Document no: **X3j16 93-0053**
WG21 N0260
Date: 28 March, 1993
Project: Programming Language C++
Reply to: John Max Skaller
maxtal@extro.ucc.su.oz.au
100236.1703@compuserve.com

In-Class Inititialisers for Class Members and Bases

DRAFT. Version 0.3

Abstract

This proposal recommends extensions to C++ allowing:

1) Default initialisers for non-static class data members and bases
2) In-class definition of static data members with initialisers
3) Optional automagical allocation of static members initialised in class

which together allow

1) Guarranteed initialisation of all class members, and to sensible values
2) Encapsulation of all specification of a class in the class declarative region
3) Enhanced robustness of classes to modification
4) Completion of the replacement of tentative declarations with full specifications
5) Considerable reduction of coding effort for classes with many variables and constructors

while being

1) Compatible with existing code
2) Easy to implement, except for automagical allocation
3) Natural for programmers, requiring little explanation

whereas automagical allocation is already required for static member of template classes and functions, which is available already in some compilers. Certain complications are introduced when considering order of initialialisation, some of these are already present in the WP.

Organisation:

0) New Terminology

0.1) In-class

In-class refers to syntax given in the region of a class declaration. See 92-1105,N0023, "Resolved issues of name lookup", Scott Turner, the region of a class declaration is the textual (lexical) body of the declaration, also called called class scope.

For example, in

```
class X {
  int f(){ return 1;}
  int g();
};
int X::g(){return 2;}
```

the definition of `f()` is given 'in-class', whereas the definition of `g()` is given 'out-of-class'.

0.2) In-class Initialiser

An initialising expression with the usual form for initialisation of a class member or base which is given in-class.

```
class X : Y(sin(3)+3) {
  static int i=sin(3)+3;
  static int j;
  int m=sin(3)+3;
  int n;
};
int X::j=cos(4)+4;
int X::n=cos(4)+4;
```

the initialiser `sin(3)+3` given for `i`, `m` and `Y` is an in-class initialiser, whereas the initialiser `cos(4)+4` is an out-of-class initialiser.

0.3) Trivial constructor.

[See also the paper X3J16/92-0125, WG21/N0202, "Trivial Default Constructors" by Tom Pennello: our notion of trivial constructors is almost the same as Tom's, however, we extend the notion by specifying that all standard types and aggregates have trivial constructors, so that every type has at least one constructor -- even if it happens to be trivial. Not every type need have a default constructor though.]

In this text we will re-interpret the meaning of 'no constructor' slightly. If a class has no explictly given constructors, then if any member or base has a constructor, then a default constructor will be generated by the compiler which invokes the default constructors for those members or bases that have them.

In the special case that no member or base has a constructor, then had such a constructor been generated it would have initialised nothing, and so the constructor is not generated. It is said the class has no constructors.

In this text we will pretend that the default contructor is in fact generated and it is called a trivial constructor. The reason for this is that in the presence of default initialisers, the compiler generated default constructor will not be trivial, and the class will no longer have no constructors.

In this context we can say that all standard types and all types without explict constructors have default constructors, even if these happen to be trivial, except if no default constructor is given and some other constructor is explicitly given.

For example 'int' has a trivial constructor. The use of this terminology simplifies the wording.

0.4) Default initialiser.

A default initialiser is an initialiser specified for a non-static data member or base which can be overriden by a user specified mem initialiser in a constructor definition, but which otherwise is used in place of the default constructor where such a default constructor would be implicitly invoked, if it existed, and provided an explicit mem-initialiser is not given.

For example,

```
class X {
  int i=0;
  int j=1;
  string s="Hello";
  X(int k) i(k){}
};
```

is equivalent to

```
class X {
  int i;
  int j;
  string s;
  X() : i(0), j(1), s("Hello") {}
  X(int k) : i(k), j(1), s("Hello") {}
};
```

0.5) In class definition.

An in-class definition is a static variable with an explicit in-class initialiser. For example, in

```
class Y {
  static int i;
  static int j=3;
  static float pi = 3.1;
  static const int n=7;
};
```

Each of the static data members `j`,`pi`,`n` is an in-class definition because an initialiser is given. However, `i` has only a declaration and a definition must be supplied.

0.6) Compiler (automagical) Allocation.

Previously, static data members had to be allocated manually. (That is, explicitly out-of-class by the programmer). Automagicall allocation refers to the allocation of storage for a function or data member with an in-class definition by the compiler.

The word 'automagical' is chosen because it is a popular description of silent and sometimes unexpected actions of the compiler, and the more appropriate phrase 'automatic allocation' already has a disparate technical meaning in C++.

The word 'allocation' is used specifically to distinguish the specification of the initialiser and the allocation of the storage for a variable. Normally, these two mechanisms are combined for variables, in which case they are called a definition. A definition may also act as a declaration.

For manually allocated static class members with in-class initialisers, the specification of the initialiser occurs in-class while the allocation occurs out of class. Neither mention of the variable seems to fit the usual meaning of definition. We have chosen to term the in-class declaration with an initialiser a definition, as it has the syntax of a definition, and, if automagical allocation is used, there are no other statements which could constitute a definition.

This draft discusses the merits of and alternatives to the automagical allocation of static members.

0.7) Compiler constant.

We define a compiler constant expression recursively. A constant integral type or enumerator initialised by a compiler constant expression is called a compiler constant. An expression containing only compiler constants previously initialised in the same translation unit and integral literals is a compiler constant expression. (floating constants may be used if explicitly cast to integers, and the sizeof operator provided the size of its argument os known) This definition differs slightly from that of a constant expression in that prior initialisation in the same translation unit is required. Compiler constants are necessarily immutable romable objects. See Appendix A.

1) In-class Initialiser Syntax.

For members the usual forms

```
T v = expr                (IC1)
T v(list)                 (IC2)
T v = T(list)             (IC3)
T v = bracelist           (IC4)
```

and for bases the forms

```
T = expr                  (IC5)
T(list)                   (IC6)
```

can be used as would normally apply for an object of type T. A '*list*' is a comma separated list of '*expr*', and a '*bracelist*' is the usual brace enclosed initialiser form with elements of '*expr*'. Note that there is no way (unfortunately) to specify default initialisation other than

```
T v = T()
```

1.1) Type. T can be a cv-qualified object type, pointer or a reference, except,

1.2) T cannot be a function (although it can be a pointer to a function). Form (IC1) might be allowed as a short form of the usual inlined wrapper function used to effect renaming, however I didn't think it was worth the effort.

Examples.

```
class complex1 {
      float real;
      float imaginary;
}; // has trivial contructor

class complex2 {
      float real=0;
      float imaginary=0;
      complex2(){}
      complex2(float r) : real(r) {}
};

class X : public Y=20, Z(20) {
  int i =20;
  int j(30);
  char *p = "Hello";
  const int x=10;
  string s("Hello",20);
  complex1 z={1.2,3.4};
  complex2 z2,z3(1);
  static int count=0;
};
```

2) Semantics - Default initialisers

2.1) Invokation. An in-class default initialiser is used whenever a constructor implementation would implicitly call a default constructor if one existed, provided that an in-class initialiser is specified.

2.2) Guarrantee of initialisation. Previously, if a class has no explicitly given constructors, the compiler will generate a default constructor (possibly trivial).

In the presence of a default initialiser, a class cannot have a trivial constructor: the presence of a default initialiser guarrantees that the member with which it is associated is initialised, however this does not change the rule that prevents a default constructor being generated by the compiler if other constructors are explicitly defined.

2.3) Name Lookup: Rewriting rule. The default initialiser, when invoked, is treated as if it had been written in an inline version of the constructor definition for which it is invoked. (But the parameters of the constructor prototype may not be refered to). That is, it is evaluated in full class scope.

2.3a) Comment. As for mem-initialisers, it is not legal to refer to the value of an uninitialised member, however, the address may be taken.

ALTERNATIVE. The default initialisers are not rewritten but evaluated in the scope in which they are written. This prevents reference to uninitialised non-static members, which is usually desirable, however it also prevents access to unseen functions, unseen static variables, and addresses of unseen members. Since default initialisers will usually be constants, this may not be a large constraint, however, it may influence the programmer to reorganise the order or declarations within a class, and may interfere with ensuring contiguous allocations if changes in protection status ensue.

2.4.1) Example.

```
class string {
public:
  string(){ ...}
  string(char*){ ...}
};

class X {
public:
  string s="Hello";
  X(){}
};
```

The default constructor for X would normally be interpreted as if written

```
X() : s() {}
```

that is, the default constructor for 'string' is called for 's' implicitly The presence of the default initialiser changes this behaviour so the interpretation is equivalent to

```
X() : s("Hello") {}
```

2.4.2) Example.

```
class Y {
public:
  int i=0;
  Y(){}
};
```

Normally, the default constructor for class Y would be accepted as written, that is, no constructor for the int 'i' would be invoked because 'int' does not have a non-trivial default constructor. Thus 'i' would not be initialised.

The presence of the default initialiser ensures that the default constructor for Y is equivalent to

```
Y() : i(0) {}
```

since a constructor for 'i' would have been called implicitly had 'i' been of a type that had a default constructor. (Or, alternatively, one considers that 'int' indeed has a default constructor that happens to be trivial).

2.4.3) Example.

```
class A {
public:
  int j=0;
  A() : j(1) {}
};
```

Here, the default initialiser for 'j' is not used by the default constructor, since an initialiser is explicitly provided in the constructor definition.

Note also the default initialiser is not used in the compiler generated copy constructor, since that constructor would not invoke the default initialisers for any members or bases of a class, and in particular would invoke the 'copy constructor' for 'int' on corresponding members for 'j'.

2.4.4) Example.

```
class E {
public:
  int i=one;
  E(){}
  enum {one=1};
};
```

This example is treated as if it had been written

```
class E {
public:
  int i;
  E() : i(one) {}
  enum {one=1};
};
```

in which case a second rewriting under existing language rules ensures that 'one' is visible as follows:

```
class E {
public:
  int i;
  E();
  enum {one=1};
};
E::E() : i(one) {}
```

2.4.5) Example.

```
class N {
public:
  int i=0;
};
```

Without the default initialiser, class N would have had a trivial constructor, and 'i' would not be initialised. With the default initialiser, the compiler will generate a default constructor that initialises 'i' to 0.

2.4.6) Example

```
class Q {
public:
  int i=0;
  int k;
  Q(int j) : k(j) {}
};
```

Class Q does not have a an explicit default constructor, but it does have an explicit constructor, so no default constructor, trivial or otherwise, will be generated, and thus instances cannot be created without explicit initialisers, as usual.

The compiler will generate a copy constructor equivalent to

```
Q(const Q& q) : i(q.i), j(q.j) {}
```

and so the default initialiser for 'i' is not invoked. However, the explictly given constructor is equivalent to

```
Q(int j) : i(0), k(j) {}
```

2.4.7) Example

```
class R {
public:
  int i=0;
  int j;
  Q(const Q& q) : j(q.j) {}
};
```

Here the copy constructor is given explicitly, and so the trivial constructor for 'i' would normally be invoked, leaving it uninitialised, however the presence of the default initialiser again ensures that 'i' is in fact initialised to 0 (but not to q.i).

2.4.8) Example

```
class S {
public:
  S(int);
};

class D : public S(0) {}
```

Here the base S of D is given an explicit initialiser, which is invoked by the compiler generated default initialiser for D, as if

```
D() : S(0) {}
```

had been written.

2.4.9) Example

```
class Random {
public:
  float r=random();
};

Random r1, r2, r3;
```

Here the variables r1, r2 and r3 are initialised by calls to the random number generator random, and it is expected that they will have different values for 'r'.

2.5) Rationale

The principal purpose of default initialisers is to ***guarrantee initialisation*** of nonstatic data members and bases, and to do so visibly ***in the class declaration***.

The ***order of initialisation*** and evaluation of the initialisers is natural and ***in the order of writing.*** (Except for virtual bases preceding non-virtual ones).

The value to which the members or bases are initialised need not be the supplied default, since it is permitted to override the default in any constructor with a constructor specific mem-initialiser. (Naturally, one can also assign a new value in the constructor body).

It is not necessary that the values to which default initialised members are initialised be the same for all objects: the value may depend on temporal context (or details of external linkages).

It is an error, however, for the value to be computed to yield a different value depending on lexical context; this is considered as a violation of the one definition rule in the same way as for mem-initialisers. (Since default initialisers are precisely default mem-initialisers)

A secondary but important effect of default initialisers is to ***reduce coding*** in certain situations. For example:

```
class Many {
public:
  const long l;
  const int i;
  const unsigned u;
  const float f;
  const double d;

  Many()              : l(0),  i(0),  u(0),  f(0),  d(0)  {}
  Many(long ll)       : l(ll), i(0),  u(0),  f(0),  d(0)  {}
  Many(int ii)        : l(0),  i(ii), u(0),  f(0),  d(0)  {}
  Many(unsigned uu)   : l(0),  i(0),  u(uu), f(0),  d(0)  {}
  Many(float ff)      : l(0),  i(0),  u(0),  f(ff), d(0)  {}
  Many(double dd)     : l(0),  i(0),  u(0),  f(0),  d(dd) {}
};
```

can be more cleanly written

```
class Many {
public:
  const long l=0;
  const int i=0;
  const unsigned u=0;
  const float f=0;
  const double d=0;

  Many()                     {}
  Many(long ll)       : l(ll) {}
  Many(int ii)        : i(ii) {}
  Many(unsigned uu)   : u(uu) {}
  Many(float ff)      : f(ff) {}
  Many(double dd)     : d(dd) {}
};
```

The advantages are clear in the case where new data members are added to a class: it is all too easy to forget to initialise these new members in all constructors, especially if the constructors are not inline but coded in separate translation units.

Provision of in-class initialisers allows ***verification of initialisation by inspection of the class declaration*** alone.

Because the semantics of in-class member and base initialisers is provided by a rewritting rule, use of such defaults is not expected to cause any great difficulties for either the compiler or programmers.

In Class definition of static members

3.1) Syntax. The forms (IC1) thru (IC4) can be applied to static data members in-class.

3.2) Manual Allocation. If a static member has an in-class initialiser, the user may provide an out-of-class allocation, however that allocation may not have an explict initialiser. Instead, the in-class initialiser is applied.

3.3) Example.

```
class X {
public:
  static int i = 10;
  static int j = 20;
};

int X::i; // initialised to 10
int X::j=20; // error, duplicated initialiser
```

3.4) If the user does not supply an in-class initialiser for a static data member, then an out of class definition is mandatory, as usual.

3.5) Example

```
class X {
public:
  static int i;
};

// a definition is not supplied, causing a linker error
```

3.6) Automagical Allocation. However, if the user does not supply an out of class allocation for a static data member with an in class definition, then the processor will allocate storage and initialise the objects automagically.

See below for discussion of alternatives.

3.7) Example

```
class X {
public:
  static int i=20;
};

// a definition is not supplied,
// so the compiler generates one automatically
```

3.8) Example.

```
class X {
  static int i;
  static int j;
  static int k=0;
  static int m;
  static int n=0;
};
int X::i; // not initialised
int X::j=0;
int X::k; // initialised to 0
// m must be allocated somewhere else by the user
// n is allocated somewhere by the compiler and initialised to 0
```

ALTERNATIVE. We considered allowing automagical allocation of all static members not allocated by the user. This was rejected on the grounds of quiet changes. Accidental removal of an allocation with an initialiser (definition) would cause a compiler allocated variable with a default initialiser to be generated, silently initialising the variable to a different value.

Addition of a new static member without an in-class initialiser would cause default initialisation if the user forgot to supply an allocation with an initialiser (definition), rather than a linker error.

ALTERNATIVE. We considered specifying that all static members must be manually allocated, and the initialisers are merely defaults. This is viable and simpler than requiring automagical allocation by the compiler, however, it defeats the simple and recommended technique of initialising all variables and never allocating them, which ensures all variables are properly initialised by generating a linker error if a static member is not given an initialiser (since in this case an allocator is not generated by the compiler).

See below for further discsussion, this alternative is recommended if automagiacl allocation is rejected as too hard.

3.9) Addressability. The address of such a member may be taken, and the member need not be const. The address of the member is unique (there is only one allocation).

3.10) Name Lookup: The initialiser is evaluated in class scope without rewriting.

3.11) Order of Initialisation. When the compiler generates allocations (definitions) automagically for static members with initialisers, it will generate all such allocations for a given class in sequence, with no intervening other code.

ALTERNATIVE. In class initialisers are initialised in order of writing, whether of not they are automagically allocated. This makes the position of the user allocations when in-class initialisers are used irrelevant.

ALTERNATIVE. For either of the above cases, the in-class initialisations are guarranteed to be done first, before any out of class initialisations.

3.11a) Order of evaluation of initialisers. Initialisers are evaluated in the order of initialisation.

3.12) Inlining. If a const romable object is initialised by an expression which can be evaluated at compile time, the compiler may 'inline' the use of the object. However, a non-inlined object must still be allocated (unless the object is private and all member functions are inline and the compiler can deduce the object is never addressed). The reason is that static data members have external linkage, and must be uniquely addressable.

3.13) Compile time constants. If a const integral member is initialised by a const-expression, the member is immutable and its value known at compile time, and it may be used to specify array bound or enumeration limits.

3.14) Example.

```
class Z {
public:
  static const int n=20;
  static const int maxi=32000;
  enum {joe=n, fred=maxi};
  char c[n+1+fred];
};
```

3.15) Rationale.

The main purpose of allowing in-class initialisers for static members is to provide an ***in-class guarrantee of initialisation.***

With automagical allocation, there is a further advantage that the ***textual body of the class region may be a complete class*** (that is, it can be cut and paste in one go).

In the special case of compiler constants (integral members initialised by const-expressions known at compile time), the member may also be used in const expressions and in particular in ***specifying array extents***.

Automagical allocation is not mandatory because in some cases non-default order of initialisation may need to be controlled by the programmer. However, I cant think of a single such case where the order cannot be determined in-class, which is superior.

In-class initialisation is not mandatory for all static members, both for compatibility, efficiency, and in cases where the initial value is intended to be hidden as an implementation detail.

3.16) Implementation difficulties with automagical allocation.

The requirements of this section are that the object be allocated once. This places a burden on the compiler/linker to ensure there is a single allocation.

There are a number of schemes to do this, one is to defer allocation until after all modules have been compiled, then to create a dummy module with one allocation for each such object, (use the file system to resolve the problem) another is to allocate space in separate object modules each time and merge the definitions at link time.

Many linkers already have facilities for this, for example, Borlands linkers should have no problems with automagical allocation, as it has facilities for overlaid allocations. (Which are use for template instantiation, vtbl merging and for automagical allocation required in C).

Older linker may also have facilities that can be used for this. The mechanism required is very much like that used by the FORTRAN named COMMON facility.

Systems with integrated environments should have no problems at all, considering.

As this problem already exists for template classes (and perhaps functions) with static variables, and as a similar problem exists in C for extern variables, an objection on grounds of implementation difficulty is not entirely sustainable.

4) Discussion, Rationale, Alternatives.

The automagical allocation of static variables is contentious. If this requirement is considered too hard, the alternative of user supplied allocations is available: the proposal should not be rejected solely on grounds that automagical allocation is too hard.

There are some issues concerning initialisation of constants to be cleaned up. The rewriting rule needs to be be weighed against the no-rewriting rule.

In weighing up this proposal, it is necessary to consider the disadvantages and advantages, and the alternatives.

The principal <u>advantages</u> of this proposal are:

A.1) It is possible to specify the complete information required for a class in the region of the class declaration. This allows 'porting' of the class as a single textual unit.

This advantage should not be trivialised: it was a principal determinant in Meyers decision NOT to separate class interface and implementation in Eiffel.

Again, in C, and even more so in C++, having to separate declarations in header files from the implementing body files is often a source of considerable frustration and must be counted as a major disadvantage of this style.

This proposal provides a similar escape route to that provide by inline functions, which are automagically allocated if necessary.

A.2) It is possible for the programmer to guarrantee all data members of a class are initialised by providing an in-class initialiser.

This is probably the crucial attribute of the proposal. Failing to initialise a variable is a serious problem and can be particularly hard to track down. This is especially true for static members/global objects, since debugging may commence with instructions executed
after the run-time system has performed initialisation.

For non static pointer members, the problem is also accute: even on systems with hardware protection which detects accesses at the 0 pointer, a pointer member in an object might begin uninitialised with any value, including a pointer to a valid existing object, which is common when frequent allocations and deallocations of the same class occur. The problem is then detected, if at all, as a magical change to some object occuring at apparently random times.

A guarrantee of initialisation already exists for class types with a constructor, and standard types which are statically allocated.

A.3) The guarrantee is in the form of a contract between the programmer and compiler:

IF the programmer provides (legal) initialisers for one, several, or all data members,
THEN the compiler will ensure all data memberswith such initialisers are initialised.

In particular, if ALL data members have initialisers, then all data members will always be initialised.

The form of the contract ensures localised cost: efficiency can be traded off against guarranteed initialisation by the programmer.

A.3) It is possible to ascertain which data members have a guarrantee of initialisation, and in particular, if the all the data members have this guarrantee of initialisation by examining the region of the class declaration alone.

In particular it is not necessary to examine the bodies of each and every explicitly given constructor, and to calculate the effects of compiler generated constructors, in order to determine whether the guarrantee applies.

It is also not necessary to know whether a given type has a constructor, which makes the extra assurance of a visible initialiser all the more robust in the presence of changes to class specifications.

A.4) The order of evaluation of default initialisers, and initialisation of their respective of non-static members will always be that of writing (except that virtual bases are initialised before non-virtual ones).

This is not the case with member initialisers, which is a frequent source of errors. One I have done myself is:

```
 class string {
    int len;
    char *data;
  public:
    string(char *p) : data(p), len(strlen(data)) {}
  };
  // error, len is initialised before data
```

This would not be so easy to miss had I written

```
  class string {
    int len=strlen(data);  //compiler reports possible use
                           // of uninitialised value of data
    char *data="";
  public:
    string(char *p) : data(p) {}
  };
```

which is unlikely as

```
  class string {
    char *data="";
    int len=strlen(data);
  public:
    string(char *p) : data(p) {}
  };
```

is clearly better ordered.

A.5) The order of evaluation of automagically allocated static variables is the order of writing. Therefore, deliberately eliding out-of-class allocations ensures a consistent ordering. The order can be overridden by the user supplying some or all of the allocations.

A.6) The overriding of the ordering of allocations does not require (or allow) supplying an initialiser. For static initialisers, the in-class initialiser is exactly the one that is always used.

A.7) The automagical allocation of static class members implies the same technique can be used for static variables in inline functions, and static members of template classes.

Conversely, solving the allocation problem for these cases implies a ready mechanism which can be used for in-class initialisers.

A.8) The proposal will not break any existing code.

A.9) Eiffel offers some prior art, as do templates and inline functions. The FORTRAN named COMMON block provides a similar mechanism.

The principal disadvantages of this proposal are:

D.1) Whereas the initialisers for non-static members are merely defaults, but the order of their application and evaluation is fixed in the order of declaration and writing, the initialisers for static members are not defaults and cannot be overriden, but the order of evaluation can be overriden by user allocation.

D.2) The appearance of default initialisers ensures initialisation -- but not necessarily to the value of the default initialiser. This may be a little misleading. This is always the case with defaulting strategies, however.

D.3) The compiler cannot enforce the one definition rule (ODR) for in-class initialisers. This is a new problem for static member initialisers, since previously multiple definitions would be detected by the linker.

However, this problem already exists for inline functions, including inline constructor mem-initialisers, and it exists for templates as well.

Further, the problem is not expected to be serious. Most initialisers will be compile time evaluable constants. Those that are not are very unlikely to be highly context sensitive.

D.4) Automagical allocation is difficult. Where are the variables allocated, and how are they initialised?

(Discussion defered till a later section because this is a major (the major) issue.

Alternatives.

We considered allowing in class initialisers for static members to be defaults. This must be rejected on grounds given before, namely, sensitivity to addition or removal of an actual overriding initialiser.

The same argument does not apply to non-static initialisation because the number of constructors is not limited to one (as is the number of allocations of a static variable). Also, static variables are always allocated out of class and thus often out of sight, whereas constructors may be written inline.

There are several alternatives to automagical allocation for static members.

All of these alternatives still allow in-class initialisation of static class members.

A) Stick with the status quo: require user allocation.

This is the principal alternative, and an acceptable fallback position.

Consequence: defeats the ability to encapsulate the whole of a class in the region of its declaration.

B) Allow automagical allocation ONLY for constant integral members whose initialisers are const expressions.

This is a compromise position. It allows (but does not require) automagical allocation of a special but important set of cases.

The main disadvantage of this is that confusion as to which members may be automagically allocated may exist.

This same confusion already exists when determining the order of initialisation of global/static variables.

Note that an enumeration constants work exactly like this.

The confusion may be elided by requring these cases not be user allocated. In addition it may be possible to specify that such cases have internal linkage, in which case the constants can be allocated in each translation unit that takes the address of such a member.

Automagical allocation - do we already need it?

We already need a form of automagical allocation in C. C cannot distinguish a mere declaration of a variable from a definition in the case where no initialiser is given.

```
  extern int x;
```

is a declaration, yet there need not be a definition given anywhere. C++ mandates a separate definition must be given, one of

```
  extern int x=0; // definition
  int x;          // definition (and declaration)
  int x=0;        // definition (and declaration)
```

must be specified. This is not true in C, so C already must solve the automagical allocation problem.

In C++, a template class might contain a static variable:

```
  template<class T> class X { static int i; };
```

It is not yet determined what must now be done to ensure a single allocation of X<T>::i for each T. Even if an out-of class 'definition' is given:

```
 template<class T> X<T>::i;
```

this does not constitute an allocation of a particular instance of the template class. In fact, in-class initialisation of template static members has no real affect on the instantiation allocation problem, the case of

```
  template<class T> class X {
    static int i=0; // no out of class definiton given
  };
```

still requires automagical allocation. Since the problem MUST be solved for templates (unless we ban static members of template classes), there seems no good reason not to extend automagical allocation to non-templated classes.

There is every reason, however, to allow in-class initialisers in template classes, where they would be particularly valuable.

A similar but more problematic issue arises for static data in template and inline functions:

```
  inline int f() { static int j=1; ++j; return j;}
```

Here, separate instantiations of the inline function may cause separate allocations of 'j', leading to unpredicatable results. Yet we cannot easily ban this behaviour, since semantically the keyword 'inline' has no meaning---it is only a hint to the compiler (except that inline implies internal linkage).

Further, it is clearly not desirable to ban use of named constants in inlined functions:

```
  inline int g(int i) { const n=20; return i*n;}
```

but separating this case depends on the highly special concept of an integral type initialised by a 'const-expr' (which is not the same thing as the more generic notion of a constant expression: in the cases

```
        const float pi = 3.141;
        const int ipi = 3.141;
```

neither pi or ipi is a const-expr).

It would seem therefore that automagical allocation of static variables is already desirable, if not required.

Automagical allocation - how hard is it?

I cant answer this question easily. I call for opinions or facts from compiler vendors, or theoreticians with ideas on how this might be done.

However, I believe:

Those compiler vendors providing their own linkers will have absolutely no problem. I believe, for example, that Borland would have no problem with automagical allocation, since it already has a mechanism for merging definitions of templated functions, literals, virtual tables and the like.

Those vendors restricted to older linker technology still might have no problem with a linker solution, since existing linkers work with both C and FORTRAN.

It is always possible for the user to supply the allocations manually to overcome this problem, and without modification of conforming code, by simply adding an appropriate allocation file to the project.

It is also possible for the problem to be solved by a little environment support. At worst, a pre-link phase can be used to determine which variables require automagical allocation (by detecting and analysing linker errors if necessary :-).

Another solution is for the compiler to generate the relevant information in a special file, which is processed before link time to generate an extra object module (possibly by generate source to be compiled by the compiler).

There are surely any number of other solutions, not the least of which is to pressure linker suppliers to upgrade their linkers.

While C++ should be aware of existing technology, it should not be completely constrained by limitations of obsolete technology. In this case, the facilities already exist for the majority of users, and in particular for users of PC computers. In most of the competitive markets, workstations and supercomputers included, there should be no problem convincing vendors -- whether the original equipment manufacturer or otherwise -- to provide linkers suitable for C++.

5) Initialisation --- References

93-0017, N0225: *Consistent Treatment of Tentative vs Complete Members* by Bill Gibbons. Our proposal completes the set of tentative(declaration) and complete(definition) allowed in classes.

93-0016, N0224, *Class Name Injection* by Bill Gibbons. Issues of name lookup in classes.

92-0056
Existing proposal for in-class initialisation of static members. I am unable to obtain a copy, so I haven't read it. Our proposal goes further, allowing complete in-class definition with automagical allocation, and allowing default initialisers for non-static members.

92-0125, N0202, *Trivial Default Constructors* by Tom Pennello. We adopt the notion of trivial constructors given by Tom.

93-0015,N0023, *Resolved issues of name lookup* by Scott Turner. Introduces the notion of region as distinct from scope. In-class declarations and definitions are syntax given in the region of a class declaration.

All references below to **93-0010, N0218,** *C++ Draft, pre-Portland mailing***.**

The list is not complete.

The proposal should quote the entire relevant text.

Changes to the WP to be detailed after general acceptance
of the proposal.

3.1/1) Declaration/definition of static data members in class scope.

3.1/2) One definition rule

3.3/2) const has internal linkage by default,
enumerator has internal linkage
static class member has external linkage
noninline class member has external linkage
inline class member has in ternal linkage, but must
have one definition

5.4/19) Cast away const (see also 7.16)

7.1.2/3) One definition rule

7.1.2/4) inline implies internal linkage, otherwise external linkage

7.1.6/1) It is legal to write to a non-romable const, but not to
a romable const

8.2.6/3) default arguments in classes: names are bound at end of the
class declaration like inline function member bodies (see 9.3.2)

8.4) Initialisers

9.4/1,3,5) initialisation order of static members is determined by
placement in file scope of the allocators.

9.4/9) Clearly this does not apply for default initialisers

12.6) Initialisation.

6) Acknowledgements.

Bjarne Stroustrup, for suggesting I work on this topic, for reading the drafts, and especially for following up on things.

Bill Gibbons, for working on the original paper, and showing that the scope of an in-class initialiser raises no more problems than an inline member function.

Andrew Koenig, for offering comments on some of the ideas.

Rajesh Waren, Executive Officer, Standards Australia IT/9/22, for encouragement and copies of the C standard and Rationale.

Ron Guilmette, for helping with related core issues and general language lawyering.

Scott Turner, and Tony Hansen, for feedback on the proposal.

Appendix A: Compiler Constants

See (0.7) for the definiton.

The notion of *compiler constant expression* is intended to replace that of *constant expression*, and the attribute *compiler constant* of a name is new terminology. Perhaps the definition of constant expression can be changed to agree with that of compiler constant expression, but in any case a special name for symbols evaluable at compile time may be useful.

Enumeration constants are intended to be compiler constants, but in the case

```
  enum {x=x;}
```

x is initialised to its own un-initialised value.This should not be allowed and can be prevented by requiring that enumerators be compiler constants. A different problem occurs when one translation unit contains the specification

```
  extern const int x = 10;
```

and a second translation unit contains

```
  extern const int n;
  int x[n]; // legal??
```

Here n is a constant expression because it is initialised (in the first translation unit) by a const expression, but this is clearly not intended. The problem goes away if compiler constant expressions are used. Similarly,

```
  extern const int n;
  int x[n]; // legal?
  const int n=10;
```

A similar problem occurs for classes:

```
  struct X {
    static const int n;
  };
  int x[X::n]; // legal?
  int X::n=10;
```

and this case has direct bearing on in-class initialisation, because we want to allow

```
  struct X {
    static const int n=10;
    int x[n];
  };
```

as if it had been written

```
  struct X {
    enum {n=10};
    int x[n]; // legal!
  };
```

The initialisation of addressable store associated with a compiler constant need not be performed at the same time as the compiler constant is evaluated, indeed it may be meaningless to specify this since the store does not exist at compile time.

The WP attempts to prevent inconsistent use of constants by requiring static variables be initialised to their compile time constant values or zero before any dynamic initialisation is performed. This implies than any attempt to access the store of a compiler constant will always occur after it has been initialised, and, since such objects are romable and immutable, the store cannot have been changed. (This fails to determine what happens when user defined types are allocated statically, however. Are they cleared bitwise to zero, or are the individual data members set to zero? What effect does this rule have on references?).

Now the following strangeness arises with initialisating static members.

```
  struct X {
    static const int n;
    static const int m;
  };
  const int X::m=n; // set to 0?
  const int X::n=m; // oh dear! Also 0
```

Now try this:

```
  struct Y {
    static const int n;
    static const int m;
  };
  const int Y::n=m; // legal, but n==1? Used before initialialised?
  const int Y::m=1;
```

According to the WP, if `m` is a constant expression, n is initialised to 1, otherwise n is initialised to 0, however it appears that `m` can be used before it is initialised. Of course, this is not the case, but the code is extremely fragile, since removing the '`const`' specifier or using a floating type for '`m`' for the initialiser will change the meaning.

```
  struct Y {
    static const int n;
    static int m;
  };
  const int Y::n=m;
  int Y::m=1;
```

This unexpected behaviour comes from the forward declaration of constants. The notion of compiler constant, however, precludes this behaviour, and makes compiler constants act more or less as if `#defined`. However, this lead to

```
  extern const int  n;
  const int m=n; // not a compiler constant
  const int n=1;
  const int p=n; // compiler constant
```

Here `m` is not a compiler constant even though n is, because n has not been previously initialised, in other words, the compiler has not yet seen its definition. However, consider

```
   struct Y {
    static const int n;
    static const int m;
  };
  const int Y::n=m;
  const int Y::m=1.0; // not a constant expression
```

The reason this issue is important is it relates directly to in-class initialisation of static variables, and in particular potential compiler constants. Consider

```
struct A {
  static const int n=m;
  int x[n];
  f() { enum {a=n,b=m};}
  enum {m=1};
  int y[n];
  g() { enum {c=n};}
};
```

It is clear that, if no rewriting of static members is done, only the definition of n and b is valid. If the static members are rewritten at the end of the class declaration region as for an inline function, however, only the array extents are invalid, because the enumeration is not rewritten. However, if there is no rewiting, then

```
const int n=1;
struct B {
  static int a = n;
  static const int n =2;
};
```

sets a to 1, whereas if there is rewriting, the example is illegal by the reconsideration rule.

Since compiler constants should work like enums, one can argue that static members should not be subject to rewriting. On the other hand, many static members may not be constants, and even those that are will usually be simple literal values in which case rewriting is not important. However, rewriting may have a more significant effect for the bulk of non-constant static members. Furthermore, programmers may expect that complex initialisers be rewritten as for inline functions.

However, with a no-rewriting rule, it is possible to organise a class consistently by putting compiler constants at the start of the class, and other static variables right at the end. In both cases, the order within such groups is critical anyhow. Forward referencing is not allowed for compiler constants, and it is usually not intended for dynamically initialised members (except in special cases where the default 0 value is used).