why no examples?
how do do iostreams?

ANSI Doc Number:	X3J16/92-0108
ISO Doc Number:	WG21/N0185
Date:	October 3,1992
Project:	Programming Language C++
Reply To:	Philip Price
	Computer Sciences Corp.
	301 North R Street
	Lompoc, CA 93436
Telephone:	(805) 736-1275
FAX:	(805) 735-3237
EMAIL:	uunet!csclompc!phil
	phil@nid.csc.com
	phil@csc.com

Intertranslation Unit Static Initialization

Philip Price

1. Abstract

Companies using the C++ language have had to use various means to direct the compilers to initialize static objects in a specific order. These techniques are necessary when a static object is used, however indirectly, by a constructor of a static object that is in a different translation unit. It is preferred that a standard means be defined to replace the more limiting non-standard techniques currently used. The **depend_on** keyword is proposed as a solution to this problem.

2. Introduction

The current working paper (X3J16/92-0091 WG21/N0168) describes the intertranslation unit static initialization sequence in paragraph 3.4.5. There are two ways to interpret the second sentence in the paragraph. The first interpretation is that the 'or' in the sentence means either condition must be met. The other interpretation is that the 'or deferred' in the second sentence means 'and no later than'. The rest of the paper will assume the first interpretation in describing the current problem. If the second interpretation is instead assumed it would only reduce, not eliminate, the number of problems that would need to be resolved. The current problem can be demonstrated with a code fragment.

```
// Name Y.H - Interface file for the Y class.
class Y{
public:
  static int lastId; // Next numerically ascending Id that is
                     // available.
  const int myId;    // A unique id?
  Y(void);
};

// Name Y.C - Definition file of Y class and data members.
#include "Y.H"
static const int a=1; // Added so that Y::lastid uses a
                      // non -const initializer.
int Y::lastId=a-1;

Y::Y(void
):myId(lastId++){
}
//...

// Name: F.C - An arbitrary translation unit.
#include "Y.H"
static Y a;

// Name G.C - An arbitrary translation unit.
#include "Y.H"
static Y b;
//...
```

What is the value of a.myId in F.C? or for that matter Y::lastId? Expand this to multiple translation units (F.C and G.C) all with a static Y instance defined and the problem becomes more apparent. Possible values for b.myId are 0 ,1 or garbage. Also note that a.myId will sometimes have the same value as b.myId depending on the order of initialization of the translation units.

3. Proposal

The proposal is to add a declaration specifier and modify the working paper to reflect the addition of the keyword. The proposed keyword has been independently defined by several individuals and through discussions at past ANSI X3J16 meetings (ANSI X3J16/91-0137 WG21/N0070, ANSI X3J16/92-0083 WG21/N0160). The keyword as presented is a synthesis of those ideas.

decl-specifier:
  depend_on

The use of the keyword in a translation unit declares that the current translation unit shall perform initialization of static objects after the translation unit that contains the definition of the identifier. The identifier should have external linkage. Add the following to paragraph 3.4.5 in the current working paper:

> *A translation unit containing a declaration with the specifier depend_on will initialize after the translation unit that defines the declarator. Translation units that are not depended upon will initialize after translation units that are depended upon.*

The depend_on keyword only specifies the order of initialization in relation to the current translation unit.

A few examples will help demonstrate how the keyword solves the intertranslation unit static initialization problem. Each example contains file fragments. Short names are used to help reduce extraneous and irrelevant information.

Example 1: Programmers Usage. A programmer wishes to use a class called Y which requires a particular translation unit to be initialized before any static instance of Y is created so that a unique ID is given to each instance of class Y. This is Problem 1 with the Depend_on keyword added.

```
// Name Y.H - Interface file for the Y class.
class Y{
public:
  static int lastId; // Next numerically ascending Id that is
                     // available.
  const int myId;    // A unique id?
  depend_on Y(void);
};


// Name Y.C - Definition file of Y class and data members.
#include "Y.H"

static const int a=1; // Added so that Y::lastid uses a
                      // non -const initializer.
int Y::lastId=a-1;

Y::Y(void
):myId(lastId++){
}
//...

// Name: F.C - An arbitrary translation unit.
#include "Y.H"
static Y a;

// Name G.C - An arbitrary translation unit.
#include "Y.H"
static Y b;
//...
```

By including Y.H, a translation unit will initialize it's own static objects after the translation unit defining Y(void) constructor. The author of the Y class will have written the Y.H file and would only have added the depend_on decl specifier to the Y::Y(void) declaration if the translation unit defining the Y(void) constructor has static objects who must be initialize before constructing a static instance of Y. The author of the Y class could have used any externally linked identifier whose definition is in the translation unit to be initialized first. In this example an author could have used a *depend_on Y::lastId* instead of *depend_on Y::Y(void)*, but either statement works.

The depend_on statement is effectively ignored when it refers to the translation unit which is being initialized. Notice the Y.C file has a static object of file scope (Y::lastId) that is used in the Y(void) constructor. This requires that the translation unit as defined by the Y.C file must be initialized before an instance of the Y class can be constructed.

The Y class is used statically in different translation units without additional coding. The translation unit encompassing F and G initialize after the translation unit containing the definition of the Y::Y(void) symbol. The order of initialization is the translation unit containing Y then F and G. The code fragments above do not specify whether F or G initializes first. In answer to the question posed in problem 1 the value of b.myId can now be 0 or 1. Also note that a.myId will not equal b.myId.

If there is a conflict such as two translation units who both say they are after the other, then undefined behavior will occur and a diagnostic message shall be generated.

Example 2: Absence of Depend_on keyword. The following is an example where the depend_on keyword is neither used nor needed. Adding the depend_on keyword to the language will not change existing code behavior. If a program relies on a particular compiler's default order of translation unit initialization (which has not been defined in the standard) and does not use the depend_on keyword, then the behavior of that program will remain unchanged and unportable. The following example does not need the depend_on keyword and would be portable.

```
// Name: X.H - Class H declaration.
#include "Y.H"
class X{
public:
//...
private:
  Y a;
};

// Name: Y.H - Class Y declaration.
class Y{
public:
  Y(void);
//...
private:
  int a;
};

// Name: X.C - Class X definition
#include "X.H"
//...

// Name: Y.C - Class Y definition
#include "Y.H"
//...
```

The absence of the definition of static objects in the translation units defining both classes causes the depend_on keyword not to be required.

Example 3: Use between an application programmer and a Library author. This example is similar to Example 2. The library author will have written the interface file Support.H, while the application author will just include it. The application author did not need to do anything to support the library author needs.

```
// Name: Support.H - Library header.
#include <Dictionary.H>
class Support{
// ...
  depend_on static Dictionary rootNode;
};

//Name: X.C - Application Programmer's own Class definitions
#include "X.H"
#include <Support.H>

static Support a;

X::X(void){
  a.DoSomeMethod();
}
```

Example 4: Use with a Dynamically Linked Library. Essentially when a depend_on keyword refers to a translation unit which resides in a Dynamically Linked Library it is the equivalent of requesting the dynamically linked library to be loaded (if not already loaded), and the translation unit referred to within the library to initialize before the translation unit (with the depend_on decl-specifier) is initialized.

```
// Name: Support.H - Library header.
#include <Dictionary.H>
class Support{
// ...
  depend_on static Dictionary rootNode;
};

//Name: X.C - Application Programmer's own Class definitions
#include "X.H"
#include <Support.H>
static Support a;

X::X(void){
  a.DoSomeMethod();
}
```

5. Alternatives

The depend_on as proposed in the paper has a set of problems it does not resolve, and it requires people to rewrite their interface files(.H's) to include all of the .Hs used in their implementation(.C's) if they wish to eliminate the chances of incorrect static initialization. Further discussion of this problem set and the alternatives will be discussed on the environment group reflector. Also depend_on is the current keyword name (it won the most votes in Boston and replaces the name 'after'), if anyone desires a different keyword name please post it on the reflector and before the deadline for the pre-Portland mailing I will poll the committee on the keyword name choices. I have analyzed a number of other alternatives (before and after the Boston meeting) which are not mentioned below, but they have problems with either strong or weak dependencies, ambiguous resolution when dealing with virtual methods or required radical changes to the environment.

The first alternative is to do nothing. If selected then a plethora of solutions will exist as each institution strives to mitigate the problem in intertranslation unit static initialization. Doing nothing is not a viable solution.

Alternately flow analysis could be added to the language specification. In this case the specification would have to describe an algorithm to be adhered to. A particular behavior as describe in an algorithm is necessary if portability is to be achieved. No algorithm will solve all cases since the set of all cases would be the set of all directed graphs. Algorithms can only solve a subset of these cases. One example of the difficulties in this alternative is with shared memory systems. In a shared memory environment the route through code a particular instantiation of a static object takes could be affected by a shared object whose value is indeterminable until actual execution of a particular statement.

A ~depend_on keyword could be created (see ANSI X3J16/91-0137 WG21/N0070) which cancels a previous depend_on keyword. This would reduce the number of hard dependencies unnecessarily generated. I will describe this problem set in more detail on the reflector since I believe that this addition will be necessary.

The granularity could be increased and the depend_on keyword meaning modified. The keyword would be defined to apply only to data objects and would cause extra code to be generated (read as a runtime hit) to check if the depend_on object has already been initialized. If it had not already been initialized it's initialization code would be run. This option resolves the largest set of initialization problems, but will cause a slight performance decrease whenever a data object which explicitly has a depend_on decl-specifier is used or referenced. It would still have ambiguity problems.

6. Summary

As demonstrated by the above examples the depend_on keyword will resolve the primary intertranslation unit static initialization problem. It will not break existing code, nor will it put significant burdens on the application author. Only authors who need to control the order of initialization will be affected, and it will be a positive effect.

```
template<class T>
class List {
    ... static T t;
}
Listinit li;                         List<String> Ls;

-----------------

class String {
    ...
}
Stringinit si;
```