Accredited Standards Committee X3
Information Processing Systems
Operating under the procedures of
American National Standards Institute

Doc No: X3J16/95-0051 WG21/N0651
Date: March 3, 1995 Page 1 of 15
Project: Programming Language C++
Ref Doc:
Reply to: Josee Lajoie
(josee@vnet.ibm.com)

Initialization Issues and Proposed Resolutions

1. implicit constructor calls in initializer list

   In an initializer list, constructor calls can be implicit.

```
struct S {
  S(int);
};
struct T {
  S s;
};
T t = { 9 };    // ok, equivalent to = {S(9)};
```

   This is already implied by 12.6.1 [class.expl.init]. Paragraph 2 shows an example where an initializer list for an array of classes initializes the array elements with both implicit and explicit calls to the class constructors. 12.6.1 needs to be expanded to take into account initialization for aggregate classes (and not just for aggregate arrays).

   Proposal:

   An object of a class type T with a user-declared constructor can be the member of an aggregate. An initializer list for the aggregate can provide an assignment-expression to initialize the member of type T. The assignment-expression can either be an explicit call to one of T's constructors, be an assignment-expression that can be treated as an argument for one of T's constructors,
| or be an assignment-expression of type T or or of a type that can be
| converted to type T.

2. What is the equivalent form of initialization when an initializer clause calls a constructor?

```
struct A {
   A(int);
};
struct B {
   A a;
};

B b = { 1 };
```

   The initialization for 'a' is equivalent to:

```
   A a = 1;
```

This is also implied by 12.6.1 [class.expl.init] which says that:
"each assignment-expression is treated as an argument in a constructor call ... _using the = form of initialization (8.5)_."
Again, 12.6.1 must be extended to discuss aggregate classes as well as aggregate arrays.

This implies that:
1) a constructor declared "explicit" must be called explicitly from an initializer list; an assignment-expression will not be taken as an argument for an "explicit" constructor.
2) The copy constructor for class A must be accessible otherwise the initialization
```
B b = { 1 };
```
is ill-formed.

3. Elided braces and ambiguities (the tough one ;-)

```
struct A {
  int i;
  operator int();
};
struct B { A a1; A a2; };
A a;
B b1 = { 1, a };  // does a initialize a2 or
                  // is a.operator int() called to initialize a2.i?
```

For this issue, the discussions on the reflector didn't converge towards a particular resolution. Two resolutions were proposed:

Proposals:

3.1 Brace elision only for POD class initializations

This resolution does not quite work since POD classes can have member functions. Further restrictions are required. Two restrictions were proposed. If the core WG decides to adopt option 3.1, it will need to adopt one of the following additional restrictions:

a) Change the definition of POD classes to say that they cannot have member functions.

```
struct A {
  int i;
  operator int();
};                          // A is not a POD class
struct B { A a1; A a2; }; // B is not a POD class
A a;
B b1 = { 1, a };          // error, must be fully braced
B b2 = { {1}, a };        // a initializes a2.
```

or:
b) When braces are elided in an initializer list for a POD class, user-defined conversions are not used to convert an assignment expression in the initializer list to the type of the object or reference being initialized.

```
struct A {
  int i;
  operator int();
};                          // A is a POD class
struct B { A a1; A a2; }; // B is a POD class
A a;
B b1 = { 1, a };          // a initializes a2.
```

3.2 Brace elision allowed for all aggregate member initializations

In this case, overload resolution decides which member is initialized by the assignment-expression in the initializer list.

If it finds that the assignment-expression can initialize both the member of class type and that member's first member (or, if this first member is itself of class type, the first member's first member, and so on) there is an ambiguity.

```
struct A {
  int i;
  operator int();
};
struct B { A a1; A a2; };
A a;
B b1 = { 1, a };          // error: ambiguous
                          // a can initialize a2 and a2.i
```

4. Redundant braces around scalar initializers

```
int j = i;
int j = { i }; // allowed in C. allowed in C++?
```

In C, redundant braces can be provided (I believe) for the initializers of struct members of scalar type as well.
i.e.

```
 struct S {
   int i;
   int j;
 } s = { {{ 1 }}, {{ 2 }} };  // OK
```

The resolution adopted for this issue should match the resolution we choose for issue 3 (eliding braces):

Proposals:

4.1 (to match 3.1): redundant braces limited to initialization for 'C like' objects:

The expression initializing an object of scalar type can be optionally enclosed in braces only if the object is a complete object or a subobject of a POD class.

or:
4.2 (to match 3.2): redundant braces allowed for any scalar object:

The expression initializing an object of scalar type can be optionally enclosed in braces.

5. empty initializer list

Currently, initialization lists are only allowed for aggregates. With the current wording [8.5.1, dcl.init.aggr], one can assume that empty initializer lists can initialize any aggregate.

```
struct A { int x; };
A a = {};   // ok
void f() {
  A a = {}; // ok
}
 class B {
  A a;
  int i;
} b = { {} , 5 }; //ok

class X { };
class C {
  X x;
  int i;
} c = { {} , 5 }; //ok, (as per core WG motion)
```

Proposal:

An empty initializer list can be specified as the initializer for any aggregate. The semantics for an empty initializer list are the same as the semantics for any aggregate initializer list that contains fewer initializers than there are members in the aggregate:
8.5.1 p1:
"... then the aggregate is padded with 0 of the appropriate type".
[More on this below. See issue 8].

6. aggregate with private and protected members

8.5.1 says:
"An aggregate is an array or an object of a class with ... no private or protected members, ..."
Should this refer to _nonstatic data_ members only?

```
class A {
  static int s;
public:
  int i;
} a = { 0 };     // ok?
```

```
    class B {
      int i;         // i is private
    } b = { 0 };     // definitely ill-formed
```

Proposal:

"no private or protected members" should be changed to "no private or protected _nonstatic data_ members". Relaxing this rule ensures that the restriction applies only to members affected by the initialization.

7. aggregates can have static members

An aggregate can have static members (either data or function). However, initializer list are not used to initialize the aggregate static data members. The static data member must be defined and initialized as described in 9.5.2 [class.static.data].

```
    struct A {
      int i;
      static int s;
      int j;
    } a = { 0, 5 };
```

The initializer list initializes 'a.i' to 0 and 'a.j' to 5.

8. When an aggregate is initialized with an initializer clause, if some members are initialized with constant expressions and other members are initialized with dynamic initialization, in which phase of initialization (3.6.2) are the members initialized?

Proposal:

- The members initialized with constant expressions are initialized before any dynamic initialization takes place.
- The members initialized with dynamic initialization are initialized when the dynamic initialization for the complete object takes place.

9. Default initialization to zero for objects of static storage duration

Question

What does initialization to zero mean for members of classes with a user-declared constructor?

```
    class A {
      int i;
      float f;
    public:
      A() : i(88) { }
    };
```

```
  class C {
    int j;
    A a;
  };

  C c;
```

What is the result of the default initialization to zero for 'c'?
Is 'j' guaranteed to be initialized to 0?
Is 'f' guaranteed to be initialized to 0.0?

What does initialization to zero mean for members of classes with a non-trivial constructor?

```
  class X {
    float f;
  public:
    virtual void f();
  };

  X x;
```

X has an implicitly-declared default constructor that is non-trivial (because X has a virtual member function). What is the result of the default initialization to zero for 'x'? Is 'f' guaranteed to be initialized to 0.0?

Proposal:

The default initialization to zero initializes all scalar members of an object of class type to zero converted to the appropriate type (and this recursively, for the scalar members of base and nested class members).

That is, in the examples above, c's members j and i are initialized to 0 and f is initialized to 0.0, and x's member f is initialized to 0.0.

Question

Do we need to define default initialization to zero for references and pointer to members?

| Proposal:
| ---
| The default initialization to zero gives a pointer to member the
| null member pointer value of that type.

10. How is an incomplete initializer list completed?

8.5.1 [dcl.init.aggr] says:
"If there are fewer initializers in the list than there are members in the aggregate, then the aggregate is padded with zeros of the appropriate types."

What does "padded to 0 of the appropriate type" mean?

| ANSI C says (3.5.7):
| "If there are fewer initializers in the brace-enclosed list
| than there are members of an aggregate, the remainder of the
| aggregate shall be initialized implicitly the same as objects
| that have static storage duration."
|
| Therefore, both initialization to zero (see 9.) and initialization
| by default constructors should be take place as the result of an
| imcomplete initializer list.
|
| Given the resolution to issue 2 above, I believe the right model for
| this is as follows:
|
| struct A {
| T1 m;
| T2 n;
| };
|
| struct B {
| A a;
| T3 o;
| } b = { { 99 } }; // identical to { { 99, T2() }, B() }
|
| Proposal:
| --------
| [This is a bit hard to describe in plain english ;-)]
| [Yes, human languages are worst than C++!   ]
| If there are fewer initializers in the list than there are members
| in the aggregate, the remainder of the aggregate shall be initialized
| implicitly the same as objects that have static storage duration;
| that is, the default initialization to 0 first takes place to
| initialize the uninitialized members of the aggregate (and this
| recursively for the members bases and members) and then dynamic
| initialization takes place for these members. The dynamic
| initialization takes place as follows: if the incomplete initializer
| list leaves a subaggregate imcompletely initialized, the
| uninitialized members of the subaggregate are initialized as if the
| initializer list was completed with expressions of the form 'T()'
| for each uninitialized member of the subaggregate (where 'T'
| represents the type of the uninitialized member); the remaining
| members of the aggregate are then initialized as if the initializer
| list was completed with expressions of the form 'T()' for each
| uninitialized member of the aggregate.

That is, given the type C defined in 9., the initialization

```
C c = { 22 };
```

is equivalent to:

```
{ 22, A() };
```

and after the initialization c.j holds the value 22, c.i holds the value 88 and c.f holds the value 0.0.

11. What is the meaning of T()?

5.2.3 [expr.type.conv] says:
"A simple-type-specifier followed by an empty pair of parentheses constructs a value of the specified type. If the type is a class with a user-declared default constructor, that constructor will be called; otherwise the result is the default value given to a static object of the specified type."

Given the class type C defined in 7.:

```
... C();
```

Is member 'f' guaranteed to be initialized to 0.0?

Here again, I believe the initialization caused by the syntax T() should behave consistently with the initialization of an object of static storage duration.

Proposal:

| An expression that is
a simple-type-specifier followed by an empty pair of parentheses
constructs a value of the specified type; the result is the default
value given to an object of static storage duration of the
specified type.

That is the default initialization to zero (see 9.) takes place. If the simple-type-specifier represents a class type, the class default constructor is also invoked to initialize the object.

For the example above, C() creates an object of type C with member c.j holding the value 0, member c.i holding the value 88 and member c.f holding the value 0.0.

12. t() in mem-initializer allowed for all types

Mike Anderson sent me the following issue:

```
struct A {
  A();
  ~A();
};

// non-template example
struct XA {
  A a;
  XA() : a() {}    // a(): clearly legal
};
```

```
struct Xi {
  int i;
  Xi() : i() {}    // i(): is this legal?
};

// template example
template <class T> struct X {
  T t;
  X<T>() : t() {}   // 'T()' valid; is 't()' valid ???
};

X<A> xA;
X<int> xi;
```

Mike indicated that for the same reasons we decided to give meaning to the syntax T() (so that it can be used in template definitions), we should also extend the mem-initializers syntax to allow 't()' for a member t of any type so that this syntax can be used in a template definition.

There are two possible meanings that could be given to the syntax 't()':

Proposals:

10.1 only performs dynamic initialization

"In a mem-initializer, the syntax 't()' can be used for a member t of any type T. If T is a class type, the default constructor for T is called; otherwise t has an undeterminate value."

The argument in favor of 10.1 is:
Since a mem-initializer is part of a constructor and since a constructor is responsible for the dynamic initialization of an object, default initialization to zero should never happen during dynamic initialization, that is, should never be triggered by a mem-initializer (because possibly very expensive) unless explicitly required by the user, that is, unless 't(T())' is specified.

or:

10.2 same semantics as default initialization to zero for statics

"In a mem-initializer, the syntax 't()' for a member t of type T means that t receives the same initial value as an object of static storage duration of type T."

The argument in favor of 10.2 is:
The call to a class default constructor always takes place for subobjects of class type whether the user explicitly calls the constructor with a mem-initializer or not. So currently, there is no difference in semantics whether 't()' appears in the ctor-initializer or not. One could assume that if the user

| writes 't()', the user wants the default initialization to zero
| to take place in addition to the dynamic initialization.
|
| Also, if T has an inaccessible copy constructor then the syntax:
| t(T())
| cannot be used to initialize 't' with the initial value given to
| an object of type T with static storage duration.

13. Initial value of a new expression

| [Jerry Schwarz, core-5185]:
| Does it make sense to have T() imply initialization but new T() not?
| Consider what I expect to be moderately common template code.
| T* array = (T*)malloc(sizeof(T)*10) ;
| for (int i = 0 ; i < 9 ; ++i) {
| new (&array[i]) T() ;
| array[i] = T() ; // needed to get initialization
| }
| The special casing we've done for the cast T() seems incomplete.
| In fact, on closer examination of 5.3.4 [expr.new] doesn't provide
| any semantics for the above if T isn't a class.
|
| There are two possible meanings that could be given to the syntax
| 'new T()':
|
| Proposals:
| ---------
| 11.1 same semantics as default initialization to zero for statics
|
| The syntax new T() means that the object created by new receives
| the same initial value as an object of static storage duration of
| type T.
|
| or:
| 11.2 only performs dynamic initialization
|
| For the expression
| new T()
| if T is a class type, the default constructor for T is called;
| otherwise t has an undeterminate value.
|
| [Andrew Koenig, core-5190]:
|
| I can imagine drawing the distinction that "new T;" implies the
| initialization of an auto variable and "new T();" implies the
| initialization of a static one. The only argument I can see against
| that one is that it is too clever.

14. What does initialization mean?

| There are currently many sections in the working paper that
| discuss initialization. However, the term 'initialization' means
| different things in these different sections.

(1) 8.5 [dcl.init] p5 distinguishes between "being initialized" and "having a constructor" or "starting off as zero".

(2) 6.7 [stmt.dcl] p4 distinguishes between "default initialization to zero" and "any other initialization".

(3) 12.6 [class.init] p1 refers to default construction as a kind of initialization.

(4) 5.3.4 [expr.new] p14 refers to construction as initialization for objects created by new expressions.

Proposal:

There are 2 steps in the initialization of objects of static storage duration:

1. default initialization to zero
   (see 9. for greater discussion on default initialization to zero)
2. user-defined initialization
   two forms:
   - o explicit - with initializers

   or:
   - o implicit - user-declared or implicitly-declared default constructor calls

An object of static storage duration is considered initialized once both default initialization to zero and user-defined initialization have completed.

For objects with automatic storage duration, objects with dynamic storage durations and temporary objects, only the user-defined initialization takes place. This means the initialization of these objects will be considered complete even though these objects may have undeterminate values.

8.5 p5 first sentence should be replaced with:
"For objects with static storage duration (3.7), default initialization to zero takes place before any user-defined initialization takes place."
The rest of 8.5 describes the initializers a user can specify for explicit user-defined initialization.

6.7 p4 first sentence should be replaced with:
"The default initialization to zero of all local objects with static storage duration (3.7) takes place before any user-defined initialization takes place."

12.6 p1 should be replaced with:
"Default initialization to 0 takes place for all class objects with static storage duration.

An object of a class type T with a user-declared constructor but without a default constructor must be explicitly initialized. The

| initializer for an object of type T can be of the form = or ();
| however, depending on the properties of T's constructors, some
| restrictions apply on the form of initializer that can be used,
| see 8.5. The initializer for a new expression creating an object
| of type T must be of the form (), see 5.3.4. If T does not have a
| default constructor, a temporary object of type T can only be
| created by using T's copy constructor (if accessible (11)) to
| copy an object already existing, see 12.2.
|
| For an object of a class type T' with a default constructor, if
| T' is an aggregate class type, an object of type T' can be
| initialized with an initializer list. Definitions and new
| expressions can initialize an object of type T' with an explicit
| call to T's default constructor. See 8.5 and 5.3.4. If an
| object of type T' is created by a definition or a new expression
| that does not explicitly initialize the object, T's default
| constructor is implicitly called. A temporary object of type T's
| can also be created by calling T's default constructor. In any
| of these situations, If T's default constructor cannot be
| implicitly defined (12.1) or if it is not accessible (11), the
| program is ill-formed.
|
| 5.3.4 p14 should be replaced with:
| Initializations in a new expression are of the form () (8.5).
| For objects created by a new expression of the form 'new T' or
| 'new T()', if T is a class type, T's default constructor is called;
| otherwise the object starts with an undeterminate value [the exact
| wording for this depends on the resolution for issue 13]. When
| creating an object of type T, the new initializer can specify a
| single expression of type T or of a type that can be converted to
| T. If T is a class type with a user-declared constructors, the
| new initializer can specify a list of expressions that is taken as
| an argument list for one of T's constructor.
|
| 8.5 p5 should be split to describe which semantics apply to
| initialization of the form = and which semantics apply to
| initialization of the form ().
| For example p10, bullets 2, 3, 4 describe initialization that
| can take place only when the = form of initialization is used,
| bullet 5 describes initialization that can take place only when
| the ( ) form of initialization is used, while initialization
| described by bullets 1, 6 and 7 can take place with both forms
| of initialization.

15. Initialization of const objects

| 7.1.5.1/p2 [dcl.type.cv]:
| "Unless explicitly declared extern, a const object does not have
| external linkage and shall be initialized (8.5, 12.1)."
|
| struct A {
| int i;
| int j;
| };

```
|   const A a; // well-formed?
|   A has a trivial default constructor that does nothing.  Does A's
|   trivial default constructor count as initialization?
|
|
|   struct B {
|     int i;
|     virtual int f();
|   };
|   const B b; // well-formed?
|
|   B has a default constructor that is not trivial (because B has a
|   virtual function).  However, B's constructor does not initialize B's
|   nonstatic data member i.  Is the definition of 'b' well-formed?
|
|
|   struct C {
|     int i;
|     int j;
|     C() : i(99) { /* does nothing */ }
|   };
|   const C c;
|
|   C has a default constructor that is user-declared.  However, C's
|   constructor does not initialize C's nonstatic data members i and j.
|   Is the definition of 'c' well-formed?
|
|
| That is, can we assume that:
| 1. after an implicitly-declared trivial constructor has completed,
|    the object is initialized?
| 2. after an implicitly-declared non-trivial constructor has
|    completed, the object is initialized?
| 3. after a user-declared constructor has completed, the object is
|    initialized?
|
| 13.1  Objects that are not of class type
|
|   The reflector discussions converged towards the following
|   resolution for objects that are not of class type:
|
|   Proposal
|   --------
|     a const object of a type that is not a class type must be
|     explicitly initialized with an initializer.
|
|   That is,
|
|       const T t ;  // ill-formed
|       const T t = T() ;  // well-formed
|
|   if T is not a class type.
```

This means that, for a declaration at namespace scope of an object of a type that is not a class type, the declaration must specify an explicit initializer; the "default initialization to zero" is not sufficient initialization for this object.

```
const int zero ; // ill-formed, must be explicitly initialized
```

13.2 Objects of class type

Proposals:

13.2.1 7.1.5.1 p2 is a syntactic constraint for objects of class type

[Jerry Schwarz, core-5199]:

I think we should interpret this constraint as a simple syntactic one rather than trying to distinguish cases in which the object is semantically initialized. I would revise this to:

"Unless explicitly ... and its declarator shall contain an initializer (8.5)."

This would result in:

```
const T t ;  // always ill-formed

const T t = T() ;  // well-formed
                   // (provided copy and default construction are
                   //  allowed)
```

or:

13.2.2 Any default constructor is initialization

[Bill Gibbons, core-5224]:
Const objects of class type must be explicitly initialized OR must have a default constructor (either user- or implicitly-declared).

This implies that the definitions of ’a’, ’b’ and ’c’ in the examples above are well-formed.

or:

13.2.3 An implicitly-declared default constructor is not initialization

[Fergus Henderson, core-5237]:
A class type is said to have implicit user-declared default initialization if either
- the class has a user-declared default constructor or
- all its bases that are of a class with nonstatic data members have a user-declared default constructor and all its non-static data members are of class types (or array thereof) with implicit user-declared default initialization.

Const objects of class type must be explicitly initialized unless the class type has implicit user-declared default initialization.

| This implies that the definition 'c' in the examples above is
| well-formed while the definitions of 'a' and 'b' are ill-formed.
|

| 16. Is the user required to initialize a const volatile object?
|
| int const   ci ;   // ill-formed
| int const volatile cvi ;   // ?
|
| Proposal
| --------
| const volatile objects are not required to be initialized.
|
| Since volatile objects can have their value set in ways unknown to
| the program, the language should not require that volatile or const
| volatile objects be initialized.