The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.7 Declarations

declaration syntax

declaration:
declaration-specifiers *init-declarator-list*$_{opt}$;
declaration-specifiers:
storage-class-specifier *declaration-specifiers*$_{opt}$
type-specifier *declaration-specifiers*$_{opt}$
type-qualifier *declaration-specifiers*$_{opt}$
function-specifier *declaration-specifiers*$_{opt}$
init-declarator-list:
init-declarator
init-declarator-list , *init-declarator*
init-declarator:
declarator
declarator = *initializer*

Commentary

The intent of this syntax is for an identifier's declarator to have the same visual appearance as an instance of that identifier in an expression. For instance, in:

```
1  int x[3], *y, z(void);
2  char (*f(int))[];
```

the identifier `x` might appear in the source as an indexed array, `y` as a dereferenced pointer, and `z` as a function call. An example of an expression using the result of a call to `f` is `(*f(42))[1]`.

C90

Support for *function-specifier* is new in C99.

C++

The C++ syntax breaks declarations down into a number of different categories. However, these are not of consequence to C, since they involve constructs that are not available in C.

The nonterminal *type-qualifier* is called *cv-qualifier* in the C++ syntax. It also reduces through *type-specifier*, so the C++ abstract syntax tree is different from C. However, sequences of tokens corresponding to C declarations are accepted by the C++ syntax.

The C++ syntax specifies that *declaration-specifiers* is optional, which means that a semicolon could syntactically be interpreted as an empty declaration (it is always an empty statement in C). Other wording requires that one or the other always be specified. A source file that contains such a construct is not ill-formed and a diagnostic may not be produced by a translator.

Other Languages

Some languages completely separate the syntax of the identifier being declared from its type (requiring the type to appear to the left of the identifier being declared, or on the right). For instance, in the Pascal declaration:

```
1  x : array[1..3] of integer;
2  y : ^ integer;
3  function z() : integer; (* Function types don't follow the pattern. *)
```

the identifiers `x`, `y`, and `z` are separated from their type by a colon.

In a few languages the identifier being declared is also part of a declarator, which may include type information (e.g., an example array declaration in Fortran is `Integer Arr(3)`). Java syntax supports two methods of declaring objects to have an array type.

```
int [] a1,
       a2[];
int b1[],    // has the same type as a1
    b2[][];  // has the same type as a2
```

Ada does not allow object declarations to occur between function definitions. The intent[10] was to avoid the poor readability that occurs when items that are visibly smaller, because they contain few characters (an object declaration) are mixed with items that are visibly larger, because they contain many characters (a function definition). However, this restriction is cumbersome and was removed in Ada 95.

Common Implementations

Some implementations (e.g., most vendors target freestanding environments) support an extension that enables developers to specify the start address to be used for the object or function defined by the declaration. The token `@` is often used.

object specifying address

```
int i @ 0x800;
```

gcc supports the `__attribute__` in declarations. Quite a large number of different attributes are supported (an identifier denoting the attribute appears between a pair of parentheses). They can apply to functions, objects, or types. For instance, `int square (int) __attribute__((const));` specifies an attribute of the function `square`.

Coding Guidelines

The visual layout of declarations can affect the reader effort required to extract information and the likelihood of them making mistakes. Declaration layout has three degrees of freedom:

1. The source file selected to contain the declaration. This issue is discussed elsewhere.
2. The order in which different declarations occur within the same file or block (individual declarations can often occur in many different orders with the same effect). Identifiers declared at file scope are often read by people having more diverse information-gathering needs than those declared at block scope (where the use is usually very specific). Also, long sequences of identifier declarations usually only occur, with any frequency, at file scope. The number of declarations a block scope tend to be small. The quantity of file scope declarations and the diverse needs of readers suggests that an investment in organizing these declarations and providing more detailed commenting will produce a worthwhile benefit.
3. The layout of a single declaration and the token sequence used to specify it. (It is often possible to use different token sequences to specify the same declaration.)

declarations in which source file

identifiers number in block scope

The ordering of declarations at file scope

There are a number of patterns, in declaration ordering, commonly seen in existing source code:

- **`#include`** preprocessing directives are invariably placed before most other declarations in a file. This directive occupies a single line and placing all of them first ensures that subsequent declarations will be able to reference the identifiers they declare.
- All declarations of macros, typedefs, objects, and function declarations usually occur before the function definitions. The issue of ordering function definitions is discussed elsewhere.
- The grouping of declarations of macros, typedefs, objects, and function declarations. This grouping may be by kind of declaration (e.g., all macros, all typedefs, etc.), or it may be by some internal subdomain (e.g., all declarations related to a tree data structure, or the organization of program output, etc.).
- New declarations of a particular kind of identifier (e.g., macro or object) are often added at the end of the list of existing declarations of those kinds of identifiers.

function definition syntax

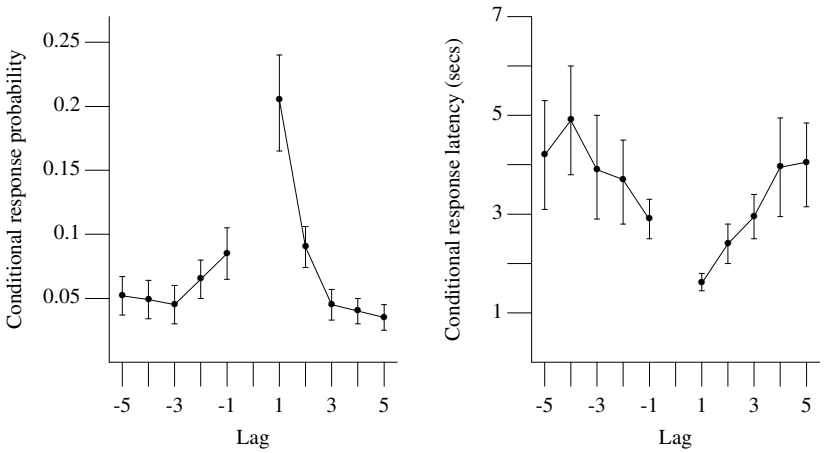

Figure 1348.1: The *lag recency effect*. The plot on the left shows the probability of a subject recalling an item having a given lag, while the plot on the right gives the time interval between recalling an item having a given lag (error bars give 95% confidence interval). If a subject, when asked to remember the list "ABSENCE HOLLOW PUPIL", recalled "HOLLOW" then "PUPIL", the recall of "PUPIL" would have a lag of one ("ABSENCE" followed by "PUPIL" would be a lag of 2). Had the subject recalled "HOLLOW" then "ABSENCE", the recall of "ABSENCE" would be a lag of minus one. Adapted from Howard and Kahana.[8]

Peoples performance in recalling, recently remembered, words from a list has been found to have a number of characteristics. Studies by Howard and Kahana[7,8] investigated the probability of a particular word on a list being recalled when subjects were given the name of another word. The results showed (see Figure 1348.1) that the words immediately following the words presented were most likely to be recalled.

When subjects were asked to perform free recall of a list of words they had seen, successive recalled words tended to be related in some way (responses could be given in any word order) rather than being completely unrelated. A measure of pairwise similarity between the words presented was calculated using latent semantic analysis. The results showed that these LSA similarity values were highly predictive of the order in which subjects recalled words.

latent semantic analysis

reading
kinds of

What are the costs and benefits of putting declarations in a particular order? As discussed elsewhere developers tend to read source on an as-needed basis. A developer is unlikely to need to read a declaration unless it declares an identifier that is referenced from other source that is currently being read. Reading an identifier's declaration, once found, may in turn create the need to read other identifier declarations.

IDE

To locate an identifier using a text editor, a developer might go to the top of the file and use a search command to find the first occurrence. Within an IDE, it is often possible to click on an identifier to bring up a window containing its declaration. A developer might also perform a quick visual search.

If the information required relates to a single identifier, the cost is the search time for that identifier declaration. The cost of obtaining information on N identifiers can be much greater than N times the search cost of a single identifier declaration. Reading a declaration can be like walking a tree— one declaration leading to another and then back to the original for more information extraction.

Having all related declarations visible on the screen at the same time (perhaps in multiple windows) minimizes the keyboard and mouse activity needed to switch between different declarations.

How can declarations be ordered to minimize total cost? Answering this question requires usage information. What percentage of searches involve locating a single identifier declaration and what percentage involve reading multiple declarations? To what extent do related declarations require readers to visually skip back and forth between them? Unfortunately, this usage information is not available.

Ordering declarations alphabetically, based on the identifier declared, may improve the performance of some identifier search strategies (e.g., paging up/down until the appropriate sorted identifier is reached). However, such an ordering may be difficult to achieve in practice because of the need to declare an identifier before it is referenced and because of the complications introduced by those declarations that contain more

than one identifier (e.g., enumeration types). Also there is nothing to suggest how large improvements in search performance will be, or whether it will give alphabetic ordering the edge over other orderings. Alphabetic ordering of identifier declarations is not discussed further here.

The following discussion assumes that for all ordering of declarations, developer search time is approximately the same. The important factor then becomes the ease with which developers can view multiple, related declarations.

If there is a dependency between two declarations, it is likely that a developer will want to read both of them. Placing dependent declarations sequentially next to each other in the visible source could reduce the need to switch between views of them (using the keyboard or mouse to change display contents). Both declarations being visible on the display at the same time. However, for all but the simplest declarations, placing dependent declarations sequentially next to each other is rarely possible:

- There may be many references to a particular identifier. How should the many declarations making these references be ordered?
- There may be many identifiers referred to in one declaration. For instance, the members of a structure type may have a variety of types. How would such multiple referenced identifiers be ordered?

Looking at existing source shows that in many cases declarations are grouped by categories (categorization is discussed in the Introduction). The chose of categories, by developers, seems to have been based on a variety of reasons, including:

categorization

- All objects declared to have the same type
- All function declarations
- Object declarations holding semantically related information (where it was not considered to worthwhile create a structure type to hold them)
- Objects having static linkage
- all declarations related to some data structure; for instance, types used in the manipulation of tree data structures may include one or more structure types, perhaps a union type, and a variety of enumeration or macro definitions (giving symbolic names to the numeric values used to denote various kinds of tree nodes)

symbolic name

- Grouped in the order in which they were originally written, which is likely to have been influenced by the order (both in time and space, within the file) in which function definitions were written

Placing declarations in the same category adjacent to each other is using an ordering relationship. (Alphabetic ordering is another one.) It might be claimed that category ordering is an effective heuristic for minimizing the developer cognitive and motor cost of switching between different views of the source while attempting to comprehend a declaration. However, there is no evidence to support this claim, or indeed any other claims of cost-effective declaration orderings.

Visual layout of adjacent and single declarations

declaration visual layout

It is quite common to see sequences of object declarations where the identifiers being declared are vertically aligned with each other. (A number of rationales can be heard for having such an alignment: ease of readability and "it looks nice" are both frequently given.) This discussion concentrates on the visual issues of adjacent identifier declarations. Vertical alignment creates an edge. When scanning the source, edges are something that readers can use to help control what they look at.

Edge detection
reading
kinds of
gestalt principles
continuation
gestalt principle of

The gestalt principle of continuation suggests that aligning identifiers creates an association between them. The principle of similarity (types in one set, objects being declared in the other) does not occur visually, although it may exist as a category in the reader's mind. Do we want to associate identifiers from different declarations with each other? The issue of these associations and reader categories are discussed elsewhere.

1358 declarator
list of

Many declarations declare a single identifier on each source line. Developers often visually search lists of declarations by scanning up or down those declarations. Consequently, when a second identifier is declared on the same line, it is sometimes missed. As the following example shows, the surrounding declarations play an important role in determining how individual identifiers stand out, visually. A reader looking for the identifiers `c` or `total_count` is likely to notice that more than one declaration occurs on the same line. However, a search for `zap` may fail to notice it because of the degree to which its declaration blends in to give the appearance of a single declaration (perhaps `zip_zap`).

```
1  int a, b, c;
2
3  int foo_bar,
4      zip,zap,
5      win_lose;
6
7  int local_val,
8      max_count, total_count,
9      global_val;
10
11 void j(int k, int l);
```

A guideline recommendation based on visual appearance would be difficult to word and very difficult for developers to judge and for tools to enforce. The simple solution is to have a simple rule covering all cases (accepting that in some situations it is redundant).

Cg 1348.1 No more than one *init-declarator* shall occur on each visible source code line.

What about identifiers appearing in other kinds of declarations?

- *Preprocessor syntax* only permits one macro to be defined per line.
- *Function declarations.* The function name usually appears in the same location as it would in an object declaration (having the same type as the function return type). Additional information is provided by the parameter declarations. The parameter information is usually only read after the function name has been located; it is rarely searched for independently of the function name. Given the localized nature of visual searches of parameter information, there is no reason to split these declarations across multiple lines.
- *Function definitions.* The identifiers declared in a function definition's parameter list need to be included in any search of locally defined objects.

enumeration
set of named constants
struct-??
declarator
one per source line

- *Members of enumeration types.* This issue is discussed elsewhere.
- *Members of structure and union types.* This issue is discussed elsewhere.

Cg 1348.2 For the purposes of visual layout the parameters in a function definition shall be treated as if they were *init-declarator*s.

type specifier
syntax

The relative ordering of some tokens within a declaration is variable. The syntax does not impose any relative ordering on type specifiers, type qualifiers, or storage-class specifiers. This issue is discussed in elsewhere.

An object having static storage duration in block scope is rare. Whether there is a worthwhile cost/benefit in drawing attention to such a definition is open to debate.

identifier definition close to usage
grouping
spatial location

There are benefits in visually grouping related items near each other. However, since it is often possible to comprehend statements without detailed knowledge of the identifiers they contain, the cost difference

of them not being visually close may be small. Given that approximately 70% of block scope declarations (excluding parameters) occur in the outermost block scope (see Figure **??**) and many function definitions are short enough (see Figure **??**) to be completed viewed on a display, the applicable definition may already be easy to locate (many identifiers are not declared within the minimum block scope required by their usage, Figure **??**).

There is cost associated with declaring identifiers within the minimum required scope, compared to declaring them in the outermost block. When writing the body of a function definition a developer may be unsure of the minimal required scope of some identifiers. Developers might chose to move declarations on an as-needed basis or might choose to move declarations to the appropriate scope once the body is finalized. Whichever choice is made there is a cost that needs to be paid. Also during maintenance the required scope may change. This cost of deducing and maintaining local declarations in a *minimal* scope appears to be greater than the benefits.

Example

```
1  typedef int foo,
2              bar;
3
4  void f(void)
5  {
6  foo(bar);               /* Declare bar as an object.   */
7  extern int foo(void);   /* Declare foo as a  function. */
8  }
```

Table 1348.1: Occurrence of types used in declarations of objects (as a percentage of all types). Adapted from Engblom[5] and this book's benchmark programs.

Type	Embedded	Book's benchmarks
integer	55.97	37.5
float	0.05	1.6
pointer	22.08 (data)/0.23 (code)	48.2
struct/union	9.88	6.1
array	11.80	6.6

Table 1348.2: Occurrence of types used to declare objects in block scope (as a percentage of all such declarations). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
int	28.1	**long**	3.0
struct *	27.7	**union** *	2.9
other-types	10.8	**unsigned short**	2.3
unsigned int	5.5	**unsigned char**	2.0
struct	4.9	**char**	1.8
unsigned long	4.8	**char** []	1.5
char *	3.5	**unsigned char** *	1.3

Table 1348.3: Occurrence of types used to declare objects with internal linkage (as a percentage of all such declarations). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
`int`	20.9	`const char []`	2.4
other-types	14.4	`unsigned int`	1.8
`struct`	13.0	`const struct`	1.8
`struct *`	8.2	`void *()`	1.7
`struct []`	7.4	`const unsigned char []`	1.6
`( const char * const ) []`	4.0	`unsigned int []`	1.4
`unsigned char []`	3.4	`int *()`	1.4
`unsigned short []`	3.3	`( struct * ) []`	1.3
`int []`	2.9	`( char * ) []`	1.3
`char *`	2.8	`unsigned long`	1.2
`char []`	2.7	`const short []`	1.2

Table 1348.4: Occurrence of types used to declare objects with external linkage (as a percentage of all such declarations). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
`int`	22.8	`char *`	3.2
`const char []`	15.4	`union *`	3.0
other-types	10.6	`enum`	2.4
`struct *`	10.3	`float`	1.4
`const struct`	10.2	`char []`	1.4
`struct`	8.2	`unsigned int`	1.2
`void *()`	4.6	`int []`	1.2
`struct []`	4.1		

Constraints

declaration shall declare identifier

A declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration. 1349

Commentary

redundant code

It is C's unusual declaration syntax that makes it possible to write a declaration that declares no identifier in the scope of the declaration. A declaration that did not declare a declarator might be regarded as simply redundant. However, such usage is suspicious; it is likely that the developer has made a mistake. This wording requires that a declaration declares an identifier that is visible in the scope of the declaration.

C90

6.5 *A declaration shall declare at least a declarator, a tag, or the members of an enumeration.*

The response to DR #155 pointed out that the behavior was undefined and that a diagnostic need not be issued for the examples below (which will cause a C99 implementation to issue a diagnostic).

```
1  struct { int mbr; }; /* Diagnostic might not appear in C90. */
2  union { int mbr; };  /* Diagnostic might not appear in C90. */
```

Such a usage is harmless in that it will not have affected the output of a program and can be removed by simple editing.

Other Languages

The declaration syntax of most languages does not support a declaration that does not declare an identifier.

Example

```
struct {
        int m1; /* Constraint violation. */
       };
struct {
        struct T {int m11;} m1; /* Declares a tag. */
       };

typedef enum E { E1 }; /* Pointless use of a typedef. */

long int; /* No declarator. */

; /* Syntactically a statement. */
```

1350 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except for tags as specified in 6.7.2.3.

declaration only one if no linkage

Commentary

Identifiers with no linkage include those having block and function prototype scope, those in the label name space, and typedef names. The special case for tags is needed to handle recursive declarations, which are discussed elsewhere.

linkage kinds of

type contents defined once

C++

This requirement is called the *one definition rule* (3.2) in C++. There is no C++ requirement that a typedef name be unique within a given scope. Indeed 7.1.3p2 gives an explicit example where this is not the case (provided the redefinition refers to the type to which it already refers).

C++ one definition rule

A program, written using only C constructs, could be acceptable to a conforming C++ implementation, but not be acceptable to a C implementation.

```
typedef int I;
typedef int I; // does not change the conformance status of the program
               /* constraint violation */
typedef I I;   // does not change the conformance status of the program
               /* constraint violation */
```

Other Languages

Some languages only allow one declaration of the same identifier in the same scope and name space. Those that require declaration before usage and support recursive data structures or recursive function calls need to provide a *forward* declaration mechanism (Pascal actually uses the keyword **forward**). Use of this mechanism effectively creates two declarations of the same identifier.

1351 All declarations in the same scope that refer to the same object or function shall specify compatible types.

declarations refer to same object
declarations refer to same function

Commentary

Occurrences of multiple declarations specifying incompatible types are probably the result of some developer mistake. Specifying that an implementation choose one of them is unlikely to be of benefit to developers, while generating a diagnostic pointing out the usage enables the developer to correct the mistake.

C++

3.5p10 *After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4). A violation of this rule on type identity does not require a diagnostic.*

C++ requires identical types, while C only requires compatible types. A declaration of an object having an enumerated type and another declaration of the same identifier, using the compatible integer type, meets the C requirement but not the C++ one. However, a C++ translator is not required to issue a diagnostic if the declarations are not identical.

```
1  enum E {E1, E2};
2
3  extern enum E glob_E;
4  extern int glob_E; /* does not change the conformance status of the program */
5                     // Undefined behavior, no diagnostic required
6
7  extern long glob_c;
8  extern long double glob_c; /* Constraint violation, issue a diagnostic message */
9                             // Not required to       issue a diagnostic message
```

Other Languages

Languages that support some form of duplicate type declarations usually require that the types be the same.

Coding Guidelines

identifier ?? declared in one file

Adhering to the guideline recommendation specifying the placement of declarations of identifiers having file scope in a single file helps ensure this constraint is met.

Semantics

---

declaration interpretation of identifier

A declaration specifies the interpretation and attributes of a set of identifiers. 1352

Commentary

value

For instance, the type of an object determines how its contents are to be interpreted. Attributes of the declaration of an object might be its storage class, while the textual location of the declaration decides the scope of the identifiers it declares.

Other Languages

Some languages do not require that identifiers appear in a declaration. Their interpretation and attributes are deduced from the context in which they occur and the operations performed on them. In other languages (e.g., Visual Basic, Perl, and Scheme) the type information associated with an object can change depending on the last value assigned to it. Values have type and this type information is assigned to an object, along with the value. Values are said to be tagged with their type. (The Scheme revised report[12] refers to the language as using *latent* types, while other terms include *weakly* or *dynamically* typed.) BCPL requires that identifiers be declared, but deduces their type from the usage.

Common Implementations

Extensions to C declarations often include creating additional specifiers that can appear in a declaration. For instance, the keywords **near/far/huge** used to specify how a pointer is to be interpreted.

Coding Guidelines

The issue of encoding information on some of these attributes is discussed elsewhere.

naming conventions

Judging whether an identifier is used in ways that involve different sets of semantic associations is something that is not yet possible, in general, using an automated tool.

Some coding guideline documents recommend that instances of particular objects, within a program, only be used for a single purpose. For instance, as a loop counter, or to hold the running total of some quantity. Using the same object for both purposes, assuming the two uses do not overlap in the source code, may reduce the total amount of storage used but experience shows that it generally increases the effort needed to comprehend the code and the likelihood of faults being introduced.[1352.1]

object role

The *purpose* of an object is rather difficult to pin down. However, the way in which an object is used (its *role* in the source) often follows a pattern that can be categorized. The role categories discussed in other coding guideline subsections are based on the kinds of values an object might have (e.g., boolean and bit-set roles) and the operators that might be applied to them. Sajaniemi[18] proposed a role classification scheme based on an analysis of a higher level of abstraction of how objects are used in the code. For instance:

boolean role bit-set role

- *constant*: an object whose value does not change after initialization,
- *stepper*: an object that iterates through a series of values that do not depend of the values of other nonconstant variables,
- *follower*: an object that iterates through a series of values that depended on the value of a stepper variable,
- *most-recent holder*: an object holding the latest value encountered during the processing of a sequence of values,
- *most-wanted holder*: an object holding the best value encountered during the processing of a sequence of values,
- *gatherer*: an object whose value represents the accumulated affect (e.g., the sum) of individual values encountered during the processing of a sequence of values,
- *one-way flag*: an object whose value is changed once, while processing some sequence of values,
- *temporary*: an object holding a value over a short sequence of code,
- *other*: all other objects.

In other, more strongly typed, languages role information is sometimes encoded as part of the type. For instance, two different integer types representing apples and oranges might be defined, with the intent that any attempt to add an object having type apple to an object having type orange will cause a diagnostic to be issued.

Is there a worthwhile benefit in limiting every object to having a single role? Experience suggests that most objects are used in a way that involves a single role (at least in programs written by non-novice developers). The storage cost associated with defining additional objects is rarely a significant consideration (although there are application domains where this cost can be very significant; and some translators attempt to minimize the storage used by a program).

1354 storage layout

Experience shows that using the same object is more than one role increases the effort needed to comprehend the code and the likelihood of faults being introduced (e.g., developers be aware of and remember the disjoint regions of source code over which an object has a particular role). The concept of an objects role is relatively new and is not specified in these coding guidelines in sufficient detail to be automatically enforced. However, this is an issue that can be covered during code review.

[1352.1] Programs written in dialects of Basic that limit the number of objects that can be used (e.g., using single letters of the alphabet to denote identifiers limits the number of objects to 26) are often forced to use the same object for different purposes in different parts of the source (e.g., using it as a loop counter, then to hold the result of some calculation, then as a flag, and so on).

Rev 1352.1 If an object has one of the roles defined by these coding guidelines it shall always be used in a way that is consistent with that role.

---

definition identifier

A *definition* of an identifier is a declaration for that identifier that: 1353

Commentary

label declared implicitly tag declare macro object-like macro function-like object external definition external definition tentative definition definition one external internal linkage exactly one external definition

This defines the term *definition*. It excludes some kinds of identifiers (labels and tags), which can be declared, and macros, which are said to be *defined* but are discussed in a separate subclause. It is common developer usage to refer to a declaration of an identifier as a label or tag— as its definition. The standard also defines the terms *external definition* (which is a definition) and *tentative definition* (which eventually might cause a definition to be created). Limits on the number of definitions of an identifier are specified elsewhere.

C++

The C++ wording is phrased the opposite way around to that for C:

A declaration is a definition unless . . .

3.1p2 tentative definition

The C++ Standard does not define the concept of tentative definition, which means that what are duplicate tentative definitions in C (a permitted usage) are duplicate definitions in C++ (an ill-formed usage).

```
1  int glob;     /* a tentative definition */
2                // the definition
3
4  int glob = 5; /* the definition */
5                // duplicate definition
```

Other Languages

Only languages that support separate translation of source files, or some form of forward declarations, need to make a distinction between declarations and definitions.

Coding Guidelines

Many developers do not distinguish between the use of the terms *declaration* and *definition* when discussing C language. While this might be technically incorrect, in the common cases the distinction is not important. The benefit of educating developers to use correct terminology is probably not worth the cost.

---

object reserve storage

— for an object, causes storage to be reserved for that object; 1354

Commentary

declaration 1348 syntax lifetime of object

The Declarations clause of the C Standard does not specify what causes storage to be reserved for an object. This information has to be deduced from the definition of an object's lifetime.

C++

The C++ Standard does not specify what declarations are definitions, but rather what declarations are not definitions:

3.1p2 *. . . , it contains the* **`extern`** *specifier (7.1.1) or a linkage-specification[24] (7.5) and neither an* `initializer` *nor a* `function-body`*, . . .*

7p6

An object declaration, however, is also a definition unless it contains the **`extern`** *specifier and has no initializer (3.1).*

Common Implementations

storage layout
byte
address unique
unary &
operator
register
storage-class
program
image
operator
()
identifiers
number in block
scope

Where is storage reserved for the object? The individual bytes of an object have a unique address, but unless its address is taken, storage need not be allocated for it. The issue of maintaining objects in registers only is discussed elsewhere.

In the majority of implementations storage for static duration objects is allocated in a fixed area of storage during program startup and that for automatic duration objects on a stack. Many translators assign storage to objects in the order in which they encounter their declarations during translation.

Most function definitions do not contain very many object definitions and the combined storage requirements of these definitions is not usually very large (see Figure **??**). Many processors (including RISC) contain a form of addressing designed for efficient access to local storage locations allocated for the current function invocation— *register* + *offset* addressing mode. The processor designers choosing the maximum possible value of *offset* to be large enough to support commonly occurring amounts of locally allocated storage. In those cases where the offset of a storage location is outside of the range supported by a single instruction, multiple instructions need to be generated. The following are two opportunities for optimizing the allocation of locally defined objects to storage locations:

register + offset

1. When the quantity of local storage required by a function definition is larger than the maximum *offset* implicitly supported by the processor. Many processors designed for embedded systems applications support relatively small *offset*s. Accesses to objects with greater offsets require the use of multiple instructions, making them more costly in time and code size. Burlin[3] uses a greedy algorithm to assign storage locations to objects, based on their access costs and the amount of storage they required.

2. On processors that contain a cache, reading a value from a storage location will also cause values from adjacent storage locations to be read (the number of storage locations read will depend on the size of a processor's cache line). Ordering objects in storage so that successive accesses to different storage locations, during program execution, are on the same cache line results in several savings.[15] The access time to other storage locations on the same cache line is reduced and the turnover of cache lines (filling a new cache line requires the contents of an existing line to be overwritten) is reduced. The relative order of objects on a cache line may also need to be considered. On most processors the contents of a cache line are filled, from storage, in a particular order. In most cases the first location loaded is the one that caused the cache miss, followed by the next highest storage location, and so on (wrapping around to any lower addresses). For instance, assuming a 32-byte cache line filled 4 bytes at a time, a read from address xx04 that causes a cache miss, will result in storage locations xx00 through xx1f being loaded in the order xx04, xx05, . . . , xx1f, xx00, xx01, xx02, xx03. Some processors fill the cache line completely before its contents become available, while others (e.g., IBM PowerPC) forward bytes as soon as they become available. Four successive bytes will become available on each of eight successive clock cycles. (There will be some storage latency before the first four bytes arrive.) Matching the order of objects in the cache line to the order in which they are most frequently accessed at execution-time minimizes latency. Knowing which objects are frequently accessed requires information on the execution time behavior of a program.[17]

cache

The assignment of objects, with static storage duration, to storage locations is usually performed by the linker. The relative ordering is often the same as the order in which the linker encounters object definitions within object files, which may or may not be the order in which the translator encounters them in the source file. The following are three main opportunities for optimizing the allocation of objects with static storage duration.

1. Some implementations use *absolute* addressing to access objects and others use *register*+*offset* (where

register + offset

the register holds the base address of storage allocated to global data). In both cases there will be a limit on the maximum offset that can be accessed using a single instruction— a larger offset requires more instructions.

2. The cache issues are the same as for local objects.

3. Some processors (usually those targeted at embedded systems[9]) include a small amount of on-chip storage. Also some processors support pointers of different sizes (i.e., 8-bit and 16-bit representations), with the smaller size executing more quickly but only being able to access a relatively small area of storage. Deciding what objects to allocate to those storage locations that can be accessed the quickest requires whole program analysis. Sjôdin and von Platen[20] mapped this optimal storage-allocation problem to an integer linear-programming problem and were able to achieve up to 8% reduction in execution time and 10% reduction in machine code size. Avissar, Barua, and Stewart[1] also used linear programming to obtain an 11.8% reduction in execution time, but by distributing the stack over different storage areas obtained a 44.2% reduction in execution time.

banked storage

Optimizing storage layout for embedded applications can involve trading off execution time performance, amount of storage required, and power consumption.[14] Some processors use banked storage, which can offer a potential solution to the problem of the growing difference in performance between the time taken to execute an instruction and the time taken to access a storage location. Optimizing the allocation of objects across banked storage is an active research area.[2]

The total amount of physical storage available to a program never seems to be enough for some applications. In most cases developers rely on a host's support for virtual memory management. Alternative solutions are for the program to explicitly manage its own large storage requirements, or for the translator to provide a mechanism that allows developers to specify what objects should be swapped to disk and when (this usually involves the use of extensions[4]).

Many storage allocation algorithms,[21] for allocated storage duration, are based on measurements of programs that have relatively short running times. Programs with long running times (e.g., server applications,[16] and multithreaded applications[13]) are less well represented.

A storage allocation policy based on *best fit* (i.e., using the smallest available free storage large enough to satisfy the request) has been found to give good results in practice.[11] The `CustoMalloc`[6] tool uses dynamic profile information, on the size of the object requests made, to build a custom storage allocator tuned to a given program (or at least the profiling data). Execution time profiling of references to objects with allocated storage duration can also be used to improve a program's performance by segregating objects with similar behaviors (leading to improved reference locality).[19]

cache alignment structure type sequentially allocated objects array row-major storage order

Various other storage layout issues are discussed in other C sentences.

Coding Guidelines

For a variety of reasons some developers take advantage of the relative addresses of storage allocated to objects. There is no guarantee where block scope objects will be allocated storage relative to each other, and the relative addresses of objects having static storage duration is even more unpredictable. There is a simpler method of outlawing such usage than enumerating all cases where a program could depend on the layout of block scope storage. The following guideline renders any such dependence useless.

Cg 1354.1 Arbitrarily reordering the object declarations in the same scope shall not result in a change of program output.

Dev 1354.1 When an object declaration contains a dependency on another object declared in the same scope the arbitrary reordering may be restricted to those orderings that do not result in a constraint violation.

Example

```
#include <string.h>

extern int e_g_1;  extern int e_g_2;
static int s_g_1;  static int s_g_2;

void f(void)
{
       int loc_1;         int loc_2;

if ((&e_g_1 + 1) == &e_g_2)
   {
   memset(&e_g_1, sizeof(e_g_1) * 2, 0xff);
   /*
    * No requirement that (&s_g_1 + 1) == &s_g_2) be true.
    */
   memset(&s_g_1, sizeof(e_g_1) * 2, 0xff);
   /*
    * No requirement that (&loc_1 + 1) == &loc_2) be true.
    */
   memset(&loc_1, sizeof(e_g_1) * 2, 0xff);
   }
}
```

1355 — for a function, includes the function body;[99)]

Commentary

A function definition is intended to contain executable statements (that may perform some action) which occupy storage space.

function definition syntax

C++

The C++ Standard does not specify what declarations are definitions, but rather what declarations are not definitions:

A declaration is a definition unless it declares a function without specifying the function's body (8.4), . . . 3.1p2

Other Languages

Languages that have a mechanism to support the building of programs from separately translated source files usually refer to the function declaration that contains its body as its definition.

1356 — for an enumeration constant or typedef name, is the (only) declaration of the identifier.

Commentary

Since only one declaration is allowed in the same scope for these kinds of identifiers it might be more accurate to always call it a definition. However, using this convention would create more wording in the standard than it saves (because some of the wording that applies to declarations would then need to be extended to include definitions; specifying these identifiers to be both declarations and definitions removes the need for this additional wording).

C90

The C90 Standard did not specify that the declaration of these kinds of identifiers was also a definition, although wording in other parts of the document treated them as such. The C99 document corrected this defect (no formal DR exists). All existing C90 implementations known to your author treat these identifiers as definitions; consequently, no difference is specified here.

C++

3.1p2 *A declaration is a definition unless . . . , or it is a* ***typedef*** *declaration (7.1.3), . . .*

A typedef name is not considered to be a definition in C++. However, this difference does not cause any C compatibility consequences. Enumerations constants are not explicitly excluded in the *unless* list. They are thus definitions.

Other Languages

Languages that contain these constructs usually refer to their declarations as definitions of those identifiers.

---

declaration specifiers

The declaration specifiers consist of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. 1357

Commentary

This says in words what the syntax specifies. Like the syntax there are no restrictions placed on the ordering of specifiers.

C++

The C++ Standard does not make this observation.

Common Implementations

Some implementations contain extensions that add to the list of attributes that can be specified by sequences of specifiers. These are dealt with in their respective sentences.

Coding Guidelines

Are there any advantages to imposing some kind of ordering on declaration specifiers? There would be if all developers used the same ordering (on the basis that source code readers would not have to scan a complete declaration looking for the information they wanted; like looking a word up in a dictionary, it would be in an easy-to-deduce position). Analysis of existing source (see Table **??**, Table **??**, and Table **??**) shows that the most commonly seen ordering is *storage-class-specifier type-qualifier type-specifier*. Given developers' existing experience with this ordering, the likelihood that it will be the ordering they are most likely to encounter in the future, and the lack of any obvious benefit to using another ordering, the recommendation is to continue using this ordering.

Cg 1357.1

Within a declaration the declaration specifiers shall occur in the order *storage-class-specifier type-qualifier type-specifier*.

storage-class specifiers
future language directions
type specifiers ordering

Placing a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is specified as being obsolescent (the same specification also appeared in C90). The relative ordering of *type-specifier* is discussed elsewhere.

Example

```
1  int const typedef I;
2  double const long static J;
3  int volatile long extern long const unsigned K;
```

---

declarator list of

The init-declarator-list is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. 1358

Commentary

This says in words what is specified in the syntax.

C++

The C++ Standard does not make this observation.

Coding Guidelines

The C declaration syntax allows type information to be specified in the *declarator*. It is possible for different *init-declarator*'s, each declaring a different identifier, in an *init-declarator-list* to declare identifiers having different types.

```
int i_1,
    *p_1,        /* *p_1 is the declarator, a pointer to ... */
    **p_2,       /* **p_2 is the declarator, a pointer to pointer to ... */
    * const p_3, /* * const p_3 is the declarator, a const pointer to ... */
    a_1[10],     /* a_1[10] is the declarator, an array of ... */
    *(a_p_1[4]); /* *(a_p_1[4]) is the declarator, an array of pointer to ... */

char *pc_1,
      pc_2;
```

A mistake commonly made by inexperienced developers is to assume that pc_2, in the above example, has type pointer to **char**. A similar mistake does not seem to occur very often for array types. This may be due to the significantly smaller number of array declarations, compared to pointer declarations, in C or because the additional tokens are not visually contiguous with the declaration specifiers but to the right of the identifier being declared.

Cg 1358.1 A *declarator* specifying a pointer type shall not occur in the same *init-declarator-list* as *declarator*s not specifying a pointer type or specifying a different pointer type.

Objects that are used to store the same set of values are usually declared to have the same type. A change to the set of values that needs to be represented usually requires changing the declarations of all the associated objects. The following are a number of techniques that can be used to declare more than one object:

- Multiple *declaration*s each of whose *init-declarator-list* contains a single *init-declarator*.

  ```
  int i;
  int j;
  int k;
  ```

  Some coding guideline documents recommend this technique. The rationale is that it reduces the likelihood of cut-and-paste editor operation mistakes, or modifications to the *declaration-specifiers* having unexpected results. (Because a single object is declared, it is not possible to affect the declaration of another object.) The disadvantage of this approach is that, if it is necessary to change the *declaration-specifiers* associated with a set of object declarations, each *declaration* has to be modified (it is possible this change will not be applied to some of them).

- A single *declaration* whose *init-declarator-list* contains more than one *init-declarator*.

  ```
  int i,
      j,
      k;
  ```

  The advantages and disadvantages of this technique are those of the above, reversed. Experienced developers will be familiar with cut-and-paste mistakes when modifying this kind of declaration. The extent to which either form of declaration forms a stronger visual association, or reduces the effort needed to read the identifier list, is not known.

- Multiple *declaration*s using a typedef name.

```
1  typedef int francs; /* French francs */
2
3  francs i;
4  francs j;
5  francs k;
```

  This approach appears to solve the *declaration-specifiers* modification disadvantages associated with the first technique and yet has its advantages, but only because it associates a unique name, `francs`, with the type that is common to the three declared objects.

  symbolic name

- Some combination of the above.

Without any experimental evidence, or record of faults introduced, it is not possible to verify any claims about either of the first two techniques being more or less prone to the creation of faults.

Some coding guideline documents recommend that *type-specifier*s other than *typedef-name* only appear in the definition of a typedef name. Such a guideline recommendation is easily followed without addressing the underlying issues. For instance, in the `francs` example above, using a typedef name of `INT` does not solve any of the problems. Because the name `INT` is not specific to the information represented in `i`, `j`, and `k` only. Such a typedef name is also likely to be used to declare objects that do not hold values denoting French francs. This issue is discussed in more detail elsewhere.

typedef name syntax

---

footnote 99

99) Function definitions have a different syntax, described in 6.9.1. 1359

Commentary

function definition syntax

During syntactic analysis of a sequence of tokens, the first difference between a function declaration and a function definition occurs when either a **;** or **{** token is encountered (which in the case of **;** is the last token of a function declaration).

Other Languages

In most languages the syntax for function definitions is usually very different from object definitions. The evolving designs of object-oriented languages is starting to blur the distinction.

The declarators contain the identifiers (if any) being declared. 1360

Commentary

declaration 1349 shall declare identifier
abstract declarator syntax

A declarator always declares at least one identifier. However, a declaration need not include a declarator, although it must declare some identifier. An abstract declarator need not declare any identifiers.

---

object type complete by end

If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer; 1361

Commentary

incomplete types footnote 109

object type for internal linkage

The rationale behind the support of incomplete types does not apply to objects having no linkage. Objects with other kinds of linkage may be declared to have an incomplete type. An initializer can only complete an incomplete array type in this context. Requirements on the completion of types of objects having external and internal linkage are discussed elsewhere.

C++

3.1p6 *A program is ill-formed if the definition of any object gives the object an incomplete type (3.9).*

The C++ wording covers all of the cases covered by the C specification above.

A violation of this requirement must be diagnosed by a conforming C++ translator. There is no such requirement on a C translator. However, it is very unlikely that a C implementation will not issue a diagnostic in this case (perhaps because of some extension being available).

Other Languages

Those languages supporting declarations that need storage allocation decisions to be made at translation time (in nearly all cases this applies to locally declared objects) have similar requirements.

Common Implementations

Although not required to do so, all known implementations issue a diagnostic if this requirement is not met.

Coding Guidelines

The issues associated with using the number of initializers to specify the number of elements in an array type are the same for declarations having any linkage, and are discussed elsewhere.

array of unknown size initialized

1362 in the case of function ~~arguments~~ parameters (including in prototypes), it is the adjusted type (see 6.7.5.3) that is required to be complete.

Commentary

Any adjusted parameter type (the case applies to function parameters, not arguments; this is a typo in the standard) will have converted incomplete array types into pointers to their element types (there are no incomplete, or complete, function types and their adjusted type is not relevant here). No other kind of incomplete types can be completed by such adjustments. The constraint in clause 6.7.5.3 applies to function definitions.

array converted to pointer

parameter adjustment in definition

The wording was changed by the response to DR #295.

C90

This wording was added to the C99 Standard to clarify possible ambiguities in the order in which requirements, in the standard, were performed on parameters that were part of a function declaration; for instance, `int f(int a[]);`.

C++

The nearest the C++ Standard comes to specifying such a rule is:

> *When a function is called, the parameters that have object type shall have completely-defined object type. [Note: this still allows a parameter to be a pointer or reference to an incomplete class type. However, it prevents a passed-by-value parameter to have an incomplete class type.]*

5.2.2p4

Other Languages

Most other languages do not treat the type of function arguments any different from other object types.

Example

```
1  extern int f_1(int p_1[]);
2  extern int f_2(int p_1[4]);     /* Number of elements is not considered. */
3
4  extern int f_3(struct foo p_1); /* foo cannot be completed. */
```

1363 **Forward references:** declarators (6.7.5), enumeration specifiers (6.7.2.2), initialization (6.7.8).

References

1. O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.
2. R. Barua. *Maps: A Compiler-Managed Memory System for Software-Exposed Architectures*. PhD thesis, M.I.T., Jan. 2000.
3. J. Burlin. Optimizing stack frame layout for embedded systems. Thesis (m.s.), Uppsala University, Information Technology Computer Science Department, Nov. 2000.
4. A. Colvin. *VIC* running out-of-core instead of running out of core*. PhD thesis, Dartmouth College, June 1999.
5. J. Engblom. Static properties of commercial embedded real-time and embedded systems. Technical Report ASTEC Technical Report 98/05, Uppsala University, Sweden, Nov. 1998.
6. D. Grunwald and B. Zorn. CUSTOMALLOC: Efficient synthesized memory allocators. Technical Report CU-CS-602-92, University of Colorado at Boulder, July 1992.
7. M. W. Howard and M. J. Kahana. Context variability and serial position effects in free recall. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 25(4):923–941, 1999.
8. M. W. Howard and M. J. Kahana. When does semantic similarity help episodic retrieval? *Journal of Memory and Language*, 46:85–98, 2002.
9. IAR Systems. *PICmicro C Compiler: Programming Guide*, iccpic-1 edition, 1998.
10. J. Ichbiah, J. Barnes, R. Firth, and M. Woodger. *Rationale for the Design of the Ada Programming Language*. Cambridge University Press, 1991.
11. M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the first International Symposium on Memory management*, pages 26–36, Mar. 1998.
12. R. Kelsey, W. Clinger, J. Rees, H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. A. IV, D. P. Friedman, E. Kohlbecker, G. L. Steele JR., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^5 report on the algorithmic language Scheme. Technical report, Feb. 1998.
13. C. Lever and D. Boreham. malloc() Performance in a multithreaded linux environment. Technical Report CITI Technical Report 00-5, University of Michigan, May 2000.
14. P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandecappelle, and P. G. K. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation and Electronic Systems*, 6(2):149–206, Apr. 2001.
15. P. R. Panda, N. D. Dutt, and A. Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):384–409, Oct. 1997.
16. P. Åke Larson and M. Krishnan. Memory allocation for long-running server applications. In *Proceedings of the First International Symposium on Memory Management ISMM'98*, pages 176–185, 1998.
17. S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *The 29^{th} Annual ACM Symposium on Principles of Programming Languages, POPL'02*, pages 140–153, Jan. 2002.
18. J. Sajaniemi. Visualizing roles of variables to novice programmers. In J. Kuljis, L. Baldwin, and R. Scoble, editors, *Fourteenth Annual Workshop of the Psychology of Programming Interest Group*, pages 111–127, June 2002.
19. M. L. Seidl and B. G. Zorn. Implementing heap-object behavior prediction efficiently and effectively. *Software–Practice and Experience*, 31(9):869–892, 2001.
20. J. Sjödin and C. von Platen. Storage allocation for embedded processors. In *Proceedings of CASES'01*, pages 15–23, Nov. 2001.
21. P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In H. G. Baker, editor, *Proceedings 1995 International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer-Verlag, Berlin, Germany, Sept. 1995.