

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.2.2 Linkages of identifiers

linkage

An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*.²¹⁾ 420

Commentary

This defines the term *linkage*. It was introduced into C90 by the Committee as a means of specifying a model of separate compilation for C programs.

It is intended that the different scopes mentioned apply to file scopes in both the same and different translation units (block scopes can also be involved in the case of tags). The need to handle multiple declarations, in the same and different translation units, of the same file scope objects was made necessary by the large body of existing source code that contained such declarations. Linkage is the mechanism that makes it possible to translate different parts of a program at different times, each referring to objects and functions defined elsewhere.

program
not translated
at same time

Rationale

The definition model to be used for objects with external linkage was a major C89 standardization issue. The basic problem was to decide which declarations of an object define storage for the object, and which merely reference an existing object. A related problem was whether multiple definitions of storage are allowed, or only one is acceptable. Pre-C89 implementations exhibit at least four different models, listed here in order of increasing restrictiveness:

Common Every object declaration with external linkage, regardless of whether the keyword `extern` appears in the declaration, creates a definition of storage. When all of the modules are combined together, each definition with the same name is located at the same address in memory. (The name is derived from common storage in Fortran.) This model was the intent of the original designer of C, Dennis Ritchie.

Relaxed Ref/Def The appearance of the keyword `extern` in a declaration, regardless of whether it is used inside or outside of the scope of a function, indicates a pure reference (`ref`), which does not define storage. Somewhere in all of the translation units, at least one definition (`def`) of the object must exist. An external definition is indicated by an object declaration in file scope containing no storage class indication. A reference without a corresponding definition is an error. Some implementations also will not generate a reference for items which are declared with the `extern` keyword but are never used in the code. The UNIX operating system C compiler and linker implement this model, which is recognized as a common extension to the C language (see §K.5.11). UNIX C programs which take advantage of this model are standard conforming in their environment, but are not maximally portable (not strictly conforming).

Strict Ref/Def This is the same as the relaxed `ref/def` model, save that only one definition is allowed. Again, some implementations may decide not to put out references to items that are not used. This is the model specified in K&R .

Initializer This model requires an explicit initialization to define storage. All other declarations are references.

Table 420.1: Comparison of identifier linkage models

Model	File 1	File 2
common	<code>extern int I; int main() { I = 1; second(); }</code>	<code>extern int I; void second() { third(I); }</code>
Relaxed Ref/Def	<code>int I; int main() { I = 1; second(); }</code>	<code>int I; void second() { third(I); }</code>
Strict Ref/Def	<code>int I; int main() { I = 1; second(); }</code>	<code>extern int I; void second() { third(I); }</code>
Initializer	<code>int I = 0; int main() { I = 1; second(); }</code>	<code>int I; void second() { third(I); }</code>

C++

The C++ Standard also defines the term *linkage*. However, it is much less relaxed about multiple declarations of the same identifier (3.3p4).

A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:

Other Languages

Few languages permit multiple declarations of the same identifier in the same scope. The mechanisms used by different languages to cause an identifier to refer to the same object, or function, in separately translated source files varies enormously. Some languages, such as Ada and Java, have a fully defined separate compilation mechanism. Fortran requires that shared objects be defined in an area known as a common block. (The semantics of this model was used by some early C translators.) Other languages do not specify a separate compilation mechanism and leave it up to the implementation to provide one.

translation unit
syntax

Fortran specifies the common block construct as the mechanism by which storage may be shared across translation units. Common blocks having the same name share the same storage. The declarations within a common block are effectively offsets into that storage. There is no requirement that identifiers within each common block have the same name as identifiers at the corresponding storage locations of matching common blocks.

```

1      SUBROUTINE A
2      COMMON /XXX/ IFRED
3      END
4
5      SUBROUTINE B
6      COMMON /XXX/ JIM
7 C    IFRED and JIM share the same storage
8      END

```

Common Implementations

Early implementations provided a variety of models for deducing which declarations referred to the same object, as described in the previous Rationale discussion.

Coding Guidelines

The term *linkage* is not generally used by developers. Terms such as *externally visible* (or just *external*) and *not externally visible* (or *not external, only visible within one translation unit*) are used by developers when discussing issues covered by the C term *linkage*. Is it worthwhile educating developers about linkage and how it applies to different entities? From the developers point of view, the most important property is whether an object can be referenced from more than one translation unit. The common usage terms *external* and *not external* effectively describe the two states that are of interest to developers. Unless a developer wants to become an expert on the C Standard, the cost/benefit of learning how to apply the technically correct terminology (*linkage*) is not worthwhile.

linkage
educating
developers

In many ways following the guideline recommendation dealing with having a single point of declaration for each identifier often removes the need for developers to think about linkage issues.

422.1 identifier
declared in one file

421 There are three kinds of linkage: external, internal, and none.

Commentary

The linkage *none* is sometimes referred to (in this standard and by developers) as *no linkage*.

C++

The C++ Standard defines the three kinds of linkage: external, internal, and no linkage. However, it also defines the concept of *language linkage*:

linkage
kinds of

All function types, function names, and variable names have a language linkage. [Note: Some of the properties associated with an entity with language linkage are specific to each implementation and are not described here.

7.5p1

For example, a particular language linkage may be associated with a particular form of representing names of objects and functions with external linkage, or with a particular calling convention, etc.] The default language linkage of all function types, function names, and variable names is C++ language linkage. Two function types with different language linkages are distinct types even if they are otherwise identical.

Other Languages

The term *linkage* is unique to C (and C++). The idea behind it— of making identifiers declared in one separately translated source file visible to other translated files— is defined in several different ways by other languages.

Coding Guidelines

The term *external* is well-known to developers; some are also aware of the term *internal*. However, the fact that they relate to a concept called *linkage*, and that there is a third *none*, is almost unknown. The term *visible* is also sometimes used with these terms (e.g., *externally visible*).

In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function.

422

Commentary

The phrase *particular identifier* means identifiers spelled with the same character sequence (i.e., ignoring any nonsignificant characters). Other wording in the standard requires that the types of the identifiers be compatible (otherwise the behavior is undefined). The process of making sure that particular identifiers with external linkage refer to the same object, or function, occurs in translation phase 8.

C++

The situation in C++ is complicated by its explicit support for linkage to identifiers whose definition occurs in other languages and its support for overloaded functions (which is based on a function’s signature (1.3.10) rather than its name). As the following references show, the C++ Standard does not appear to explicitly specify the same requirements as C.

3.2p3 *Every program shall contain exactly one definition of every non-inline function or object that is used in that program; no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 12.1, 12.4 and 12.8). An inline function shall be defined in every translation unit in which it is used.*

Some of the consequences of the C++ *one definition rule* are discussed elsewhere.

object
external linkage
denotes same
function
external linkage
denotes same

same object
have compat-
ible types

translation
phase
8

C++
one definition rule

3.5p2

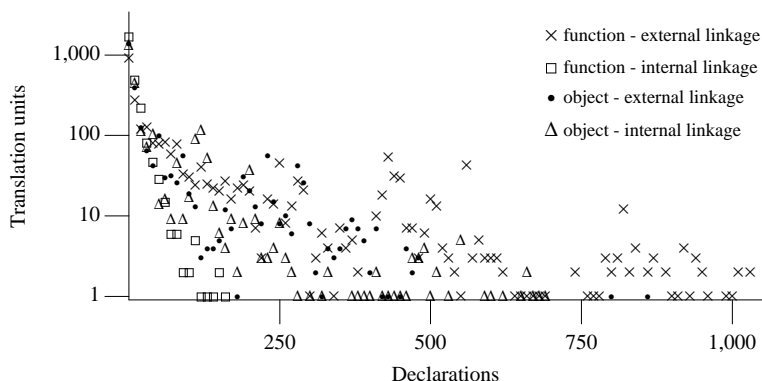


Figure 421.1: Number of translation units containing a given number of objects and functions declared with internal and external linkage (excluding declarations in system headers). Based on the translated form of this book’s benchmark programs.

A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:

— *When a name has external linkage, the entity it denotes can be referred to by names from scopes of other translation units or from other scopes of the same translation unit.*

At most one function with a particular name can have C language linkage.

7.5p6

Common Implementations

As a bare minimum, translators have to write information on the spelling of identifiers having external linkage to the object code file produced from each translation unit. Some translators contain options to write out information on other identifiers. This might be used, for instance, for symbolic debugging information.

Most linkers (the tool invariably used) simply match up identifiers (having external linkage) in different translated translation units with the same spelling (the spelling that was written to the object file by the previous phase, which will have removed nonsignificant characters). While type information is sometimes available (e.g., for symbolic debugging), most linkers ignore it. Very few do any cross translation unit type checking. The commonly seen behavior is to use the start address of the storage allocated (irrespective of whether an object or function is involved).

linkers

Issues, such as identifiers not having a definition in at least one of the translation units, or there being more than one definition of the same identifier, are discussed elsewhere.

external
linkage
exactly one
external definition

Coding Guidelines

There is no requirement for a translator to generate a diagnostic message when a particular identifier is declared using incompatible types in different translation units. The standard defines no behavior for such usage and accessing such identifiers results in undefined behavior. A translator is not required to, and most don't, issue any diagnostic messages as a result of requirements in translation phase 8.

same object
have compatible
types
translation phase
8

Because of the likely lack of tool support for detecting instances of incompatible declarations across translations units, these guidelines recommend the use of code-structuring techniques that make it impossible to declare the same identifier differently in different translation units. The lack of checking by translators, and other tools, then becomes a nonissue. The basic idea is to ensure that there is only ever one textual instance of the declaration of an identifier. This can never be different from itself, assuming any referenced macro names or typedefs are also the same. The issue then becomes how to organize the source to use this single textual declaration.

The simplest way of organizing source code to use single declarations is to place these declarations in header files. These header files can be included wherever their contents need to be referenced. A consequence of this guideline is that there are no visible external declarations in .c source files.

Cg 422.1

Only one of the source files translated to create a program image shall contain the textual declaration of a given identifier having file scope.

The C Standard requires all identifiers to be explicitly declared before they are referenced. However, this requirement is new in C99. The following guideline recommendation is intended to cover those cases where the translator used does not support the current standard, or where the translator is running in some C90 compatibility mode.

operator
()

Cg 422.2

A source file that references an identifier with external linkage shall **#include** the header file that contains its textual declaration.

header name
same as .c file
header name
external linkage
exactly one
external definition

A commonly used convention is for the .h file, containing the external declarations of identifiers defined in some .c file, to have the same header name as the .c file. The issue of ensuring that there is a unique definition for every referenced identifier is discussed elsewhere.

Example

The types of the object array in the following two files are incompatible.

```

_____ file_1.c _____
1 extern int glob;
2 extern char array[10]; /* An array of 10 objects of type char. */

_____ file_2.c _____
1 extern float glob;
2 extern char *array; /* A pointer to one or more objects of type char. */

```

Usage

A study of 29 Open Source programs by Srivastava, Hicks, Foster and Jenkins^[1] found 1,161 identifiers with external linkage, referenced in more than one translation unit, that were not declared in a header, and 809 instances where a header containing the declaration of a referenced identifier was not **#included** (i.e., the source file contained a textual external declaration of the identifier).

identifier
same if internal
linkage

Within one translation unit, each declaration of an identifier with *internal linkage* denotes the same object or function. 423

Commentary

This describes the behavior for the situation, described elsewhere, where there is more than one declaration of the same identifier with internal linkage in a translation unit. Other wording in the standard requires that the types of the identifiers be compatible

An object, defined with internal linkage in a translation unit, is distinct from any other object in other translation units, even if the definition of an identifier, denoting an object, in a different translation unit has exactly the same spelling (the linkage of the object in the other translation units is irrelevant).

Coding Guidelines

Why would a developer want to have more than one declaration of an identifier, with internal linkage, in one translation unit? There is one case where multiple declarations of the same object, with internal linkage, are necessary—when two or more object definitions recursively refer to each other (such usage is rare). There is also one case where multiple declarations of the same function, with internal linkage, are required—when two or more functions are in a recursive call chain. In this case some of the functions must be declared before they are defined. While it may be necessary to create type definitions that are mutually recursive, the declared identifiers have *none* linkage.

Some coding guideline documents recommend that all functions defined in a translation unit be declared at the beginning of the source file, along with all the other declarations of file scope identifiers.

The rationale given for a guideline recommendation that identifiers with external linkage have a single textual declaration included a worthwhile reduction in maintenance costs. This rationale does not apply to declarations of identifiers having internal linkage, because translators are required to diagnose any type incompatibilities in any duplicate declarations and duplicate declarations are not common.

Example

```

1 struct T {struct T *next;};
2
3 static struct T q;
4

```

function call
recursive
EXAMPLE
mutually refer-
ential structures

identifier 422.1
declared in one file

```

5  static struct T p = { &q }; /* Recursive reference. */
6  static struct T q = { &p }; /* Recursive reference. */
7
8  static void g(int);
9
10 static void f(int valu)
11 {
12     if (valu--)
13         g(valu);
14 }
15
16 static void g(int valu)
17 {
18     if (--valu)
19         f(valu);
20 }

```

424 Each declaration of an identifier with *no linkage* denotes a unique entity.

Commentary

Here the phrase *no linkage* is used to mean that the linkage is *none*. The term *no linkage* is commonly used to have this meaning. The only entities that can have a linkage other than linkage *none* are objects and functions, so identifiers denoting all other entities have *no linkage*.

Tags are the only identifiers having no linkage that may be declared more than once in the same scope. Other wording in the standard makes it a constraint violation to have more than one such identifier with the same name in the same scope and name space.

C++

The C++ *one definition rule* covers most cases:

No translation unit shall contain more than one definition of any variable, function, class type, enumeration type or template.

no linkage
identifier decla-
ration is unique

432 identifier
no linkage
incomplete
types
declaration
only one if no
linkage

3.2p1

However, there is an exception:

*In a given scope, a **typedef** specifier can be used to redefine the name of any type declared in that scope to refer to the type to which it already refers. [Example:*

```

typedef struct s { /* ... */ } s;
typedef int I;
typedef int I;
typedef I I;

```

—end example]

7.1.3p2

Source developed using a C++ translator may contain duplicate typedef names that will generate a constraint violation if processed by a C translator.

The following does not prohibit names from the same scope denoting the same entity:

A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:

— When a name has no linkage, the entity it denotes cannot be referred to by names from other scopes.

3.5p2

This issue is also discussed elsewhere.

declaration
only one if no
linkage

static
internal linkage

If the declaration of a file scope identifier for an object or a function contains the storage-class specifier **static**, the identifier has internal linkage.²²⁾ 425

Commentary

no linkage 435
block scope object
storage-class
specifier
syntax
declarator
syntax

If the same declaration occurs in block scope the identifier has no linkage. The keyword **static** is overworked in the C language. It is used to indicate a variety of different properties.

C++

3.5p3

A name having namespace scope (3.3.5) has internal linkage if it is the name of

- *an object, reference, function or function template that is explicitly declared **static** or,*
- *an object or reference that is explicitly declared **const** and neither explicitly declared **extern** nor previously declared to have external linkage; or*

```
1  const int glob; /* external linkage */
2          // internal linkage
```

identifier 422.1
declared in line file
identifier ??
definition
shall #include

Adhering to the guideline recommendations dealing with textually locating declarations in a header file and including these headers, ensures that this difference in behavior does not occur (or will at least cause a diagnostic to be generated if they do).

Other Languages

Some languages specify that all identifiers declared at file scope are not externally visible unless explicitly stated otherwise (i.e., the identifiers have to be explicitly exported through the use of some language construct).

Coding Guidelines

linkage 420
educating
developers

Developers often think in terms of the keyword **static** limiting the visibility of an identifier to a single translation unit. This describes the effective behavior, but not the chain of reasoning behind it.

extern identifier
linkage same as
prior declaration

For an identifier declared with the storage-class specifier **extern** in a scope in which a prior declaration of that identifier is visible,²³⁾ if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. 426

Commentary

The C committee were faced with a problem. Historically, use of the keyword **extern** has been sloppy. Different declaration, organizational styles have been used to handle C's relaxed, separate compilation model. This sometimes resulted in multiple declarations of the same identifier using the storage-class specifier **extern**. Allowing the linkage of an object declared using the storage-class specifier **extern** to match the linkage of any previous declaration maintained the conformance status of existing (when C90 was written) source code.

While there may be more than one declaration of the same identifier in a translation unit, if it is used in an expression a definition of it is required to exist. If the identifier has internal linkage, a single definition of it must exist within the translation unit, and if it has external linkage there has to be exactly one definition of it somewhere in the entire program.

C90

The wording in the C90 Standard was changed to its current form by the response to DR #011.

definition
one external
external
linkage
exactly one
external definition

Coding Guidelines

While an identifier might only be textually declared in a single header file, that header may be **#included** more than once when a source file is translated. A consequence of this multiple inclusion is that the same identifier can be declared more than once during translation (because it is a tentative definition).

tentative
definition

An identifier declared using the storage-class specifier **static** that is followed, in the same translation unit, by another declaration of the same identifier containing the storage-class specifier **extern** is considered to be harmless.

Example

```

1  static int si;
2  extern int si; /* si has internal linkage. */
3
4  static int local_func(void)
5  { /* ... */ }
6
7  void f(void)
8  {
9  extern int local_func(void);
10 }
11
12 static int glob; /* Declaration 1 */
13
14 void DR_011(void)
15 {
16 extern int glob; /* Declaration 2. At this point Declaration 1 is visible. */
17
18     {
19     /*
20      * The following declaration results in an identifier that refers
21      * to the same object having both internal and external linkage.
22      */
23     extern int glob; /* Declaration 3. Declaration 2 is visible here. */
24     }
25 }
```

427 21) There is no linkage between different identifiers.

footnote
21

Commentary

Nor is there any linkage between identifiers in different name spaces; only identifiers in the *normal* name space can have linkage. Identifiers whose spelling differ in significant characters are different identifiers. The number of significant characters may depend on the linkage of the identifier.

internal
identifier
significant charac-
ters
external
identifier
significant charac-
ters

C++

The C++ Standard says this the other way around.

A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:

3.5p2

Coding Guidelines

Translators that support a limited number of significant characters in identifiers may create a linkage where none was intended. For instance, two different identifiers (when compared using all of the characters appearing in their spelling) with external linkage in different translation units may end up referring to each other because the translator used does not compare characters it considers nonsignificant. This issue is discussed elsewhere.

identifier
number of charac-
ters

Example

Because there is no linkage between different identifiers, the following program never outputs any characters:

```

1  #include <stdio.h>
2
3  const int i = 0;
4  const int j = 0;
5
6  int main(void)
7  {
8  if (&i == &j)
9      printf("Linkage exists between different identifiers\n");
10 return 0;
11 }
```

22) A function declaration can contain the storage-class specifier **static** only if it is at file scope; see 6.7.1. 428

Commentary

This sentence appears in a footnote and as such has no normative meaning. However, there is other wording in the standard that renders use of the static storage-class at block scope to be undefined behavior.

C++

7.1.1p4 *There can be no **static** function declarations within a block, . . .*

This wording does not require a diagnostic, but the exact status of a program containing such a usage is not clear.

A function can be declared as a static member of a **class** (**struct**). Such usage is specific to C++ and cannot occur in C.

Other Languages

Languages in the Pascal/Ada family supports the nesting of function definitions. Such nested definitions essentially have static storage class. Java does not support the nesting of function definitions (although classes may be nested).

Common Implementations

gcc supports nested functions, as an extension, and the static storage-class can be explicitly specified for such functions. The usage is redundant in that they have block scope (inside the function that contains their definition) and are not visible outside of that block.

If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage. 429

Commentary

A prior declaration might not be visible because it does not exist or because a block scope declaration of the same identifier (in the same name space) hides it. Here the term *no linkage* is used to mean that a linkage of external or internal was not assigned (using the rules of the standard to interpret a prior declaration).

C90

The wording in the C90 Standard was changed to its current form by the response to DR #011.

Coding Guidelines

If a prior declaration exists, the situation involves more than one declaration of the same identifier. As such, it is covered by the guideline recommendation dealing with multiple declarations of the same identifier.

Example

```

1  extern int doit_1(void); /* external linkage. */
2  static int doit_2(void); /* internal linkage. */
3
4  void f(void)
5  {
6  int doit_1, /* no linkage. */
7     doit_2; /* no linkage. */
8
9     {
10    extern int doit_1(void); /* external linkage. */
11    extern int doit_2(void); /* external linkage. */
12    }
13 }

```

- 430 If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier **extern**.

function
no storage-class

Commentary

A significant amount of existing code omits the **extern** specifier in function definitions. The Committee could not create a specification that required **externs** to be inserted into long-existent source code. The behavior is as-if **extern** had appeared in the declaration. This does not automatically give the function external linkage (there could be a prior declaration that gives it internal linkage). This behavior is different from that used for object declarations in the same circumstances.

426 **extern
identifier**
linkage same as
prior declaration
431 **object**
file scope no
storage-class

Other Languages

Languages that use some sort of specifier to denote the visibility of file scope identifiers usually apply the same rules to functions and objects.

Coding Guidelines

Developers generally believe that a function declaration that does not include a storage-class specifier has external linkage. While this belief is not always true, the consequences of it being wrong are not far-reaching (i.e., a program may fail to link because a call was added in a different source file to a function having internal linkage when the developer believed it had external linkage). There does not appear to be a worthwhile benefit in a guideline recommendation that changes the existing practice of not including **extern** in function definitions.

426 **extern
identifier**
linkage same as
prior declaration

A future revision of the standard may require a storage-class specifier to appear in the declaration if a previous declaration of the identifier had internal linkage.

**identifier
linkage**
future language
directions

Example

```

1  static int f2(void);
2  int f2(void); /* Same behavior as-if extern int f2(void) had been written. */
3
4  extern int f(void)
5  { return 22; }
6
7  int g(void)
8  { return 23; }

```

- 431 If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

object
file scope no
storage-class

Commentary

There is no `as-if` here; the object is given external linkage. The behavior differs in more than one way from that for functions. If no storage-class specifier is given, the declaration is also a tentative definition. One possible reason for wanting to omit the storage-class specifier in the declaration of a file scope object is that such a declaration is also a definition (actually a tentative definition unless an explicit initializer is given). Another way of creating a definition at file scope with external linkage is to explicitly specify an initializer; in this case it does not matter if the storage-class specifier **extern** is or is not given.

Prior to C90, many implementations treated object declarations that contained both an **extern** storage-class specifier and an explicit initializer as a constraint violation. There was, and continues to be, a significant amount of existing code that omits the specification of any linkage on object declarations unless internal linkage is required. Without breaking existing code, the Committee continues to be in the position of not being able to change the specification, other than to make external linkage the default (when a storage-class specifier is not explicitly specified).

Support for declaring an identifier with internal linkage at file scope without the storage-class specifier **static** may be withdrawn in a future revision of the standard.

Coding Guidelines

The issue of what linkage objects at file scope should be declared to have is discussed elsewhere.

Example

```

1  static int sil; /* internal linkage. */
2  int sil;      /* external linkage, previous was internal, linkage mismatch. */
3
4  int eil;      /* No explicit storage-class given, so it is external.      */
5
6  extern int ei2; /* Same linkage as any prior declaration, otherwise external. */
```

The following identifiers have no linkage:

432

Commentary

Here the phrase *no linkage* is used to mean that the linkage is *none*.

an identifier declared to be anything other than an object or a function;

433

Commentary

This list includes tags, typedefs, labels, macros, and a member of a structure, union, or enumeration.

C++

3.5p3 *A name having namespace scope (3.3.5) has internal linkage if it is the name of*
— *a data member of an anonymous union.*

While the C Standard does not support anonymous unions, some implementations support it as an extension.

3.5p4 *A name having namespace scope (3.3.5) has external linkage if it is the name of*
— *a named class (clause 9), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (7.1.3); or*
— *a named enumeration (7.2), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (7.1.3); or*
— *an enumerator belonging to an enumeration with external linkage; or*

Names not covered by these rules have no linkage. Moreover, except as noted, a name declared in a local scope (3.3.2) has no linkage.

3.5p8

The following C definition may cause a link-time failure in C++. The names of the enumeration constants are not externally visible in C, but they are in C++. For instance, the identifiers E1 or E2 may be defined as externally visible objects or functions in a header that is not included by the source file containing this declaration.

```
1 extern enum T {E1, E2} glob;
```

There are also some C++ constructs that have no meaning in C, or would be constraint violations.

```
1 void f()
2 {
3 union {int a; char *p; }; /* not an object */
4                               // an anonymous union object
5
6 /*
7 * The following all have meaning in C++
8 *
9 a=1;
10 *
11 p="Derek";
12 */
13 }
```

434 an identifier declared to be a function parameter;

parameter
linkage

Commentary

A parameter in a function definition is treated as a block scope object with automatic storage duration. A parameter in a function declaration has function prototype scope.

Other Languages

Some languages allow arguments to be passed to a function in any order. This is achieved by specifying the parameter identifier that a particular argument is being passed to at the point of call. The identifiers denoting the function parameters are thus visible outside of the function body. In such languages parameter identifiers can only occur in the context of an argument list to a call of the function that defines them.

Many languages do not permit identifiers to be specified for function declarations that are not also definitions. In these cases only the types of the parameters may be specified.

435 a block scope identifier for an object declared without the storage-class specifier **extern**.

no linkage
block scope
object

Commentary

Block scope identifiers are only intended to be visible within the block that defines them. The linkage mechanism is not needed to resolve multiple declarations. Use of the storage-class specifier **extern** at block scope is needed to support existing (when the C90 was first created) code. Use of the storage-class specifier **static** at block scope creates an identifier with linkage none (the purpose of using the keyword in this context is to control storage duration, not linkage).

identifier
linkage at block
scope
static
storage dura-
tion

C++

3.5p8 *Moreover, except as noted, a name declared in a local scope (3.3.2) has no linkage. A name with no linkage (notably, the name of a class or enumeration declared in a local scope (3.3.2)) shall not be used to declare an entity with linkage.*

The following conforming C function is ill-formed in C++.

```

1 void f(void)
2 {
3     typedef int INT;
4
5     extern INT a; /* Strictly conforming */
6                 // Ill-formed
7
8     enum E_TAG {E1, E2};
9
10    extern enum E_TAG b; /* Strictly conforming */
11                       // Ill-formed
12 }
```

Other Languages

Few languages provide any mechanism for associating declarations of objects in block scope with declarations in other translation units.

Coding Guidelines

identifier
declared in one file 422.1

The guideline recommendation dealing with textually locating declarations in a header file is applicable here.

linkage
both internal/external

If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined. 436

Commentary

There are a few combinations of declarations where it is possible to give the same identifier both internal and external linkage. There are no combinations of declarations that can give the same identifier two other kinds of linkage.

EXAMPLE
linkage

C++

The C++ Standard does not specify that the behavior is undefined and gives an example (3.5p6) showing that the behavior is defined.

Common Implementations

Some translators issue a diagnostic for all cases where the same identifier appears with both internal and external linkage.

Coding Guidelines

An instance of this undefined behavior is most likely to occur when a .c file includes a header containing the declaration of an identifier for an object, where this .c file also contains a declaration using the storage-class specifier **static** for the same identifier (that is intended to be distinct from the identifier in the header). The situation occurs because the developer was not aware of all the identifiers declared in the header (one solution is to rename one of the identifiers). The same identifier is rarely declared with both internal and external linkage and a guideline recommendation is not considered worthwhile.

Example

```
1 extern int glob_0; /* external linkage. */
2 static int glob_0; /* internal linkage (prior declaration not considered). */
3
4 static int glob_1, glob_2; /* internal linkage. */
5     int     glob_2; /* Linkage is specified by the prior declaration. */
6
7 void f(void)
8 {
9     extern int glob_1; /* Same linkage as one visible at file scope. */
10
11     {
12         /*
13          * We need to enter another block to exhibit this problem.
14          * Now the visible declaration is in block scope, so the
15          * wording in clause 6.1.2.2 does not apply and we take the
16          * linkage from the declaration here, not the linkage from
17          * the outer scope declaration.
18          */
19         extern int glob_1; /* external and internal linkage. */
20     }
21 }
```

Usage

The translated form of this book's benchmark programs contained 27 instances of identifiers declared, within the same translation unit, with both internal and external linkage.

437 **Forward references:** declarations (6.7), expressions (6.5), external definitions (6.9), statements (6.8).

References

1. S. Srivastava, M. Hicks, J. S. Foster, and P. Jenkins. Modular information hiding and type-safe linking for C. In *Proceedings*

of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, pages 3–14, Apr. 2007.