

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.9.2 External object definitions

Semantics

object external definition 1848
 If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier.

Commentary

external definition
 object reserve storage
 An external definition is an external declaration that is also a definition. The presence of an initializer turns a declaration into a definition, irrespective of its linkage. The definition of an object causes storage to be reserved for it.

C++

The C++ Standard does not define the term *external definition*. The object described above is simply called a *definition* in C++.

Other Languages

translation unit syntax
 The conditions under which a declaration is also a definition of an object vary between languages, depending on the model of separate translation they use.

Common Implementations

The base document did not support the use of initializers on declarations that included the **extern** storage-class specifier.

Coding Guidelines

Common usage is for developers to use the term *definition*, rather than *external definition*, to refer to such declarations. There does not appear to be a worthwhile benefit in attempting to change this common usage.

tentative definition 1849
 A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier **static**, constitutes a *tentative definition*.

Commentary

This defines the term *tentative definition* (which is only used in this, 6.9.2, subclause). Tentative definitions are a halfway house. They indicate that a declaration might be a definition, but the door is left open for a later declaration, in the same translation unit to be the actual definition or simply another tentative definition.

The concept of tentative definition was needed, in C90, because of existing code that contained what might otherwise be considered to be multiple definitions, in the same translation unit, of the same object. Defining and using this concept allowed existing code, containing these apparent multiple definitions of the same object, in the same translation unit (an external definition of the same in more than one translation unit is a constraint violation) to be conforming.

With two exceptions all external object declarations are tentative definitions; (1) a declaration that contains an initializer is a definition, and (2) a declaration that includes the storage-class specifier **extern** is not a definition.

C++

The C++ Standard does not define the term *tentative definition*. Neither does it define a term with a similar meaning. A file scope object declaration that does not include an explicit storage-class specifier is treated, in C++, as a definition, not a tentative definition.

A translation unit containing more than one tentative definition (in C terms) will cause a C++ translator to issue a diagnostic.

```
1 int glob;
2 int glob; /* does not change the conformance status of program */
3          // ill-formed program
```

definition
 one external

Coding Guidelines

The term *tentative definition* is not generally used by developers and tends only to be heard in technical discussion by translator writers and members of the C committee. There does not appear to be a worthwhile benefit in educating developers about this term and the associated concepts. Their current misconceptions (e.g., declarations that include the storage-class specifier **static** become definitions at the point of declaration) appear to be harmless.

Multiple *external-declaration*'s of the same object are redundant (this general issue is discussed elsewhere), but otherwise harmless (a modification of the type of one of them will result in a diagnostic being issued unless all of the declarations have compatible type, i.e., they are similarly modified).

redundant
code

-
- 1850 If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to 0.

object definition
implicit

Commentary

This specification requires implementations to create a definition of an object if the translation unit does not contain one (i.e., there is no declaration of the object that includes an initializer). For an object having an incomplete array type the effect of this specification is to complete the type and define an array having a single element. In the case of objects having an incomplete structure or union type the size of the object is needed to define it, which in turn requires a completed type. Thus, the behavior is undefined if an external object definition has an incomplete structure or union type at the end of a translation unit.

1848 object
external defini-
tion

1853 **EXAMPLE**
tentative array
definition
object
reserve storage
footnote
109

C++

The C++ Standard does not permit more than one definition in the same translation unit (3.2p1) and so does not need to specify this behavior.

Coding Guidelines

It is common practice to omit the initializer in the declaration of an object. Developers invariably assume that an object declaration that omits a storage-class specifier is a definition (which does eventually become). The fact that an object might not be defined at its point of declaration is purely a technicality.

-
- 1851 If the declaration of an identifier for an object is a tentative definition and has internal linkage, the declared type shall not be an incomplete type.

object
type for inter-
nal linkage

Commentary

This requirement applies at the point of declaration, not at the end of the translation unit. The affect of this difference is illustrated by the following example:

```
1 static char a[]; /* Internal linkage, undefined behavior. */
2 char b[]; /* External linkage, equivalent to char b[] = {0}; */
```

One consequence of this requirement is that implementations can allocate storage for objects having internal linkage as soon as the declaration has been processed, during translation.

In the case of objects having external linkage the behavior is not undefined if the object has an incomplete array type (see previous C sentence). For objects having no linkage it is a constraint violation if the type of the object is not completed by the end of the declarator.

object
type complete
by end

Common Implementations

Some implementations (e.g., gcc) support the declaration of objects having a tentative definition and internal linkage, using an incomplete type (which is completed later in the same translation unit).

EXAMPLE
linkage

EXAMPLE 1

```

int i1 = 1;           // definition, external linkage
static int i2 = 2;   // definition, internal linkage
extern int i3 = 3;   // definition, external linkage
int i4;              // tentative definition, external linkage
static int i5;       // tentative definition, internal linkage

int i1;              // valid tentative definition, refers to previous
int i2;              // 6.2.2 renders undefined, linkage disagreement
int i3;              // valid tentative definition, refers to previous
int i4;              // valid tentative definition, refers to previous
int i5;              // 6.2.2 renders undefined, linkage disagreement

extern int i1;       // refers to previous, whose linkage is external
extern int i2;       // refers to previous, whose linkage is internal
extern int i3;       // refers to previous, whose linkage is external
extern int i4;       // refers to previous, whose linkage is external
extern int i5;       // refers to previous, whose linkage is internal

```

Commentary

```

1  extern int i1;      // external linkage
2  extern int i2;      // external linkage
3  extern int i3;      // external linkage
4  extern int i4;      // external linkage
5  extern int i5;      // external linkage
6
7  int i1;             // external linkage
8
9  int i1 = 1;         // definition, external linkage
10 static int i2 = 2;  // internal linkage: undefined behavior -> external linkage on previous declaration
11 extern int i3 = 3;  // definition, external linkage
12 int i4;             // external linkage
13 static int i5;      // internal linkage: undefined behavior -> external linkage on previous declaration

```

C++

The tentative definitions are all definitions with external linkage in C++.

EXAMPLE
tentative array
definition

EXAMPLE 2 If at the end of the translation unit containing

```
int i[];
```

the array `i` still has incomplete type, the implicit initializer causes it to have one element, which is set to zero on program startup.

Commentary

The implicit initialization (equal to 0), at the end of a translation unit, of the tentative definition of an object describes an effect. In the case of an object declared to be an array of unknown size, the initializer is treated as specifying a single element.

C90

This example was added to the C90 Standard by the response to DR #011.

C++

This example is ill-formed C++.

object def-1850
inition
implicit

Coding Guidelines

This usage might be considered suspicious in that declaring an object to have an array type containing a single element is unusual and if it was known that only one element was needed why wasn't this information specified in the declaration. If the usage was unintended it is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults. Any intended usage is likely to be rare and thus a guideline recommendation (if shown to be cost effective technically) is not cost effective.

guidelines
not faults

References

1. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, May 2005.