

Source Code Serialization and Use Scopes in *e*

Revision 0.6
10/3/2005

written by Matan Vax
send comments to matan@verisity.com

Table of Contents

1.	INTRODUCTION	2
1.1.	THE ORDERING PROBLEM IN E.....	2
1.2.	THIS DOCUMENT	3
2.	WITHIN A SINGLE MODULE.....	4
3.	IMPORTING AND DEPENDENCY.....	6
4.	CONCRETE LOAD ORDER	8
5.	VISIBILITY SCOPE OF PREPROCESSOR DIRECTIVES	10
5.1.	OVERVIEW.....	10
5.2.	CASES WHERE ORDER DIFFERS.....	10
5.3.	EXAMPLES.....	11

1. Introduction

1.1. *The Ordering Problem in e*

From a structural point of view a program consists of definitions of named entities (types, fields, methods, etc.) in terms of others. Since a program is analyzed in a serial manner there is the question where one can refer to a named entity relative to where it is introduced. Given that entities must have a name that is unique in their kind irrespective of the order of analysis, this question concerns only semantic correctness, that is, whether the program is legal or not (it may not be the case with namespaces, where different resolutions of names are possible).

In *e* the way source code is serialized in its analysis has much farther reaching consequences. This is due mainly to its AO nature, by which the components of a system are described gradually. Since the definition of named entities can be spread between different locations in the source code, the order of analysis of the code doesn't only determine the semantic correctness of the code but also affects the behavior of the program. For example, where fields are added to a struct in different extensions the analysis order of the extensions determines how an object of the struct is packed or unpacked. An even more typical example is the way analysis order determines how method actually runs if it has different extensions in different files.

As a part of the AO modeling paradigm, files are being given an important role in the structure of a system. They are treated in *e* as modules (and thus source file and module are used interchangeably henceforth). This is unlike other languages such as C or Java in which the separation of code into different files either carries no significance at all, or else corresponds exactly to the distribution of code between classes. So in *e* the question of serialization becomes the question of ordering the source files or modules in the process of analysis.

Another peculiarity of *e*, on which the ordering question has similar bearing, is the ability to extend and modify the syntax language by the program with macros (*define as* and *define as computed*). Here too, the order may determine whether such modification applies in some context, and thus have effect on the correctness of the program, and in some cases even on its behavior.

Two other *e* features are directly connected to order semantics. One is the ability to forward reference an entity within the module in which it is introduced. This ability is extended to address cyclic dependencies between modules. The other is the *e* preprocessor-like sublanguage. The preprocessor rules determine (among other things) the load order, and are actually analyzed by a separate phase according to Gstyle serialization.

1.2. *This Document*

The aim of this document is to address the two questions: a) how the source code of a program is serialized, and b) how does this affect the semantics both of definition and use of named entities, macros, and preprocessor directives.

The definitions in this document try to capture the language rules that correspond to existing practice and code base. Chapter 2 is a first step in answering question *b* above, by considering the simplest case, in which a module with no *import* statements is loaded. Chapter 3 describes the abstract considerations of *import* statement semantics, and its effect on definition order and use scopes. Chapter 4 addresses question *a* by describing the concrete semantics of *import* statements.

A whole new complication is brought about by the question of visibility scopes of preprocessor rules (identifiers declared by *#define* statement, either empty – as compilation flags, or with replacement string – as constants). This issue must be addressed both in itself, and also since it has direct bearing on load order. The reason is that *import* statements may themselves be inside *#ifdef* scope. The issue of preprocessor directive visibility rules is treated in chapter 5. In this chapter two examples are given which illuminate the definitions also of previous chapters.

2. Within a Single Module

We start by considering the case in which a single module with no *import* statements is loaded. We have to consider both entities that are declared by the current module and entities that were declared by modules already loaded by previous *load* commands (and specifically core library entities). The reason for elaborating on some apparently trivial matters will become clear only in the next chapter.

2.1. Use Scope

The rules on the use of entities within a single module are more liberal than those of other languages and perhaps from what one might expect. Obviously, entities declared in modules already loaded can be referred to anywhere in the current module. However, also entity declared in the module itself can be used anywhere within that module. Here are some cases for example:

- A struct can have a field of a struct type declared further down in the source file.
- Constraint can be put on a field that is declared later.
- An enum item can be presupposed by the implementation of a method and be actually added to the enum type only later in the file by an extend statement.
- A *when subtype* can be declared on a field that will only be declared for that struct later in the file.

This permissive policy takes care of the problem of mutually dependent definitions. It releases the language from the need for forward declaration constructs, as are common in other languages. At the same time it imposes a relaxation algorithm in resolving the references of named entities. References of named entities are being resolved in a serial order, which is just the order of the code, but in as many iterations as needed (a second iteration may not be enough for the resolution of types, since new fields may be introduced under *when* constructs in any depth of nesting, and other *when* subtypes may depend in turn on them). This iterative resolution process guarantees that whenever there is a resolution it would be found.

A further rule is that ambiguities cannot arise from forward references. Obviously when two entities by the same names are declared in the same module, only the second one is reported as an error. More importantly, in the resolution of short name of *when* subtype references (such as ‘big packet’ as opposed to ‘big’size packet’), a declaration of a new field or addition of a new enum item cannot result in an ambiguity of code in previous lines of the same module.

Unlike named entities, which can be forward referenced anywhere in the same module, macros apply to code in the same module from the point of definition and onwards. They cannot be forward referenced, since they are purely syntactic rules; there is no entity they introduce to be referred to.

2.2. Definition Order

Some named entities in e are extensible in the sense that they can be declared and defined initially at one place in the program's source code, and then their definition can be extended in other places. Extensible entities in e are structs, methods, events and enum types. The initial definition and each of the extensions of such entities is given by some single linguistic construct (e.g. in the case of structs, the *struct* statement and the *extend* statement). We call the part of the definition that is given by a single construct a *layer* of the definition. More than one layer of the definition of the same entity may be located in the same module, and of course, layers may be located in different modules.

The order of layers in the definitions of an entity counts for more than just resolution of reference, so here the straightforward rule applies. A new layer that is added in the module currently loading to an entity that was declared previously comes after layers in the modules already loaded. The order of layers of the same entity in the currently loading module depends on the order of the constructs' appearance in the source file. Here, as opposed to the use scope of names, the declaration must appear before any extension, even in the same module. For example, one can forward-reference a struct-type in the declaration of a variable, but one cannot extend the struct or inherit it before the struct itself was declared.

Here are a few examples for the significance of order within definition layers:

- When a method is extended twice in the same module with the 'is also' modifier the extension that appears last in the source file would run last when the method is called.
- If a method declared by previous modules is extended with an 'is first' modifier, this code would run before any other, including any layers of that method defined by a super-type of that struct.
- When enum items are added to an enum type by two different statements (the first being a *type* statement and the second necessarily an *extend* statement) the values of the last items added take the subsequent integer values (unless, of course, they are explicitly given ones).

3. Importing and Dependency

Definitions in one module make use not only of entities defined within it or in e 's core library, but also of entities declared in other modules. The *import* statement is the way to declare that a module relies on the declarations of another. It guarantees that the imported module is loaded before the importing module (this will be refined later). Thus an *import* statement declares **direct dependency** between two modules, and **dependency** in general is the transitive closure of the direct dependency relation.

The dependency relation is not necessarily asymmetric. Sometimes the definitions in one module presuppose declaration of another and vice versa. In such a case, whichever way we should serialize the definitions in these modules, there would be a forward reference in the use of some named entity across the “module boundary”. Therefore, in cyclic dependencies the code is actually treated as if it were all located in a single module in the sense described above, namely being one single use scope of entities. In other words entities declared in any of the mutually dependent modules can be used anywhere within these source files. Therefore code in modules that depend on each other must be serialized so that no other module comes in between. So cyclic dependencies affect load order.

These considerations call for the introduction of a generalized concept – a **dependency unit**. Dependency units are either single modules or sets of mutually dependent modules (identified by *import* statements). One dependency unit depends on another when a module of that dependency unit imports a module of the other. Unlike dependency between modules, the dependency relation between dependency units is asymmetric, and can thus be sorted topologically. Rephrasing the semantics of *import* statement, we may say it guarantees that the imported module is loaded previously or in the same dependency unit as the importing module.

We can sum things up by stating three requirements on the load order of modules:

- I. If module a depends on module b but b does not depend on a , module b loads before a .
- II. Modules in a single dependency unit load in consecutive order. More precisely, when modules a and b depend on each other, and module c loads between them then also module a and c depend on each other.
- III. Whenever possible the order of import statements in the source code should be taken into consideration. Module a should load before module b if both are imported by module c and the *import* statement of a appears in the source code before the import statement of b . This is true only in case c is the first module that imports b and circular dependencies don't require otherwise.

The requirements above may be taken as the user-view definition of the load order determined by *import* statements. Any ordering of modules that satisfies requirements I and II is in principle a legitimate implementation of the semantics of *import* statement

from the user's point of view. The extent to which consideration III should play a role in the definition is open.

In particular, rules I and II do not fully determine the load order of modules within a dependency unit on the one hand, or the order between unrelated dependency units on the other. Consideration III may only reduce the indeterminacy but still leave room for different orderings.

Ideally, this should not be something the user needs to know. The correctness and behavior of a program that is well designed in terms of aspect orientation should not be affected by different sorting of its dependency unit, or by different ordering of modules within a dependency unit. Dependencies should be declared by *import* statements whenever they are presupposed to guarantee that the entities used are in scope. Moreover, methodologically a set of mutually dependent modules should not add more than one aspect to each named entity they extend. Still, the concrete ordering of modules in *e* should be specified if one wishes to guarantee semantic equivalence. We turn now to describe the algorithm that determines the concrete load order.

4. Concrete Load Order

The module that is explicitly mentioned in the *load* command (or equivalently a single compilation) is called the *root module* of that load. We call the set of modules that are loaded by a single *load* command a *load cluster*. These are all the modules upon which the root module depends, except those that are already loaded. The order by which the modules should be loaded during the execution of the *load* command is uniquely determined by the code in files of the load cluster of that load. Modules that are already loaded are ignored in this process since their source files may not even be available.

One can think of a directed graph, where the modules of some load cluster are nodes, and the import statements correspond to edges from the importing module to the imported. We call this the *import graph* for a given root module. In the following we use the terminology of T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms*, MIT Press, 1990. We also refer the reader to this book for a description and correctness proof of a SCC algorithm we presuppose below.

1. We allocate *discovery time* and *finish time* to each node in the import graph using a simple DFS. The algorithm starts from the root module, where the exploration order of modules adjacent to a given module corresponds to the order of the *import* statements in the source file (requirement III above).

2. We then find the strongly connected components (SCC henceforth) of the graph, which correspond to dependency units. Collapsing all nodes in a SCC to a single node and keeping all edges between SCCs, we get a DAG, which we can call the SCC-DAG.

Note: Load order according to any topological sort on the SCC-DAG will yield requirement I above, and of course any consequent order within each SCC is just requirement II.

3. We determine the load order of the whole load cluster using a DFS on the SCC-DAG. This DFS starts from the SCC of the root module and explores new SCCs in the following order: It starts from edges that originate from nodes with greatest *finish time* and proceed to edges originating from nodes in descending *finish time* order. The edges originating from the same node are traversed in an order corresponding again to the order of the *import* statements in the module. After allocating load time to all dependent SCCs we can go on to determine the order of the modules inside the current one, which will simply be by descending *finish time*.

Here are a few definitions that will serve to describe the algorithm in pseudo code:

IG is an import graph.

$V[IG]$ is the set of all modules in the import graph – the load cluster.

$root[IG]$ is the root module of the load cluster.

$Adj[u]$ is the set of all modules imported by module u .

$d[u]$ and $f[u]$ are respectively the discovery time and finish-time of vertex u calculated by the DFS on the import graph.

$SCC[u]$ is the strongly connected component to which vertex u belongs, as is calculated by the SCC algorithm.

We make use of array *color* (with the usual meaning) for each vertex and a global variable *time*.

Calculate-Load-Order(IG)

```
for each  $u \in V[IG]$  do
    color[u]  $\leftarrow$  WHITE
time  $\leftarrow$  0
Visit-SCC(SCC[root[IG]])
```

Visit-SCC(c)

```
color[c]  $\leftarrow$  GRAY
for each  $u \in c$  in descending  $f[u]$  order do
    for each  $v \in Adj[u]$  (in the import statement order) do
        if color[SCC[u]] = WHITE
            Visit-SCC(SCC[u])
for each  $u \in c$  in descending  $f[u]$  order do
    load-time[u]  $\leftarrow$  time  $\leftarrow$  time+1
color[c]  $\leftarrow$  BLACK
```

The resulting load order is given in the *load-time* array, where each module has a unique index in the load process. Thus we can determine the order: module u loads after module v iff $load-time[u] < load-time[v]$.

A few notes on the module load order determined by this algorithm:

First, if there are no circular dependencies, each SCC consists of just one module. The load order in this case is simply the ascending order of the finish-time, since the algorithm runs just like a simple DFS.

Second, a circular dependency breaks this intuitive ordering, since the nodes from one SCC to another are not explored in an order corresponding to that of the *import* statements of the corresponding source file. Specifically, adding an import statement at some file in a big design may affect the load order globally, and not just for modules dependent on it.

5. Visibility Scope of Preprocessor Directives

5.1. Overview

The *#define* statement in *e*, along with *#ifdef*, *#ifndef*, *#else* and *#undef*, is intended to be used in a way similar to that of C preprocessor. This makes the visibility scope of *#define* rules very different from “real” *e* entities – both named entities and macros.

So far we discussed the order that figures in the use of named entities and macros. We may simply call it the *load order* to distinguish it from the different conception of ordering that is involved in determining the visibility scope of *#define* statements. This order is given by treating the *import* statements just like C preprocessor treats the *#include* directive. A *#define* statement that appears before an *import* statement is visible by, or applies to, all the code in the imported file (unless that module was loaded previously), although in terms of load order the declaration of the *#defined* name actually comes after it. We may call this order for short the *include order*.

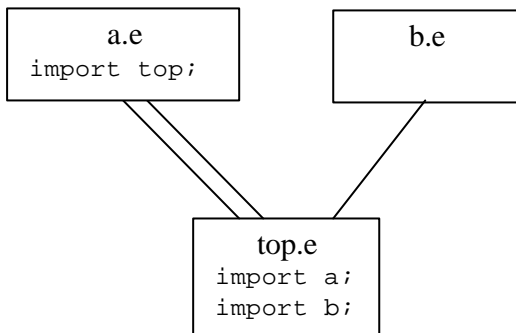
In general, the same source files are analyzed by two separate phases. The preprocessing phase is responsible for the discovery of the dependency relation. It figures out the load cluster and the order within it. It also executes the preprocessor directives, but it does that according to the *include order*, as the *load order* is not yet known. The second phase is the actual parsing of the full *e* code and its analysis. It imposes a serialization on the source code that may differ from the first. This discrepancy between the scopes of applicability of different statements in *e* has some unintuitive consequences that are demonstrated in the examples below.

5.2. Cases Where Order Differs

There are two patterns where the include order is different from the load order, and so the scope of application of *#define* statements is reversed:

Case 1

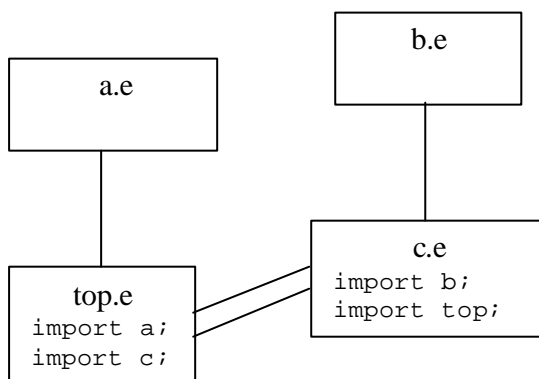
The simplest case occurs where cyclic import rule (requirement II in chapter 3 above) overrides the *import* statement order in the source file (requirement III):



Here module *b* must load before module *a* even though *a* is discovered first in the DFS. Definitions in *b* will be available in *a* if they obey load order but not if they obey include order, and vice versa.

Case 2

The effect of *e*'s concrete ordering (in chapter 4) which is not inherent in the abstract requirements:



Here module *a* was discovered before module *b* but loads after it. In this case too, definitions in *b* will be available in *a* if they obey load order but not if they obey include order. What is interesting about this example is that the two modules whose ordering reverses have no import statements (at least none relating to the cycle) and so have no trace of cyclic dependency.

5.3. Examples

The following two examples demonstrate the surprising consequence of the above definition.

Example 1

We use case 1 above to show how order of definition and use of the preprocessor versus proper *e* is reversed.

top.e

```
<'
import a;
import b;
'>
```

a.e

```
<'
import top;
#define A_SCANNED;

type t_a: t_b;    // 't_a' definition presupposes 't_b' definition
'>
```

b.e

```
<'
#ifdef A_SCANNED {

type t_b: int;    // 't_b' definition presupposes the C-like
                  // define 'A_SCANNED'
};
'>
```

In this test case, it may seem that whichever way you order the code in module *a* and *b*, module *a* should fail to load. But it does load, since the *#define* directive of *A_SCANNED* precedes the *#ifdef* statement according to *include order*, but the type declaration of *t_b* precedes its use in the declaration of *t_a* according to the *load order*. If module *a* would not import module *top*, the load order would fall back to the *#include* order and the code would not load.

Example 2

We use case 2 above to show how the definition of syntactic rules with *e* macros applies according to the load order, and can be made to stand in reverse order to a preprocessor C-like define statement.

a.e

```
<'
#define A_SCANNED;

sys_add_field foo;
'>
```

b.e

```
<'
#ifdef A_SCANNED {

define <sys_add_field'statement> "sys_add_field <name>" as {
    extend sys {
        <name>: int;
    };
};

};
'>
```

```
c.e  
<'<br>import b;<br>import top;<br>'>
```

```
top.e  
<'<br>import a;<br>import c;<br>'>
```

Again, this loads perfectly well. But if one gets rid of the cyclic dependency between module *top* and *c* (by commenting out the second line in 'c.e') the code fails to load, because the load order between module *a* and *b* reverses.