1 5 10 15 20 25 30 35 40 45 50 55

Annex C

(normative)

Source code serialization

This annex addresses two questions.

a) How is the source code of a program serialized?
b) How does this affect the semantics of the definition and use of named entities, macros, and preprocessor directives?

The definitions here capture the language rules that correspond to existing practice and code base.

— C.2 gives the first step in answering question *b* above, by considering the simplest case where a module with no **import** statements is loaded.
— C.3 describes the abstract considerations of **import** statement semantics and its effect on definition order and use scopes.
— C.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, which are identifiers declared by **#define** statements. These can be empty (as compilation flags) or have replacement strings (constants). This issue is addressed both in itself, and, also because it has direct bearing on the load order, since **import** statements can themselves be inside a **#ifdef** scope. The issue of preprocessor directive visibility rules is addressed in C.5.

x-ref 15.1 to this clause: "To understand how ***e*** code is serialized, see Clause C.";
x-ref clause 21 to sect C.3; x-ref clause 20 to sect C.5

C.1 The ordering problem in e

From a structural viewpoint, a program consists of definitions of named entities (types, fields, methods, etc.) in terms of other entities. Since a program is analyzed in a serial manner, there is the question of how to reference a named entity relative to where it is introduced. Given that all entities have a name that is unique in their kind irrespective of the order of analysis, this question concerns only semantic correctness, i.e., whether the program is legal or not (this might 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 mainly due to its analysis-ordered (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 not only determines the semantic correctness of the code, but it also affects the behavior of the program. For example, when fields are added to a *struct* in different extensions, the analysis order of the extensions determines how a struct object is packed or unpacked. An even more typical example is the way the analysis order determines how a method actually runs if it has different extensions in different files.

As a part of the AO modeling paradigm, files are given an important role in the structure of a system. They are treated in ***e*** as modules; thus, the terms source file and module are used interchangeably henceforth). This is unlike other languages, such as C or Java, where the separation of code into different files 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 how to order source files or modules in the process of analysis.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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. Here, too, the order can determine whether such modification applies in some context, and, thus, effect the correctness of the program or even 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 sub language. The preprocessor rules determine the load order and are actually analyzed by a separate phase according to C-style serialization.

C.2 Within a single module

Consider a case where a single module with no **import** statements is loaded. Both entities that are declared by the current module and entities that were declared by modules already loaded by previous **load** commands (i.e., core library entities) need to be considered here. This is further elaborated upon in C.3.

C.2.1 Use scope

The rules on the use of entities within a single module in ***e*** are more liberal than those of other languages. Obviously, entities declared in modules already loaded can be referred to anywhere in the current module. However, an entity declared in the module itself can be used anywhere within that module. For example:

— A *struct* can have a field of a struct type declared further down in the source file.
— Constraints can be put on a field that is not declared yet.
— An **enum** item can be presupposed by the implementation of a method and actually be added to the enum type later in the file (by an **extend** statement).
— A **when** subtype can be declared on a field that is actually 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 resolved in a serial order, which is just the order of the code, but in as many iterations as needed. (A second iteration might not be enough for the resolution of types, since new fields can be introduced under **when** constructs at any depth of nesting and other **when** subtypes might depend on them as well). This iterative resolution process guarantees that whenever there is a resolution, it shall be found.

A further rule is ambiguities shall not arise from forward references. Obviously, when two entities with 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 **when** subtype references (such as, 'big packet' as opposed to 'big'size packet'), a declaration of a new field or the addition of a new enum item shall not 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 onwards. Macros cannot be forward-referenced, since they are purely syntactic rules; there is no entity they introduce which can be referenced.

C.2.2 Definition order

Some named entities in ***e*** are extensible in the sense 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** and **extend** state-

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

ments). The part of the definition given by a single construct is called a *layer* of the definition. More than one layer of the definition of the same entity can be located in the same module and different layers can be located in different modules.

The order of layers in the definition 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, which at was declared previously, comes after any 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 shall appear before any extension, even within the same module. For example, a struct type can be forward-referenced in the declaration of a variable, but the *struct* cannot be extended or inherited before the *struct* itself is declared.

Here are a few examples of 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 shall run last when the method is called.
- If a method declared by previous modules is extended with the **is first** modifier, this code shall 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 (a **type** and an **extend** statement) the values of the last items added take the subsequent integer values (unless they are explicitly assigned).

C.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 declares that one module relies on the declarations of another module. This statement guarantees the imported module is loaded before the importing module (which is 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 module and vice-versa. In such cases, whichever way the definitions are serialized these modules, there can 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 need to be serialized so no other module comes in between them. So, cyclic dependencies affect load order.

These considerations call for the introduction of a generalized concept — a dependency unit. *Dependency units* are 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 dependency unit. Unlike dependency between modules, the dependency relation between dependency units is asymmetric and can be sorted topologically. Rephrasing the semantics of **import** statement — it guarantees the imported module is loaded previously or in the same dependency unit as the importing module.

There are three requirements on the load order of modules.

a) If module *a* depends on module *b,* but *b* does not depend on *a*, module *b* loads before module *a*.
b) 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 modules *a* and *c* also depend on each other.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

c) Whenever possible, the order of **import** statements in the source code needs to be taken into consideration. Module *a* should load before module *b* if both are imported by module *c* and the **import** statement of *c* appears in the source code before the import statement of *b*. This **is true only when?? module *c* is the first module that imports module *b* and circular dependencies don't require otherwise.

The requirements above can be taken as the user-view definition of the load order determined by **import** statements. Any ordering of modules that satisfies requirements #a and #b is, in principle, a legitimate implementation of the semantics of the **import** statement from the user's viewpoint. The extent to which consideration #c plays a role in the definition, however, remains open.

In particular, rules #a and #b 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 #c might reduce the indeterminacy, but still leave room for a different ordering.

Ideally, this is not something the user needs to know. The correctness and behavior of a program that is well designed in terms of aspect orientation is not be affected by the different sorting of its dependency unit or ordering of modules within a dependency unit. However, to guarantee any entities used are in scope, declare any dependencies by using **import** statements as necessary.

C.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. The set of modules that are loaded by a single **load** command is called 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 is loaded during the execution of the **load** command is uniquely determined by the code in the files of the load cluster of that load. Modules that are already loaded are ignored in this process, since their source files might not even be available.

Consider 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 one. This is called the *import graph* for a given root module. In the following, the terminology of T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms*, MIT Press, 1990 is used to **define the SCC algorithm??

Ref this and add it to the Bibliography

a) *Discovery time* and *finish time* are added 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 (consideration #c in C.3).
b) The strongly connected components (SCC) of the graph are found, which correspond to dependency units. Collapsing all nodes in a SCC to a single node and keeping all edges between SCCs leaves a **DAG??, which can be called the SCC-DAG.
   Load ordering according to any topological sort on the SCC-DAG yields requirement #a (in C.3) and any consequent order within each SCC is just requirement #b (in C.3).
c) The load order of the whole load cluster is determined by 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 the greatest *finish time* and proceeds to edges originating from nodes in a 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, the order of the modules inside the current one can then be determined, which is by descending *finish time*.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Now, the following definitions also 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.

Plus, an array *color* (with the usual meaning) is used for each vertex and a global variable *time*:

```
Calculate-Load-Order(IG)
    for each u ∈ V[IG] do
        color[u] ← WHITE
    time ← 0
    Visit-SCC(SCC[root[IG]])

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

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

NOTES

1—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.

2—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 might affect the load order globally and not just for modules dependent on it.

C.5 Visibility scope of preprocessor directives

This section describes the issue of preprocessor directive visibility rules.

C.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, only the order that figures in the use of named entities and macros has been discussed. This can be called 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

a 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 **#define**d name actually comes after it. This order is called 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 does so 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. This phase imposes a serialization on the source code that can differ from the first phase. This discrepancy between the scopes of applicability of different statements in *e* has some unintuitive consequences that are demonstrated in the following sections.

C.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 the **#define** statements is reversed.

C.5.2.1 Case 1

The simplest case occurs where the cyclic import rule (requirement #b (in C.3)) overrides the **import** statement order in the source file (requirement #c (in C.3)), as shown in Figure A1.

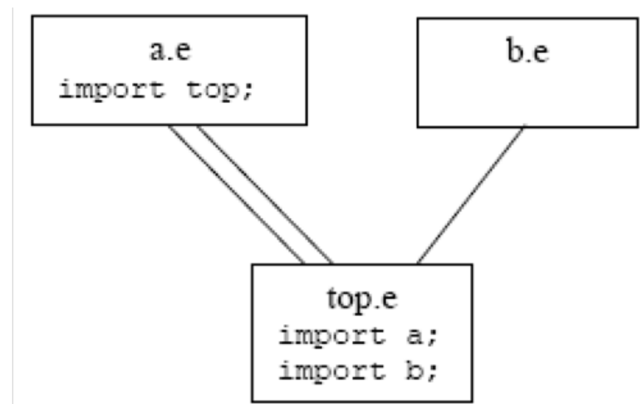

Figure A1—Overriding the import statement

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

C.5.2.2 Case 2

This case (see Figure A2) shows the effect of e’s concrete ordering (see C.4), which is not inherent in the abstract requirements.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

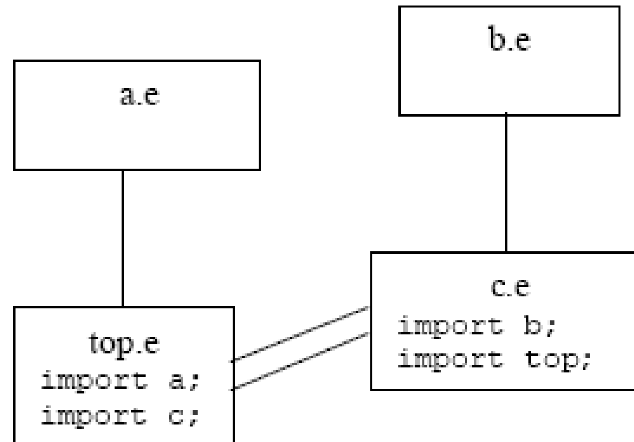

Figure A2—Ordering reversals without cyclic dependencies

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

C.5.3 Examples

The following examples demonstrate the surprising consequence of C.5.2.

Example 1

This uses Case 1 (see C.5.2.1) to show how the 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'
};
'>
```

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

In this test case, it might seem that whichever way the code in modules *a* and *b* are ordered, module *a* will fail to load. But it does load, since the **#define** directive of `A_SCANNED` precedes the **#ifdef** statement according to *include order*, but (**even though??) the type declaration of `t_b` precedes its use in the declaration of `t_a` according to the *load order*. If module *a* did not import module *top*, the load order falls back to the **#include** order and the code fails to load.

Example 2

This uses Case 2 (see C.5.2.2) to show how the definition of syntactic rules with ***e*** macros applies according to the load order and how this can be made to stand in reverse-order compared 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

```
<'
import b;
import top;
'>
```

top.e

```
<'
import a;
import c;
'>
```

This also loads perfectly well. But, if the cyclic dependency between modules *top* and *c* is removed (by commenting out the second line in `c.e`), the code fails to load because the load order between modules *a* and *b* reverses.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.