Document Title: Static Initialization Summary
Document Numbers: X3J16/91-0137, WG21/N0070
Author: Steven Kearns, Software Truth

Introduction

This paper discusses the initialization of objects with file scope (section 3.2 of the working draft), static class members (section 9.4), and objects with local scope declared static (section 7.1.1). It then makes a number of proposals relating to this issue.

Current Status

What the working draft as of 11/9/91 has to say about initialization is summarized here. Each statement is labeled for future reference.

(Statement S1) Objects with file scope must be initialized before the first use of any function or object defined in that translation unit (section 3.4), and may take place before main is entered, though this is not required.

(Statement S2) Objects with storage class static that are not initialized and do not have a constructor are guaranteed to start off as 0 converted to the appropriate type. If the object is a class or struct, its data members start off as 0 converted to the appropriate type (section 8.4).

(Statement S3) Aggregates are initialized as described in (section 8.4.1); besides an array, an aggregate can be an object of a class with no constructors, no private or protected members, no base classes, no reference members, and no virtual functions. The order of initializations of individual array elements or members is undefined.

(Statement S4) Static members of a global class are initialized exactly like global objects and only in file scope (section 9.4).

(Statement S5) Objects with local scope declared static are initialized the first time control passes through its declaration (only). Where a static variable is initialized with an expresion that is not a constant-expression, default initialization to 0 of the appropriate type happens before its block is first entered (section 6.7).

(Statement S6) Constructors for nonlocal static objects are called in the order they occur in a file; destructors are called in reverse order (section 12.6.1).

Current practice i lize translation units one at a time, in a random order.

Initialization in C i. simp because objects with file scope can only be initialize h a c t expression, and be no objects are of class t nd th functions or construc ill be called during initi ion

---

The Problems

This section describes a number of probl vith the current of the C++ rules governing initialization.

(Problem P1) Statement S1 can be impossible to fulfill, as in the following example:

```
(Example E1)

file f.cpp:
    extern int y;
    int x = y;
    int foo() { return 18; }
    extern int b;
    int a = b+1;

file g.cpp:
    int foo();
    int y = foo();
    extern int a;
    int b = a+1;
```

The problem is that g.cpp must be initialized before f.cpp, since f.cpp's initialization uses object y in g. However, f.cpp must be initialized before g.cpp because g.cpp's initialization uses function f in f.cpp. It is impossible to satisfy both requirements.

(Problem P2) Even if Statement S1 was ammended to handle cases of Problem P1, implementing Statement S1 appears to be a problem.

Using DYNAMIC ANALYSIS means having the program detect the first use of a function or variable in a translation unit by checking every use as the program executes. Unfortunately, this method introduces runtime overhead. The overhead must be paid even after initialization. The DYNAMIC ANALYSIS section below discusses this approach.

Another idea is to have the environment do GLOBAL STATIC ANALYSIS to correctly order the initialization of the translation units. This would require the environment to simultaneously examine all of the translation units that make up the program, and trace through the initializations in order to deduce a correct ordering.
Unfortunately, requiring this method may place too great a burden on C++ vendors. Doing a less sophisticated global analysis makes it

easier for vendors, but less useful for users; the less sophisticated analysis will fail to find an ordering in cases where the more sophisticated analysis would succeed. The GLOBAL STATIC ANALYSIS section below discusses this idea in depth.

Allowing EXPLICIT PROGRAMMER CONTROL is another option. The idea is to allow the programmer to say "this translation unit must be initialized after these translation units". Then the environment orders the initializations of translation units to satisfy these requests, or reports an error if an impossible ordering is requested. This approach is the most powerful. However, it might invite errors since it requires the programmer to do the global analysis of initialization. The NAME/AFTER section describes this solution in detail, below.

(Problem P3) The C++ standard insists that every object be initialized, even if it is not obviosly used in the program (section ????????). The reason for this is that the object's constructor or initializer may have side effects which affect other parts of the program. Much existing code depends on this assumption, as well as the additional assumption that this initialization occurs before main(). Unfortunately, this contradicts statement S1, which says that a translation unit does not have to be initialized until an object or function in it is used. It also poses problems for programs that use dynamically loaded libraries. We would not want to require that every translation unit of a dll is initialized before main(), especially if most of these translation units will never be used. This latter topic will be discussed in depth in a paper by John Wilkinson.

(Problem P4) Statment S2 talks about an object of class type that doesn't have a constructor. However, the compiler will always generate a constructor for a class without a programmer-defined constructor (section ??????).

(Problem P5) The following initializations make sense only if f.cpp is initialized before g.cpp, but not the opposite. However, C++ offers no way to ensure that one translation unit is initialized before another.

```
(Example E2)

file f.cpp:
    int f();
    int x = f();

file g.cpp:
    extern int x;
    int y = x + 1;
    int f() { return 18; }
```

(Problem P6) There is an implicit assumption by C++ programmers that non-class variables without initializers or with a constant initializer, are initialized at the time a program is loaded. This implies that they are initialized "before initialization", which itself is a little strange. However, the standard does not explicitly say this. The problem is illustrated in the following example:

```
(Example E3)

file f.cpp:
    extern int x;
    int y = x++ + 2;
    int x = 12;
```

If all non-class variables start out at 0 and initialization goes from top to bottom, then y will have 2 and x will have 12, but the initialization to x is really a re-initialization or perhaps an assignment. If all non-class variables without initializers or with contant initializers are inited first and only once, then y will have 14 and x will have 13, but then the order of initialization does not go from top to bottom. If all non-class variables without initializers or with contant initializers are inited first and then reassigned when their definition is passed, then y will have 14 and x will have 12. If all non-class variables without initializers or with contant initializers are initially random and initialization proceeds from top to bottom, then y will be random and x will have 12.

It appears that many programs assume that all non-class variables without initializers or with contant initializers are initialized first and only once.

(Problem P7) In the initialization "int i = 5/0;", does the compiler report a compile time error or will an error occur at runtime?

(Problem P8) If a temporary is created during static initialization, when is the temporary destroyed?

```
(Example E4)

file f.cpp:
    class String { ...};
    char * s1 = String("hello ") + String("world!");
```

When are the temporary Strings generated for "hello" and "world" destroyed?

(Problem P9) Current practice is to initialize translation units in a random order. This can lead to serious difficulties. For example, consider an implementation of a list class with value semantics:

```
(Example E5)

file list.h:

    class listel;

    struct list {
        listel * thelistel;

        list();  // points thelistel to listel::nillistel
        ~list();
```

```
            ...
        };

        struct listel {
            int refc;    // reference count
            void * data;
            list next;

            static int list_els_allocated;
            static listel nillistel;

            listel();
            listel(int);  // used only to init nillistel
            ~listel();
            ...
        };

    file list.cpp:

        int listel::list_els_allocated = 0;

        // initing nillistel is very tricky because the list and
        // listel constructors assume nillistel is already constructed.
        // Therefore we use a special constructor especially designed
for
        // initing nillistel.
        listel listel::nillistel(1);

        list::list() { thelistel = &listel::nillistel; thelistel-
>refc++; }
        list::~list() { if (--thelistel->refc == 0) delete thelistel; }

        listel::listel()  {  data = 0; refc = 0; list_els_allocated++; }

        // The following is only used to init nillistel.
        // Setting the refc to 2 ensures that nillistel will never be
deleted,
        // even when nillistel is destructed at the end of the program.
        listel::listel(int) { refc = 2;  }


        listel::~listel()
        {
          // note that .next is destructed, which may cause other
listels
          // to be destructed.  Since nillistel.next points to
nillistel,
          // care must be taken that destructing nillistel does not
          // cause a repeat destruction of nillistel.
        }
        ...

    file string.h:
        class string {...};

    file string.cpp:
        #include "list.h"
        list mylist;
        ...
```

This list implemention uses two static class members: listel::list_els_allocated keeps track of the number of allocated listels, and listel::nillistel provides a listel that represents the nil list. The advantage of using nillistel instead of a 0 ptr to represent the nil list is that the list constructors and destructors don't have to check for a zero ptr.

It is crucial for the correct operation of the list class that nillistel be constructed before any other list is constructed, and that list_els_allocated is set to 0 before any listel is constructed. However, notice that string.cpp allocates a global list named mylist. With current implementations, it is unpredictable which translation unit will be initialized first; as a result, it is quite possible that this program will operate incorrectly. A similar problem was present in the iostream class: if you used printed to cout in a constructor for a global object before cout was initialized, BOOM!

The simplest solution is to avoid allocating nillistel statically. Instead, it is allocated dynamically the first time required:

```
(E  mple E5.5)

f:   list.h:

 ...
   struct listel {
    ....
      static listel * nillistel_p..;  //!!!!!' changed
      ...


   list


   ..
   list.  listel::nillistel_ptr         '!'      anged

 ''!!!!!! a  d
   stel * g   illistel(
    if (     ::nillis    tr) listel::nillist      = new
iste   ;
    return    el::nilli    tr;

 //!!!!!! c  ged
   list::list() { theliste_ = get_nillistel()          ->refc++;

      he followi  is only u    o init nilli
      etting the  c to 2 e     s that nilliste        ever be
deleted
      even when n  stel is     ucted at the e         program
    tel::listel   ) { refc     }
   ....
```

The   oblem with this solution is that it adds the time and sp   ead of a function call to every list construction. There e   gramming trick that can he   his problem. The   le shows how to rewrite list   p to alleviate   m:

```
   mple E6)    difying !        )

   list.h:
```

```
class listel;

struct list {
    listel * thelistel;

    list();  // points thelistel to *listel::nillistel_ptr
    ~list();
    ...
};

struct listel {
    int refc;   // reference count
    void * data;
    list next;


    static int list_els_allocated;
    // !!!!!! changed: from a listel to a listel *
    static listel * nillistel_ptr;
    // !!!!!! added: call it to initialize the list package
    static int initialize();

    listel();
    listel(int);  // used only to init nillistel
    ~listel();
    ...
};

// !!!!!!! added:  force a call from
//                 every translation unit that includes list.h
static int listhelper = listel::initialize();
```

file list.cpp:

```
int listel::list_els_allocated = 0;

// !!!!!!!!!!!!!! changed:
listel * listel::nillistel_ptr;

// initing nillistel is very tricky because the list and
// listel constructors assume nillistel is already constructed.
// Therefore we use a special constructor especially designed
for
// initing nillistel.
// !!!!!!!!!! added:
static int already_inited = FALSE;
int listel::initialize()
    {
      if (already_inited) return 1;
      nillistel_ptr = new listel(1);
      already_inited = TRUE;
      return 1;
    }
#define nillistel (*nillistel_ptr)

list::list() { thelistel = listel::nillistel_ptr; thelistel-
>refc++; }
list::~list() { if (--thelistel->refc == 0) delete thelistel; }

listel::listel()   {  data = 0; refc = 0; list_els_allocated++; }

// The following is only used to init nillistel.
// Setting the refc to 2 ensures that nillistel will never be
deleted,
```

```
        // even when nillistel is destructed at the end of the program.
        listel::listel(int) { refc = 2;  }


        listel::~listel()
        {
          // note that .next is destructed, which may cause other
listels
          // to be destructed.  Since nillistel.next points to
nillistel,
          // care must be taken that destructing nillistel does not
          // cause a repeat destruction of nillistel.
        }
        ...

    file string.h:
        class string {...};

    file string.cpp:
        #include "list.h"
        #include "string.h"
        list mylist;
        ...
```

Now, since any translation unit that uses "list" must include list.h, the "static int listhelper = listel::initialize();" will be included as well, which forces a call to the list initialization function before any list variable is constructed in that translation unit. Note, however, that we had to define nillistel as a pointer instead of statically allocating it. Also note that even this solution depends on the assumption that C++ initializes all global pointers and integers to 0 or to a constant expression in every translation unit before pursuing other initialization (Statement S2). Otherwise, already_inited might be initialized to FALSE after being set to TRUE, and nillist_ptr might be reset to 0 after being initialized.

Continuing with our example, note that string.h does not mention "list", but string.cpp uses a list to implement the string class. Look what happens if another file allocates a global string:

```
    (Example E7)   (Assuming Example E6)

    file myfile.cpp:
        #include "string.h"
        string Myname = "steve";
```

During initialization of myfile.cpp, the definition of Myname causes the string(char*) constructor to be called in the string.cpp translation unit. However, it is quite possible that the string.cpp translation unit has not been initialized yet, which means that mylist has yet to be initialized, and possibly the list package has yet to be initialized. This is very bad. One might try using the same trick that we used for list, on string:

```
    (Example E8)   (modifying Example E6)

    file string.h:
        class string {
            ...
```

```
            // !!!!! added:
            static int initialize();
        };
        // !!!!!! added:
        static int stringIniter = string::initialize();

    file string.cpp:
        #include "list.h"
        #include "string.h"
        // !!!!!! changed:  from list to list*
        list *mylist_ptr;
        // !!!!!! added:
        static int already_inited = FALSE;
        int string::initialize()
            {
              if (already_inited) return 1;
              mylist_ptr = new list();
              already_inited = TRUE;
              return 1;
            }
        #define mylist (*mylist_ptr)
        ...
```

Unfortunately, even this does not work! It ensures that string::initialize() is called before any global string is allocated, but it doesn't ensure that list::initialize() is called before calling "new list()". In fact, string.cpp may not even be aware of the need to initialize the list package. One solution is for string::initialize() to call list::initialize(). Another solution is to modify string.h to include list.h:

(Example E9)

```
    file string.h:
        // !!!!!! added
        #include "list.h"

        class string {
            ....
        };
        ...
```

This works because every translation unit that includes string.h will also include list.h, which means that list::initialize() will always be called before string::initialize(). What is very strange about the solution is that string.h includes list.h, even though "list" is never mentioned in declaring the string class. "list" is only used in implementing the string class.

In summary, it IS possible to make a class Y for which it is safe to define a variable of type Y at file scope. However, to do so, the implementor of Y must add overhead to every Y constructor, or use the initialization trick; cannot use global variables of class type; must avoid initializers, or restrict initializers to constant expressions; and must ensure that any other classes used in implementing Y are initialized before Y is. Yech.

(Problem P10) Neither Section 12.1 nor Section 12.8 describes the semantics of the compiler-generated default constructor.

The Unit of Initialization

Should initializations take place one translation unit at a time, or one variable at a time?

In a C program, all initializers are constant expressions, and cannot refer to each other. As a result, a C program behaves the same no matter how the environment orders initializations of variables. However, in C++ this is not true. In Example E10, y must be inited before x.

```
(Example E10)

file f.cpp:
    extern int y;
    int c = 3;
    int x = y;
    extern int b;
    int a = b+1;

file g.cpp:
    int foo();
    int y = c;
    extern int a;
    int b = a+1;
```

If all of f.cpp is initialized before all of g.cpp, then x will get a bad initialization because it uses y before y is inited. On the other hand, if all of g.cpp is inited before f.cpp, then y will get a bad initialization because c is uninited. So, if the unit of initialization is the translation unit, there is no good way to initialize f.cpp and g.cpp.

However, if you allow the unit of initialization to be a single definition then you can initialize variables in any order as long as a variable is initialized before it is used. So in Example E10, c would be initialized first, then y, then x, but b and a could not be initialized because each assumes the other is initialized first. This is similar to how a spreadsheet works, or a dataflow program.

The advantages of the smaller unit of initialization include
* can (theoretically) correctly initialize programs that a larger unit of initialization would not.

Disadvantages of the smaller unit of initialization include:
* May not be able to be implemented without runtime cost, a runtime cost you pay even after initialization.
* Adds non-determinism to the execution of initializations, which can be very confusing if initializations have side effects. This is illustrated below in Example E11.
* Unintuitive for people who expect initializations to proceed from top to bottom, one translation unit at a time.
* More information to keep track of to execute destructors in reverse order.

The advantages of a larger unit of initialization include:
* Probably simpler to implement.
* More predictable initializations when initializations have side effects.

The disadvantages of a larger unit of initialization include:
* unable to initialize some cases that the smaller unit succeeds on.
* less intuitive for people who expect initializations to proceed in a dataflow manner, like a spreadsheet.

Here is an example showing the introduction of non-determinism when initializations have side effects.

```
(Example E11)

file f.cpp:
    int x = 0;
    int y = ((x += 2), x);
    int z = ((x *= 3), x);

file g.cpp:
    extern int y;
    extern int z;
    int g = y;
    int h =z;
```

One legal ordering (where "legal" == "variables inited before used") would be init x, init y, init z, init g, init h. This would result in x being 6. Another possibility is init x, init z, init y, init g, init h; then x ends up with 2. Although Example E11 looks forced, it is just a simplified version of Example E8, in which string::initialize() has side effects of setting up certain variables to crucial values.

In addition, most people feel that it is natural to interpret the initialization as proceeding from the top of a file to the bottom (except that constant initializations take place before any other). If this discipline were followed then one could at least calculate what the end result would be in the case of side effects. Note that Examples E6 thru E8 assume this initialization discipline.

Interleaved Initialization

It seems natural to believe that one translation unit will be completely initialized before another. However, there have been proposals for implementing initialize-by-need (i.e. Dynamic Analysis) that violate this idea. Consider Example E12:

```
(Example E12)

file f.cpp:
```

```
    extern int d;
    int a = 1;
    int b = d-1;

file g.cpp:
    extern int c;
    int c = e;
    int d = c;

file h.cpp:
    int f = 3;
    int e = 4;
```

One proposal would init a, then start to init b but notice that b depends on d which is in a different translation unit. So g.cpp would be initialized, first initing c; however, c depends on e so h.cpp would be inited. After h.cpp was inited, g.cpp's initialization would finish, then f.cpp's initialization would finish. We call this "interleaved initialization" because even though initialization proceeds from top to bottom in a translation unit, initializations from different translation units are interleaved.

In any case, interleaved initialization appears to be a bad idea because it can cause unpredictable interactions due to side effects of initializations.

AFTER/NAME

The AFTER/NAME idea is a way to allow the programmer to specify the order of initialization of translation units.

First, each translation unit can be "named" by adding a declaration of the form "name <identifier>;" at file scope.

Then, each translation unit can include any number of declarations of the form "after <identifier>;". Such a declaration says that the module named <identifier> must be initialized before the translation unit containing the "after" declaration. It is the environment's job to order the translation units so that each "after" declaration is satisfied, or report an error if this is impossible.

Declarations of the form "! after <identifier>;" can also be included. This declaration cancels out any "after" declaration in the same translation unit with the same identifier. Its usefulness will be illustrated below.

Here is an example showing how to use "name" and "after":

```
(Example E13)

file string.cpp:
```

```
        #include "string.h"
        name string;
        ....

    file string.h:
        after string;
        ...

    file foo.cpp:
        #include "string.h"
        String s1 = "hi";
        ...

    file goo.cpp:
        #include "string.h"
        ...
```

In this example, both foo.cpp and goo.cpp will be initialized after string.cpp, because string.h includes an "after string;" declaration and both foo.cpp and goo.cpp includes string.h. However, the environment could initialize foo.cpp before goo.cpp, or vice versa.

The general rule that programmers can follow is:

(*) if foo.h file declares some functions or objects, then the foo.h file should include "after" declarations naming each translation unit containing the functions or objects.

This rule works because in order for a file goo.cpp to access objects or functions from another translation unit such as foo.cpp, it will normally include a header file such as foo.h; as a result, the proper "after" declaration will be included automatically.

Unfortunately, this rule of thumb breaks down in the case where two translation units call each other, and thus include each other's .h files:

```
    (Example E14)

    file string.cpp:
        name string;

    file string.h:
        after string;

    file foo.cpp:
        #include "string.h"
        #include "foo.h"
        #include "goo.h"
        name foo;
        int f() { cout << string("hi there"); }
        int f2() { return g2(); }

    file foo.h:
        after foo;
        int f();
        int f2();

    file goo.cpp:
        #include "foo.h"
```

```
        #include "goo.h"

        name goo;
        int a = f();
        int g2() { return 18; }
        ...

    file goo.h:
        after goo;
        extern int a;
        int g2();
        ...

    file hoo.cpp:
        #include "goo.h"
        int h = g2();
        ...
```

The problem is that the environment will report that goo must be initialized after foo which must be after goo which is impossible. In fact, for the initialization to proceed correctly, goo must be initialized after foo because the initialization of "a" in goo calls a function f() in foo, whereas no initialization in foo uses anything from goo. So the "after goo" declaration included in foo must be disabled. However, you cannot just delete the "after goo" from goo.h, because that in effect deletes it from hoo.cpp as well. The only recourse is to use the "!after goo" declaration within foo to cancel out the "after goo" declaration.

```
    (Example E15) (modifying Example E14)

    file foo.cpp:
        #include "string.h"
        #include "foo.h"
        #include "goo.h"
        name foo;
        ! after goo;
            // equivalent to stating that no initialization in foo, and
            // no function in foo that might be called during
            // initialization of any object in any translation unit,
            // uses any objects or functions from goo.
        int f() { cout << string("hi there"); }
        int f2() { return g2(); }
```

Summarizing the advantages of the AFTER/NAME proposal:

* it enables the programmer to easily specify the relative ordering of translation units.

* it can only fail to work if the programmer forgets to include an "after" declaration, or if the programmer incorrectly breaks a cylce with a "! after" declaration, or if the programmer uses a function or object from a translation unit without including the corresponding .h file.

Summarizing the disadvantages of the AFTER/NAME proposal:

* Extends the language with 2 new keywords and 3 new declarations, burdening the novice with more to learn and worry about.

* Could be error prone, especially compared to a method like global static analysis which can be done automatically by the environment.

GLOBAL STATIC ANALYSIS AT LINK TIME

Global Static Analysis is a technique that can be used by compiler vendors to automatically and correctly order the translation units for initialization. The basic idea is to analyze the program at link time or at the start of runtime, to figure out a correct order of initialization of translation units.

To do global static analysis of initialization at link time, a program must read in all of the translation units that make up a program. For illustration purposes call these units TA, TB, TC, and TD. Then, for each initialization in a translation unit the global analyzer must figure out which functions and objects from other translation units might be called. If an initialization in TA might use something from TB and TD, then both TB and TD must be initialized before TA. Here is an example:

```
(Example E16)

file TA:
    extern int f();
    int a = f();

file TB:
    extern int g();
    class b1 {};
    class c1 : public b1 {};
    c1 myc;
    int c = g();
    int d = 3;
    b1::b1() {}

file TC:
    class c1 : public b1 {...}
    c1::c1() { return; }
    int afunc() {}

file TD:
    extern int afunc();
    extern int d;
    int f() { return 3; }
    int g() { return afunc() + d; }
```

The global analyzer would calculate that the init of "a" in TA calls f() in TD, so it would note:

TA must be inited after TD

The init of "myc" in TB calls a constructor in TC which implicitly calls a constructor in TB. Therefore,

TB must be inited after TC
TC must be inited after TB

which illustrates an error (TB after TB) that the user must fix. The init of "c" in TB calls g() in TD, which calls afunc() in TC and uses d in TB; however, d is a non-class type inited with a constant expression so it is already initialized. Therefore,

TB must be inited after TD
TD must be inited after TC

Files TD and TC contain no initializations, so no further constraints are found. From the list of generated constraints the global analyzer would attempt to construct an ordering that satisfied the constraints. Since the constraints form a directed graph, this just reduces to a topological sorting of the directed graph, which always succeeds unless the graph contains a cycle, which is an error that should be reported.

There are a number of subtleties involved in this global analysis. One was illustrated in Example 16: analyzing a constructor involves understanding all the implicit functions that get called, just as analyzing a function call involves understanding all the implicit conversion functions that might be called, and just as analyzing an expression involves resolving overloaded operators.

Another subtlety involves calls to virtual functions. Consider:

```
(Example E17)

file TA:
    class Base { virtual int doit(); }
    int Base::doit() {return 1;}

file TB:
    class Base { virtual int doit(); }
    class Derived : public Base { virtual int doit(); }
    int Derived::doit() {return 1;}

file TC:
    class Base {...};
    Base * getBase();
    Base * obj = getBase();
    int c = obj->doit();
```

The problem is that getBase() might return a ptr to any type derived from Base, so the call to obj->doit() might invoke one of several functions. Therefore the global analyzer must assume that it could call either Derived::doit() in TB or Base::doit() in TA, and add appropriate constraints.

Another potential difficulty is analyzing function pointers and other pointers. Consider:

```
(Example E18)

file TA:
    extern long l;
    long * f() {return &l;}

file TB:
    extern long * f();
    extern int * ((*funcs)());
    long b0;
    long b = *f();
    long c = *((*funcs)());

file TC:
    long l = 3;

file TD:
    long * f();
    int * ((*funcs)()) = f;
```

In Example E18, the initialization of b involves the dereference of a pointer that might point to any global variable in the program. The initialization of c calls an unknown function, in an unknown translation unit. Actually, this may not be a problem at all: the initialization of b uses the result of f() in TA, and the global analyzer can tell that f() uses l in TC, so the initialization of b generates "TB after TA" and "TA after TC". Similarly, the initialization of c generates "TB after TD" and "TD after TA". I believe that such global analysis handles function pointers and variable pointers correctly, but more work is needed to prove it. (What about member function pointers?)

Another potential problem is libraries, for which one does not usually have the source code. For example, an initialization within the library might call a function libinit() that it expects you to write. It is not clear how to handle libraries.

Summarizing the advantages of Global Static Analysis at link time:

* gives the responsibility for correct initialization to the environment, which can presumably do it with much fewer errors and effort than a programmer.

Summarizing the disadvantages of Global Static Analysis at link time:

* May be difficult for compiler vendors to implement.
* Not clear how to handle libraries.
* May increase link time dramatically

GLOBAL STATIC ANALYSIS AT RUN TIME

Another possibility, suggested by Jerry Schwarz and others, is to do a less sophisticated form of globa' analysis at runtime, before the program starts. Instead of ana which functions and variables are accessed during initializat just assume that ANY global function or variable may be ca ring initialization. The advantage is that it is easy to ment with current compiler technology. The disadvantage is that the global analysis may be too unsophisticated to be generally useful. Here is Jerry's description of the implementation, with sor diting from me:

For each global variable or func. we invent an "initorder" function that can be called when that symbol is used in an initializer of a static.

E.g if we have

```
        // file foo.c
        class S { S() ; } ;
        extern int y ;
        extern void f(int) ;
        void g(int i) { f(i) ; }
        int x = g(y+1) ;
        S s ;
```

```
The initorder functions are something like

        initorder_g() {
            sti_foo();
        }

        initorder_x() {
            sti_foo() ; // initialize foo static variables
        }

        initorder_s() {
            sti_foo();
        }
```

The initializing function for foo has to do something like

```
        void sti_foo() {
            static int initialization_state = 0 ;
            if ( initialization_state != 0 ) return ;
            initialization_state = 1 ;

            // call all initorde  uncti
            initorder_f() ;
            initorder_y() ;
      initorder_S_constructor(),

      // do the initializations for this translation unit
            x=g(y+1) ;
            S_constructor(&s) ;
            initialization_sta   = 2 ;
        }
```

I assume that the initorder function for f only calls initorder functions for functions and variables that are explicitly named in it's bodies. This can miss a variety of dependencies due to indirection, virtual functions, function variables. (Actually, this may not be the case for indirection or function variables: before a function or variable can be accessed through a pointer, it has to be accessed explicitly. This may ensure that the initialization takes place in the correct order. However, this requires more thought to say for sure.)

If there is a circularity in the dependency I propose to say the order is undefined, rather than saying the program is illegal. It does seem important to notify the programmer if there is a circular dependency, however. The point of distinguishing initialization_state's 1 and 2 is to allow some diagnosis of circularity if we choose. (A slightly more elaborate version of the initorder functions might be required)

It is likely that many initorder functions will be identical to each other. Provision must be made to de-initialize the translation units in the reverse order of initialization. There is scope for compiler optimization. John Wilkinson suggests you can read off the call graph of the init_ and sti_ functions, do a topological sort, patch in the sti_ calls in the right order, and eliminate the init_ calls.

Advantages:

* Can be implemented by the compiler easily, by looking at only 1 translation unit at a time.
* Doesn't need more environmental support than current schemes

Disadvantages:

* More code must be generated
* Increased startup time, as much as (ngb*ntu*tfc), where
  ngb = (number of global variables and functions),
  ntu = (number of translation units),
  tfc = (time for a function call and integer check).
  This time is paid even if a program contains no global initialization.
* May or may not deal with virtuals, indirect function references, etc.
* It isn't clear how to explain in the RM exactly what is guaranteed. However, see Proposal 6D below.

A possible problem with this approach is that it is conservative; i.e. it will produce a random initialization in many cases where a correctly ordered initialization exists. The reason is that before initializing a module, this method initializes all functions and variables obviously accessed by the module, EVEN FUNCTIONS AND VARIABLES THAT MAY NOT BE CALLED DURING INITIALIZATION. This has to be done because when compiling a module there is no way to know which functions or variables in the module will be called as a result of an initialization outside the module.

Is this approach too conservative? Notice that if two translation units call each other, then this method will see a cycle, and it must initialize the translation units in a random order. This happens, for instance, if you implement a class using two translation units.

DYNAMIC ANALYSIS

Dynamic Analysis is a technique that detects for the first use of function or variable in a translation unit, at runtime, and initializes the translation unit before this first use.

This sounds great on paper, but in practice there seems to be a significant runtime cost to implement this idea.

Consider the problem of detecting the first call to a function in a translation unit FOO. This requires adding a function call to every global function in FOO, including inline functions:

```
f()
{
   /** compiler added the following statements **/
   FOO_initcheck();
   /** compiler added the previous statements **/

   .... real f() .....
}

FOO_initcheck()
{
   static int already_inited = FALSE;
   if (already_inited) return;
   .... init variables in FOO ....
}
```

Thus, every function expands in size, even inline functions. Also, every function call has time overhead, even after initialization, of calling another function, checking an integer, and returning.

Consider the problem of detecting the first use of a variable in translation unit FOO. This requires that every module different from FOO must check every use of an extern variable to see if the corresponding unit has been initialized. So if function g() in unit GOO uses an "extern int v":

```
g()
{
   /** compiler added the following statements **/
   if (! v_inited) v_INIT();
   /** compiler added the previous statements **/

   int x = v;
}
```

This expands the size of all functions that use extern variables by an "if" and a function call, it adds one integer flag for every global variable in the program, and it slows all functions by the number of extern variables it has times the time to check an integer flag and branch, even after initialization.

Summarizing the advantages of Dynamic Analysis:

* Cedes the responsibility for correct initialization to the environment, which can presumably do it with much fewer errors and effort than a programmer.
* It is the most powerful method of determining a correct initialization
  order, i.e. succeeds in all cases that other methods succeed, and fails in fewer cases.

Summarizing the disadvantages of Dynamic Analysis:

* Appears to be a runtime cost which is paid even after initialization is over.
* Libraries may be a problem.
* Leads to interleaved initialization of different translation units.

Proposals

(Proposal 1) Amend Statement S2 to say "objects of class type that don't have a constructor get initialized with the compiler generated default constructor". Or, do as Shopiro suggests: "address problem P4 by editorial changes. Classes which are 'brace initializable' are simple. There are default constructors that make a class not simple (e.g., the class has no constructor, but has a data member that has a constructor)."

This solves Problem P4.

(Proposal 2) Get the Core language working group to describe the semantics of the compiler-generated default constructor. We suggest that non-class members get initialized to 0 of the appropriate type, and that class members get intialized with the default constructor.

This solves Problem P10.

(Proposal 3) Add to section 3.4 something like the following:
"Define a 'simple object' as an object of non-class type defined without an initializer, or an object of non-class type with a constant initializer, or an object that can be brace initialized. Then all simple objects at file scope in all translation units get initialized before any non-simple object at file scope in any translation unit. "

This solves Problem P6.

(Proposal 4) Do nothing about Problem P7.

This [illegible]ves Pr[illegible] [illegible]y leaving it up to t[illegible] [illegible]iler writer what [illegible] [illegible] in [illegible] case

[illegible]ro[illegible] 5) [illegible]llowing to section 1[illegible] [illegible]rary Objects: Temporaries [illegible]ced at file scope durin[illegible] [illegible] as if all the definitions at file scope in a translatio[illegible] [illegible] a compound statement, with the order of definitions [illegible] order as in the file. In particular, all [illegible]oraries a[illegible] destroyed before initialization of the translatio[illegible] [illegible]t is comp[illegible]ed."

This solves Problem P8.

(Proposal 6) One of the Proposals 6A through 6E should be adopted.

(Proposal 6A) Change Statement S1 in Section 3.4 to the following:
"Define a 'simple object' as an object of non-class type defined without an initializer, or an object of non-class type with a constant initializer, or an object that can be brace initialized. Then all simple objects at file scope in all translation units get initialized before any non-simple object at file scope in any translation unit.

Translation units are initialized one-at-a-time [illegible] implementation defined (possibly random) order. [illegible]in a translation unit, non-simple objects at file scope are initialized [illegible] the order of their appearance in the translation unit."

Possible Ammendment, suggested by Shopiro: static objects in a translation unit are initialized before the first use of any object or function in that translation unit "on a thread from main." This allows the possibility of delaying the initialization of a module until after main() begins to execute.

This solves Problem P1 by removing Statement S1; solves Problem P2 by specifying a requirement which is in current practice; IGNORES Problem P5; solves Problem P6; and IGNORES Problem P9.

(Proposal 6B) Change Statement S1 in Section 3.4 to the following:
"Define a 'simple object' as an object of non-class type defined without an initializer, or an object of non-class type with a constant initializer, or an object that can be brace initialized. Then all simple objects at file scope in all translation units get initialized before any non-simple object at file scope in any translation unit.

Translation units are initialized one-at-a-time, in an order defined by the programmer. Implementations must provide a way to order the initialization of translation units, but the specific way this is specified is implementation defined. If the programmer does not specify an order, than the environment chooses an arbitrary ordering. Within a translation unit, non-simple objects at file scope are initialized in the order of their appearance in the translation unit."

This solves Problem P1 by removing Statement S1; solves Problem P2 by specifying a requirement which is the minimum requirement on vendors if ordering is allowed; solves Problem P5; solves Problem P6; and solves Problem P9. Actually, although Problems P5 and P9 are solved by Proposal 6b, two new problems appear for the programmer: (1) correctly deducing the right order of initialization, and (2) having to recode the order of initialization whenever the program is ported to a new compiler.

(Proposal 6C) Add a new language feature:
"At most one declaration of the form 'module <identifier>;' may appear at file scope within a translation unit. The <identifier> names the translation unit for later reference by a matching 'after <identifier>;' declaration. <identifier> is in a separate namespace used to name modules. "
"Any number of declarations of the form 'after <identifier>;' may appear at file scope within a translation unit. The declaration ensures that the current translation unit is initialized after the translation unit named with a matching 'module <identifier>;' declaration. "
"Any number of declarations of the form '! after <identifier>;' may appear at file scope within a translation unit. The declaration cancels out any and all 'after <identifier>' with matching <identifier>."

Change Statement S1 in Section 3.4 to the following:
"Define a 'simple object' as an object of non-class type defined without an initializer, or an object of non-class type with a constant initializer, or an object that can be brace initialized. Then all simple objects at file scope in all translation units get initialized before any non-simple object at file scope in any translation unit.
Translation units are initialized one-at-a-time, in any order that respects the "after <identifier>;" declarations within the translation units. The environment must generate an error if no correct ordering is possible, or if two translation units are named with the same identifier. Within a translation unit, non-simple objects at file scope are initialized in the order of their appearance in the translation unit."

This solves Problem P1 by removing Statement S1; solves Problem P2 by requiring the user to specify constraints between individual translation units and by requiring the environment to produce a compatible ordering; solves Problem P5; solves Problem P6; and solves Problem P9.

(Proposal 6D) Change Statement S1 in Section 3.4 to the following:
"Define a 'simple object' as an object of non-class type defined without an initializer, or an object of non-class type with a constant initializer, or an object that can be brace initialized. Then all simple objects at file scope in all translation units get initialized before any non-simple object at file scope in any translation unit.
Translation units are initialized one-at-a-time, in any order that ensures that no object or function is used before its containing

translation unit is initialized, as far as this can be deduced with reasonable effort before run-time. If a correct ordering among two modules cannot be deduced, the environment must generate a warning and will initialize the modules in a random order. Within a translation unit, non-simple objects at file scope are initialized in the order of their appearance in the translation unit."

This solves Problem P1 by removing Statement S1; limits Problem P2 by requiring the environment to use global static analysis at link time or run time to order the translation units; IGNORES Problem P5; solves Problem P6; and solves Problem P9.

(Proposal 6E) Change Statement S1 in Section 3.4 to the following:
"Define a 'simple object' as an object of non-class type defined without an initializer, or an object of non-class type with a constant initializer, or an object that can be brace-initialized. Then all simple objects at file scope in all translation units get initialized before any non-simple object at file scope in any translation unit.
Initialization of non-simple objects at file scope may proceed in any order such that each translation unit is completely initialized before any object or function defined within is used, and such that each object is initialized after all others that appear before it in the same translation unit. The environment must generate an error at runtime if it cannot produce a correct ordering.

This is the dynamic analysis solution. This solves Problem P1 by removing Statement S1; IGNORES Problem P2; IGNORES Problem P5; solves Problem P6; and solves Problem P9.

Recommendations

The Working Group recommends that Proposals 1 through 5 be accepted. The Working Group seems to be evenly split between proposals 6A (do nothing) and 6D (relaxed global static analysis). Note that relaxed global static analysis can be considered one way that an implementation can define initialization to take place; thus, you can argue that 6A subsumes 6D. On the other hand, the worst case of relaxed global static analysis is just a random initialization of translation units; thus, you can argue that 6D subsumes 6A.

My own personal belief is that 6D is the best solution. We have given a straightforward but simple implementation, and using John Wilkinson's optimization the overhead of this technique can be reduced to 0. The C philosophy is to avoid any unnecessary overhead, so we should not force the runtime overhead on any program that doesn't want it. However, I would always choose to use 6D for my programs, and Wilkinson's optimization suggests that all overhead can be eliminated.