be related to initializer lists with a single argument. Consider:

```
vector<double> v2 { 7 };
```

A naïve reader will have no way of knowing that this creates a **vector** of one **double** initialized to **7.0** and not a **vector** of seven **doubles**. Obviously, making the second interpretation the correct one would be even worse. Consider

```
vector<double> v1 { };      // a vector with no elements
vector<double> v2 { 7 };    // a vector with one element
vector<double> v3 { 7, 8 }; // a vector with two elements
```

We feel that this must work as stated. This also eliminated the possibility of making the initialization of **v2** ambiguous. Consequently, we consider the proposed design the best possible (at least of the ones we have seen so far).

1.10 Assignment

We have discussed initializer lists in the context of initialization. However, we could imagine them used elsewhere. For example:

```
X v = {1,2};
v = {3,4};
v = v+{3,4};
v = {6,7}+v;
```

When we consider operators as syntactic sugar for functions, we naturally consider the above equivalent to

```
operator=(v,{3,4});
v = operator+(v,{3,4});
v = operator+({6,7},v);
```

It is therefore natural to extend the use of initializer lists to expressions. We have not explored this in detail and suggest that is should be explored. At least the use of the right hand of an assignment should be allowed to be an initializer list. We see no obvious problems with this general use of initializer lists and suspect that people will expect it to work if the simpler uses work.

Whether we should allow lists on the right hand side of an assignment is a separate issue. For example:

```
{a,b} = x;
```

We make no proposal or recommendation about this. It is a separate question.

5 Literal constructors

A literal constructor is a constructor that allows the compiler to determine the value of an object. That is, the constructor basically reduces to a constant expression, possibly involving addresses of statically allocated entities. For example:

```
class complex {
        double re, im;
```

```
public:
        complex"" (double r, double i) :re(r), im(i) { }
        // …
};
```

In the way we used the empty sequence notation, **{}**, to indicate "sequence constructor", we use the empty string notation, **"",** to indicate "literal constructor". A single quote (" or ') would cause lexical problems and that an "empty character" " could cause parsing problems and could be mistaken for a double quote (by humans).

Assume for a moment that complex have no other constructors. Then we get:

```
complex z  = { 1, 2 }; // ok
int i = 1;
complex z2 = { i, 2 }; // error:  i is not a constant expression
```

But naturally, most classes that have literal constructors will also have "ordinary" constructors. For example:

```
class complex {
        double re, im;
public:
        complex""(double r, double i) :re(r), im(i) { }      // handle constants
        complex(double r, double i) :re(r), im(i) { }        // handle all
        // …
};
```

To handle this smoothly, we need a new overloading rule: If all arguments of a constructor invocation are constant expressions, prefer a literal constructor to an "ordinary" constructor with the identical parameter types. For example:

```
complex z { 1, 2 };     // ok: call the literal constructor
int i = 1;
complex z2{ i, 2 };     // ok: call the ordinary constructor
```

The assumption is that making a constructor literal will help compilers identify static initialization and ROMable objects. This would be especially so for templates that would ordinarily not be evaluated until much later. For example:

```
const complex<double> z{1,2};
```

Only if it was known that complex had a constructor that was so simple that it reduced to a constant expression would a compiler attempt to optimize that by compile-time initialization.

Discussion: But what If I really want to invoke the literal constructor and want an error if it can't be used? One way would often be not to have ordinary and literal constructors with similar signatures. However, there is an argument that a user – not just the class designer – should have a way of saying "I want a literal constructor and nothing else!" Here's how:

```
complex z  = ""{ 1, 2 };       // ok: call the literal constructor
int i = 1;
complex z2 = ""{ i, 2 };       // error: can't call the literal constructor
```

As in the declaration of the literal constructor, we use the empty string to indicate that we want a literal. We are not at all sure that the **""{ … }** construct is necessary, so we just mention it in passing without proposing it. If you see a need for it, please demonstrate its utility.

Discussion: does the idea of literal constructors have sufficient value? The aim is to allow constructors in constant expressions and through that make simple class objects ROMable.

1.11 Destructors

Can an object constructed by a literal constructor be destroyed? Yes. The constructed object is "perfectly normal". In particular, use of a literal constructor does not imply that an object is constant. If you want a constant, use **const**. If you don't want a destructor called, don't define a destructor. The optimization opportunities for literal constructors are only significant for very simple classes.

6 Forwarding and literals

As we get more kinds of constructors the chance that two constructors do something very similar increases significantly. Examples and a literal and a non-literal constructor for initializing by constants and by variables and a sequence constructor that does something very similar to a template for initialization by an STL sequence.

```
class complex {
        double re, im;
public:
        complex""(double r, double i) :re(r), im(i) { }     // handle constants
        complex(double r, double i) :re(r), im(i) { }        // handle all
        // …
};
```

Saying exactly the same thing twice is sloppy and a maintenance hazard. This particular example is not too bad in practice (the initialization is minimal and trivial), but in general we need something better. The proposal for forwarding constructors (Glassborow ???) comes to our rescue:

```
class complex {
        double re, im;
public:
        complex""(double r, double i) :complex(r,i)  { }   // handle constants
        complex(double r, double i) :re(r), im(i) { }       // handle all
        // …
};
```

So the literal constructor says "initialize in the usual way, but only for constant expressions". If the "ordinary" constructor uses non-constant expressions in its implementation the forwarding fails. Note that this requires generalization of the notion of constant expressions to allow certain inline functions (Dos Reis ???).

> **Discussion:** We have the notion of a literal constructor and of a forwarding constructor. However, neither forwarding nor the notion literals are restricted to constructors. How do we, in general, forward from one function to another and how do we specify that a function should be callable in a constant expression?
>
> Generalizing the syntax used for forwarding constructors is not attractive:
>
> ```
> int f(int a, int b) { /* … */ }
> int f(int a) : f(a,0)  { ??? }
> ```
>
> How would we specify the return value? What would be the notational advantage over "plain old function calls"? We plan to return to the general issue of forwarding, but not in the context of forwarding constructors.

7 Defaults

By default you can do many things with a class (or objects of a class):

- Construct a default value
- Copy an object by a constructor
- Copy an object by an assignment
- Take the address of an object
- Use an object in a comma expression
- Derive from a class (and inherit a whole bunch of members)
- Allocate an object on free store (using **new**)

That is there is a number of "default features" with a default semantics that you can choose to change through the definition of functions (or sometimes wish you could):

- sometimes, a default is not suitable, so we want to suppress its use

- sometimes, we want to specify our own version of one of these operations (writing the appropriate function)
- sometimes, we want to be explicit about using the default.

The last reason comes from myths that have arisen about the suitability of the default copy operations. Sometimes, they are indeed unsuitable (typically when a class has a destructor), but when they are suitable, a user cannot define them better than the compiler.

Discussion: Is this the full set of default behaviors that we'd like to control?

The design questions are

- "How can you suppress defaults when they are not needed or not suitable?" ("give me the usual except the following: …")
- "How can you explicitly say that you want some or all of the defaults?" ("give me nothing except the following: …")

There have been suggestions for additional "default semantics" or facilities, such as "make all functions **virtual**", "implicitly define **==** and **!=**", "implicitly define **+=** based on **+** and **=**" "allow **x.f()** to be called as **f(x)**", etc. We don't propose any such extensions, but the possibility raises the design question of how to express such "additional defaults:

- "How can you request additional 'defaults'"? ("Given me the usual plus the following: …")

Consequently, we propose a mechanism for stating a list of defaults, for subtracting from the set of defaults, and for adding to the list of defaults.

First, we need a way of referring to each of the default operations. We could use function signatures, but that's verbose and indirect. Here is the suggestion for "names" for the implicit operations:

- Derivation **:** (so its absence is what is referred to as final in Java)
- Copy **=** (by constructor and by assignment)
- Address of **&** (only explicit use)
- Comma **,** (only explicit use)
- Construction **()** (default construction)
- Free store **new** (construct on free store)

The obvious weakness (which could be resolved by some more "magic" syntax) is that we cannot restore the default meaning of copy by copy constructor without also getting the default copy by assignment (and vice versa). We suspect that's actually a good thing because cases where we want the one copy operation default and not the other are hard to imagine. We can express that the listed defaults (and only the listed defaults) are available for a class using a "**default {** list of defaults **}**" syntax. Here are some examples:

```
class X {
    default { = () } // default copy and default construction
                     //  but no inheritance, & or ,
    // …
};

class Y {
    default { : }    // inheritance
                     // but no copying, default construction, &, or ,
    // …
};

class Z {
    default { : = () & , new }      // explicitly give me "all the usual"
    // …
};

class AE {
    default { }      // give me no defaults
    // …
};
```

We expect these cases to be among the most common.

> **Discussion:** Obviously we use **default** to avoid introducing a new keyword. It would be tempting to leave out the **{ }** after the default, but then we'd need a terminator. It is tempting to use a **default:** (note the colon) but then we'd need a terminator and a different notation for derivation.
>
> Why have we placed the specification of defaults within the class rather than modifying outside with the **class** keyword and the class name? Because most of the defaults have to do with the definition of operations and such operations are defined "inside" the class.
>
> **Discussion:** space is unusual as a separator in C++, but we can't use comma as it is one of the default operators.

The **default {** … **}** notation serves people who want an explicit statement that a default should be used. It is, however, verbose and somewhat brittle for people who just want to suppress a single default (without having to remember the full set of defaults). We can serve such users by a version of default that simply says "suppress the mentioned defaults. The syntax is based on the notion that we are subtracting from the set of defaults. For example:

```
class XX {
    default -{ : }   // no inheritance (“finally”)
                     // all the other defaults are available
    // …
};

class YY {
    default -{ = }   // no copy
                     // all the other defaults are available
    // …
};

class ZZ {
    default -{}      // suppress no default
                     // i.e., explicitly give me “all the usual”
    // …
};
```

That notation for “give me all the usual defaults” is convoluted, but then there is an even simpler way of getting all the usual defaults: just say nothing.

1.12 Access

Can we make the default operations **protected** or **private**? Yes, but we don’t propose to use this notation for that: if you want, say, a private default constructor, just declare one.

1.13 Should we mess with the defaults?

Since 1984 or so, people have suggested that copying should be prohibited under certain circumstances (e.g. for classes with pointers or classes with destructors). Similarly, people have suggested that a class with a virtual function should automatically have a virtual destructor. We are sympathetic, but note that these are separate issues.

We think that the right thing to do would be for the presence of a destructor to imply that default copy is suppressed (without any other effects). Suppressing copy if a pointer is present would cause problems with some PODs.

Making a destructor implicitly virtual if a class has a virtual function is attractive to avoid a common, but well-known error. However, it implies “magic” addition of functionality and overhead and is rumored to break a lot of COM code. Thus, we don’t propose it.

1.14 Should we add to the defaults?

Given the mechanism of removing all defaults and then selectively adding, we can consider what else we might want to add. People have asked for a feature like Java or C# interfaces. We could provide something like:

```
class Z {
    default +{ virtual  =0 }      // in addition to the usual defaults
                                  // make every function a pure virtual
    // …
};
```

We don't propose that, but note that what we have here is a general mechanism for adding class-wise semantic mechanisms.

1.15 Inherited constructors

One of the most frequently requested convenience features is "let me inherit the constructors from my base class. Except for a quirk of naming, we already have that! Consider:

```
class Base {
public:
        Base(int);
        Base();
        Base(double);

        void f(int);
        void f ();
        void f (double);

        // …
};

class Derived : public Base {
public:
        using Base::f;          // lift Base's f into Derived's scope
        void f(char);           // provide a new f
        void f(int);            // prefer this f to Base::f(int);

        using Base::Base;       // lift Base's f into Derived's scope
        Derived(char);          // provide a new constructor
        Derived(int);           // prefer this constructor to Base::Base(int);

        // …
};
```

Little more than a historical accident prevents using to work for a constructor as well as for an ordinary member function. Had a constructor been called "ctor" or "constructor" rather than being referred to by the name of their class, this would have worked. We propose this as the mechanism for inheriting constructors.

8 Casting

When a user-defined type it involved, we can define the meaning of C-style casting **(T)v** and functional style construction **T(v)** through constructors and conversion operators. However, we cannot change the meaning of a new-style cast and **T(v)** is by definition an old-style cast so its default meaning implies really nasty casts (reinterpret casts) for some built-in type combinations. For example, **int(p)** will convert a pointer **p** to an **int**. This leads to two common suggestions:

- Allow user-define **static_cast**, etc.
- Default **T(v)** to mean **static_cast<T>(v)** rather than **(T)v**.

The two suggestions are related because often the reason for wishing **T(v)** to mean **static_cast<T>(v)** is to be able to define it as a range-checked operation for some built-in type **T**.

We have also heard the suggestion that T(v) should be "proper construction" and thus not allow narrowing conversions (e.g. **char(123456)**). However, the functional notation is used to be explicit about narrowing, so banning narrowing by default would be too radical.

We don't propose to allow overloading of the new-style casts. If you want a different cast, you can define one using the same notational pattern, such as **lexical_cast<T>(v)**. The **T(v)** problem is worse, it basically defeats attempts to make casting safer and more visible. It also, takes the ideal syntax for the least desirable semantics. We conjecture that is not widely used for "nasty casts" (in correct code). To make any progress this conjecture must be tested through code analysis. Assuming that a change is possible, we propose the following:

- By default, **T(v)** is defined as **static_cast<T>(v)**
- The meaning of **T(v)** can be defined by a definition **T operator(V)** even if both **T** and **V** are built-in types.

These proposals are independent. Either would be useful without the other. The second proposal would break no existing code, but it would violate the dictum "a user-defined operator requires a user defined type as operand". The reason to accept that violation is that most of the built-in casting operations for **T(v)** are implementation defined or undefined anyway.

Either or (preferably) both of these proposals would allow a systematic notation and prevent unintentional bad casts.

> **Discussion:** If overloading of **static_cast**, etc. were allowed, the meaning of **(T)v** and **T(v)** would have to be reconsidered because their meaning is defined in terms of the

new style casts. We could consider providing no default for **T(v)** where **T** is a built-in type. However, that would almost certainly break a lot of correct code: people use the **T(v)** notation for conversions among the integral types, e.g. **int(d)**.

9 Rvalue constructors

Rvalue constructors (as defined in N???) do not appear to interfere with or be interfered by other proposals related to initialization. Had the rvalue semantic proposal simply affected constructors a syntax similar to the one proposed for sequence constructors and litaral constructors would have been more appropriate. For example:

```
class X {
        X={const X&);          // move constructor
        // …
};
```

However, the rvalue proposal appears to dig too far into the type system for that.

10 Acknowledgements

Obviously, much of this initializer list and constructor design came from earlier papers and discussions. The main papers are listed in Section 1.