

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.9.1 Function definitions

function definition
syntax

```
function-definition:
    declaration-specifiers declarator declaration-listopt compound-statement
declaration-list:
    declaration
    declaration-list declaration
```

Commentary

Syntactically it is not possible to tell the difference between a function definition and a function declaration until the first token after the declaration list is seen (which could be either a semicolon, an opening brace, or an identifier).

C++

The C++ Standard does not support the appearance of *declarator-list_{opt}*. Function declarations must always use prototypes. It also specifies additional syntax for *function-definition*. This syntax involves constructs that are not available in C.

Other Languages

Many other languages use a keyword to indicate that a body of code is being defined. The keywords used include **procedure**, **function**, and **subroutine**. Fortran supports the creation of statement functions within a subroutine. They essentially specify a form of parameterized expression, that can also access the objects visible at the point they are defined. They are invoked using the function call notation.

Common Implementations

An extension supported by gcc allows developers to specify attributes that a function definition possesses. For instance, some functions do not modify any objects that are not defined within their body. Making such information available to an optimizer means it does not have to make worst-case assumptions about the affects of a function call. The following declaration specifies that the function square has this attribute:

```
1 int square(int) __attribute__((const));
```

The IAR PICmicro compiler^[18] supports the **__monitor** specifier, which causes machine code to be generated that ensures the specified function is executed as an atomic entity (i.e., interrupts are disabled during its execution).

value
profiling

Recent research has shown that the value of some function parameters is the same over a large percentage of calls (for instance, based on using the SPEC95 dataset as input to gcc, in 70% of cases the value of the third parameter of calls to the function `simplify_binary_operation` was 34^[27]). A cost/benefit analysis can be used to decide whether it is worthwhile creating a specialized version, optimized for a known value of one of the parameters, of a function. The cost of such specialization is the addition, by the translator, of a check against the common value to decide whether to jump to the specialized or unspecialized version. The benefit occurs when the specialized version executes much more quickly (flow analysis making use of the known parameter value to improve the quality of machine code generated for the specialized version). To be worthwhile the overall increase in performance in the specialized version has to be greater than the decrease in performance caused by the test against the frequently occurring value on function entry.

Coding Guidelines

A program usually contains more than one function. Splitting a program up into what are sometimes called *modules* (i.e., C functions) has a long history. A variety of reasons for creating and using functions have been proposed, including the following:

- Use of appropriately sized functions (not too big and not too small, sometimes known as the *Goldilocks principle*) is believed to have various benefits (although studies validating these claims appear to be

non-existent^[12]). Park^[32] discusses how statements might be counted, and Fenton^[11] discusses using size metrics to predict the likely number of defects contained in software. While it is possible to plot measurements of various quantities against each other (e.g., a plot of defect density against function size produces a U-shaped curve, showing that very small functions and very large functions contain more defects per line of code than medium sized functions^[17]), the measurements are usually averaged over all functions in a program and causal links between the two quantities are rarely established. One problem with measurements based on the size of individual functions is that it is possible artificially change the results obtained, by splitting one function into many, or merging several into one function. For instance, reducing the number of control flow paths^[42] through a function is often seen as beneficial, because it reduces the amount of path testing that needs to be performed. A reduction in the number of paths in any function definition can be achieved by breaking it up into smaller functions. However, the reduction achieved is an artifact of the measuring process, which is based on individual definitions. The number of paths through the program has not been reduced.

- It is part of the culture of writing source code (it's what students are told they must do when learning to program, and developers who don't split their code into appropriately sized functions are often chastised for this behavior). Various program designed methodologies have specified rules for how a program should be decomposed. For instance, the structured design movement (the original Stevens, Myers, and Constantine paper was recently republished^[38]) used the idea of coupling and cohesion between statements as a means of deciding which of them belonged in a module. There have been proposals to define the concepts of coupling^[31] and cohesion^[28] in more mathematical terms. However, while this formalism enables calculations to be made by automatic tools, studies of applicability of these definitions to human readers are lacking (see Halliday and Hasan^[14] for a discussion of cohesion in English text). Concept analysis is a technique for identifying groupings of objects, in existing source code, that share common attributes. It has been used to identify possible C++ classes in C source code.^[37] coupling and cohesion
- It enables the available work to be distributed to be distributed across more than one person.
- Reducing the amount of duplicate, or very similar, code in a program. This can reduce maintenance costs by minimizing the amount of existing code that needs to be modified when changes to the behavior of a program are made (measurements of duplicate code remaining in existing programs are discussed elsewhere). 1821 duplicate code
- Splitting a program into independent modules makes it easier to change parts of it in the future. The flexibility offered by the ability to upgrade parts of a hardware system, as technology improved, rather than having to buy a new system has been shown to offer significant economic advantages for both vendors and users.^[2] Breaking a system up into modules is not enough, it is also necessary to hide information; to be exact, information that is likely to change has to be hidden within individual modules.^[33] Predicting in advance which parts of a program are likely to be changed is a difficult problem. One proposed solution is based on the economic theory of options,^[39] valuing software structures on the basis of the cost of changing them in the future.
- Reuse of software once written (e.g., libraries of functions). This issue is not considered in these coding guidelines.

A large part of comprehending a program involves comprehending the side effects generated from executing the statements contained in particular function definitions.^{1821.1} A developer's ability to predict the behavior of a program is based on their knowledge of the behavior of individual function definitions (e.g., how they modify data structures and which other functions they might call). A question often asked, by developers, about a function is "what does it do?". In many ways a function definition represents an episode of a program, which when executed in sequence with other functions tell the story that is program execution.

The coding guideline discussion on statements drew a parallel with studies of human sentence processing, statement syntax

^{1821.1} Many source code metrics are based purely on counting some attribute of the contents of function definitions.

in an effort make use of some of the findings of those studies. The discussion in of function definitions, in this coding guideline subsection, extends the use of this parallel to studies of story comprehension. It is assumed that existing human story telling and comprehension skills are something developers apply to the reading and writing of code.

A function definition is the largest unit of contiguous source code that readers might generally expect to read from start to finish.^{1821.2} Reading the story of a programs execution invariably requires readers to look at a variety of different functions which are not usually visually close to the text of the definition current being read. This behavior differs from that required to read the written form of prose stories, where the intended narrative flows sequentially through visually adjacent text (although readers might only read a few chapters at a time).

Developer memory for a function definition may depend on when they last read it. For instance, a definition that has just been read might be stored in episodic memory, while one that was read some a few days ago might be stored in semantic memory. Whether or not these differences in human memory storage mechanism, for source code information, affects developer performance is not known.

Adults have an extensive knowledge of routine activities (e.g., eating in a restaurant or going shopping) and spend a significant amount of their time performing these activities (they are very practiced at performing some of them). The *script theory* of Schank and Abelson^[36] proposes that part of a person's knowledge is organized around a large number of stereotypical situations involving activities that are often performed.

Studies of the structure of narrative stories have a long history, while the study of the structure of source code is still in its infancy. The following illustrates some of the studies that have investigated regularity and repeated elements in prose stories and source code:

- A study by Mandler and Johnson^[25] analyzed the structure of simple stories and created a grammar to describe it. In the following grammar STATE may be external (e.g., a current conditional of the world) or internal (e.g., an emotional state of mind). An EVENT is any occurrence or happening and may also be external or internal. The terminals AND, THEN, and CAUSE represent various relationships that may connect states and events.

```

STORY      -> SETTING AND EVENT_STRUCTURE
SETTING    -> STATE* (AND EVENT*) | EVENT*
STATE*     -> STATE ((AND STATE)N)
EVENT*     -> EVENT (((AND | THEN | CAUSE) EVENT)N)((AND STATE)N)
EVENT_STRUCTURE -> EPISODE ((THEN EPISODE)N)
EPISODE    -> BEGINNING CAUSE DEVELOPMENT CAUSE ENDING
BEGINNING  -> EVENT* | EPISODE
DEVELOPMENT -> SIMPLE_REACTION CAUSE ACTION | COMPLEX_REACTION CAUSE GOAL_PATH
SIMPLE_REACTION -> INTERNAL_EVENT ((CAUSE INTERNAL_EVENT)N)
ACTION     -> EVENT
COMPLEX_REACTION -> SIMPLE_REACTION CAUSE GOAL
GOAL       -> INTERNAL_STATE
GOAL_PATH  -> ATTEMPT CAUSE OUTCOME | GOAL_PATH (CAUSE GOAL_PATH)N
ATTEMPT    -> EVENT*
OUTCOME    -> EVENT* | EPISODE
ENDING     -> EVENT* (AND EMPHASIS) | EMPHASIS | EPISODE
EMPHASIS   -> STATE

```

The following is an example of a story following this grammar:^[25]

- 1 It happened that a dog had got a piece of meat
- 2 and was carrying it home in his mouth.
- 3 Now on his way home he had to cross a plank lying across a stream.
- 4 As he crossed he looked down

^{1821.2}In object-oriented languages the largest such unit might sometimes be a class.

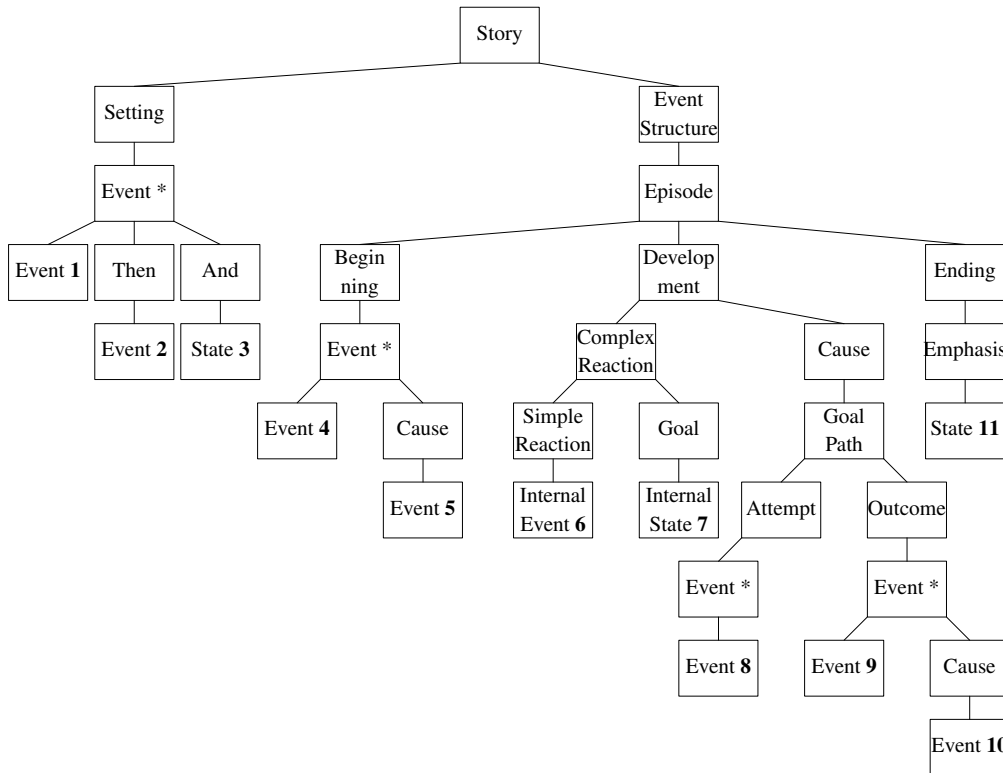


Figure 1821.1: Parse, using the Story grammar, of the tale of a dog and piece of meat. Adapted from Mandler and Johnson.^[25]

5 and saw his own shadow reflected in the water beneath.
 6 Thinking it was another dog and another piece of meat,
 7 he made up his mind to have that also.
 8 So he made a snap at the shadow,
 9 but as he opened his mouth the piece of meat fell out,
 10 dropped into the water,
 11 and was never seen again.

- Most attempts to find semantic narrative in software have been based on what are called *programming plans*. Global plans being built in a bottom-up fashion from a database of known local plans. A local plan is based on the control and data flow constructs of a group of statements (e.g., a loop over the elements of an array that performs some action when one of the elements matches a fixed value may match a linear search plan) (Hartman^[16] discusses the recognition of local plans based on concepts from a number of domains, including the application domain, mathematics, and known programming techniques; Wills^[43] describes building a database of clichés and searching for them in a graphical representation of the program). The problem of recognizing a particular program plan in source code is NP-hard^[45] ($O(S^A)$, where S is the size of the program, measured in units matched by subplans, and A is the number of subplans within the plan). Various matching heuristics have been proposed, including constraint based methods Woods.^[44] Once detected plans (clichés, concepts, schemas, templates, or some other term) have been used to locate and correct common novice programmer mistakes,^[20] recognize algorithms that can be transformed into a more efficient form (e.g., matrix multiply).^[7]

When presented with a sequence of actions peoples attempt to match them against patterns of action they are familiar with. A brief description, providing an overview, can have a significant impact on comprehension

and recall performance (this issue is discussed elsewhere).

A persons experience with the form and structure of these repeated elements seems to be used to organize the longer-term memories they form about them. The following studies illustrate the affect that peoples knowledge of the world can have on their memory for what they have read, particular with the passage of time, and their performance in interpreting sequences of related facts they are presented with:

- A study by Bower, Black, and Turner^[4] gave subjects a number of short stories describing various activities (i.e., scripts), such as visiting the dentist, attending a class lecture, going to a birthday party, etc., to read. Each story contained about 20 actions, such as looking at a dental poster, having teeth x-rayed, etc. There was then a 20 minute interval, after which they were asked to recall actions contained in the stories. The results showed that subjects around a quarter of recalled actions might be part of the script, but that were not included in the written story. Approximately seven percent of recalled actions were not in the story and would not be thought to belong to the script. A second experiment involved subjects reading a list of actions which, in the real world, would either be expected to occur in a known order or be not expected to have any order (e.g., the order of the floats in a parade). The results showed that, within ordered scripts, actions that occurred at their expected location were recalled 50% of the time while actions occurring at unexpected locations were recalled 18% of the time at that location. The recall rate for unordered scripts (i.e., the controls) was 30%.
- A study by Graesser, Woll, Kowalski, and Smith^[13] read subjects stories representing scripted activities (e.g., eating at a restaurant). The stories contained actions that varied in the degree to which they were typical of the script (e.g., Jack sat down at the table, Jack confirmed his reservation, and Jack put a pen in his pocket).

Table 1821.1: Probability of subjects recalling or recognizing typical or atypical actions present in stories read to them, at two time intervals (30 minutes and 1 week) after hearing them. Based on Graesser, Woll, Kowalski, and Smith.^[13]

Memory Test	Typical (30 mins)	Atypical (30 mins)	Typical (1 week)	Atypical (1 week)
Recall (correct)	0.34	0.32	0.21	0.04
Recall (incorrect)	0.17	0.00	0.15	0.00
Recognition (correct)	0.79	0.79	0.80	0.60
Recognition (incorrect)	0.59	0.11	0.69	0.26

The results showed (see Table 1821.1) that recall was not affected by typicality over short periods of time, but that after one week recall of atypical actions dropped significantly. Recognition (i.e., subjects were asked if a particular action occurred in the story) performance for typical vs. atypical actions was less affected by the passage of time.

- A study by Dooling and Christiaansen^[8] asked subjects to read a short biography containing 10 sentences. The only difference between the biographies read by subjects was that in some cases the name of the character was fictitious (i.e., a made up name), while in other cases it was the name of an applicable famous person. For instance, one biography described a ruthless dictator and used either the name Gerald Martin or Adolph Hitler. After a period of two days, and then after one week, subjects were given a list of 14 sentences (seven sentences that were included in the biography they had previously read and seven that were not included) and asked to specify which sentences they had previously read. To measure the impact of subjects' knowledge about the famous person, on recognition performance, some subjects were given additional information. In both cases the additional information was given to the subjects who had read the biography containing the made up name (e.g., Gerald Martin). The *before* subjects were told just before reading the biography that it was actually a description of a famous person and given that persons name (e.g., Adolph Hitler). The *after* subjects were told just before performing the recognition test (they were given one minute to think about what

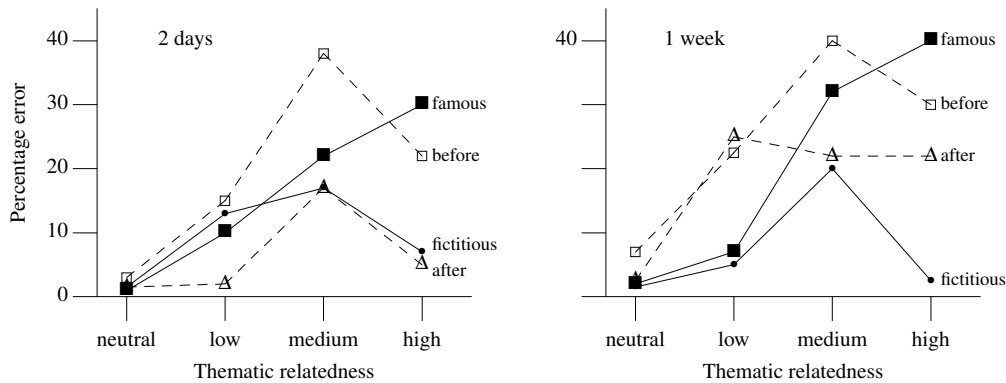


Figure 1821.2: Percentage of false-positive recognition errors for biographies having varying degrees of thematic relatedness to the famous person, in *before*, *after*, *famous*, and *fictitious* groups. Based on Dooling and Christiaansen.^[8]

they had been told) that the biography was actually a description of a famous person and given that persons name.

The results (see Figure 1821.2) are consistent with the idea that remembering is constructive. After a week subjects memory for specific information in the passage is lost. Under these conditions recognition of sentences is guided by subjects general knowledge. Variations in the time between reading the biography and identity of a famous character being revealed affected the extent to which subjects integrated this information.

- A study by Kintsch, Mandel, and Kozminsky^[22] measured the time taken to read and summarize 1,400 word stories. In the text seen by some students the order of the paragraphs (not the sentences) forming the story were randomized. The results showed that while it was not possible to distinguish between the summaries produced by subjects reading ordered vs. randomised stories, reading time for random paragraph ordering was significantly longer (9.05 minutes vs. 7.34).

Developers do not read the source code of a function definition every time they encounter a reference to it. If it has been read before they may be able to recall sufficient information about it to satisfy their immediate needs (how people trade-off the costs of recalling knowledge in their heads against obtaining knowledge from the real world is discussed elsewhere). The results of studies of story recall performance suggest that recall of events is more accurate when the events that occur correspond to reader expectations (which might be based on their knowledge of program plans, or application domain knowledge) about what events should occur.

cost/accuracy
trade-off

Developers use their knowledge of what functions do in several ways, including:

- to deduce the expected affects of performing a call to that function,
- to work out which functions need to be updated when modifying the behavior of a program, and
- to work out which functions may be affected by changes to the definition of other functions,

In all cases forgetting about an action performed by the function definition or assuming that it performs some action, when it does not it, can result in unexpected behavior during program execution.

The expectations that a reader might have about the actions performed by particular function definitions may require knowledge of the application domain, algorithms, C idioms, and development group conventions. The analysis of what actions do, or don't, meet these expectations can only be carried out (for the time being) by developers familiar with these domains. Also it might not be possible to simultaneously satisfy expectations from all of these domains, it is likely that trade-offs will need to be made.

An example of an action carried out within a function definition that would be surprising, or unexpected, might be the disabling of interrupts within a function that performs block copies of storage (perhaps special registers used by the operating system are being used to speed up the copy operation, which are restored once the copy has completed).

The parameters in a function definition are likely to be read more often than the parameters in any declaration of it. For this reason it is likely to be worth investing more in laying out the visible form of parameter declarations. The issue of declaration layout is discussed elsewhere.

Order of declarations and statement

Developers often have some flexibility in how they order declarations and statements within a function. For instance, one or more of the following patterns of usage are often seen in source:

- The declarations for all of the locally defined objects occur at the start of the function (it is rare to see a compound block opened and closed around a short sequence of statements purely to declare an object whose required lifetime and scope is limited to those statements) (see Figure ??).
- Grouping statements by the action they perform. For instance, initializing all objects at the start of a function, or performing all output at the end of a function.
- Grouping statements that access the same objects together. For instance, initializing objects close to where they are first accessed, or performing output as soon as the necessary values are known.

There are a number of reasons why grouping statements by the objects they access might offer greater benefits than other grouping algorithms, including:

- Minimizing the number of separate sequences of source that need to be cut-and-pasted when copying or reorganizing code.
- Minimizing the amount of source that intervenes between two statements containing information that needs to be integrated (into a model of function behavior) increases the probability that readers will make the correct inferences.

Recommending that related sequences of statements and/or declarations be grouped together begs the question of how to measure *related*. Given the current lack of an algorithmic measure the only alternative is to fall back on developer preferences (e.g., code reviews).

Rev 1821.1

Where possible, statements that are related to each other shall be visible close to each other in the source code.

Some coding guideline documents recommend a maximum number of lines, or statements, in a function definition. However, arbitrarily splitting a function into smaller functions purely to meet some recommended limit (experience suggests that some developers do arbitrarily split function that are considered to be too large; adherence to guideline recommendations cannot force developers to think deeply about what they are doing).

Knowing when to split a large function into smaller functions depends on developer ability to spot separate concepts. It is a matter of experience.

One practical consideration is the finite number of source lines that can be viewed on a display device at any time. Being able to view the complete source of a function removes the need for any external effort apart from eye movements (developers trading off the costs of physical and mental effort, needed to obtain information, is discussed elsewhere). It may, or may not, be possible to comprehend a function by reading its visible source without referring back to previously read material that are no longer visible on the display.

Functions definitions do not always increase in size. For instance, duplicate sequences of code, appearing in different parts of the same or different functions, may suggest the creation of a function containing a single instance of that code (this process reduces the size of existing functions).

declaration
visual layout

mixing
declarations
and statements

statements
integrating infor-
mation between

cost/accuracy
trade-off

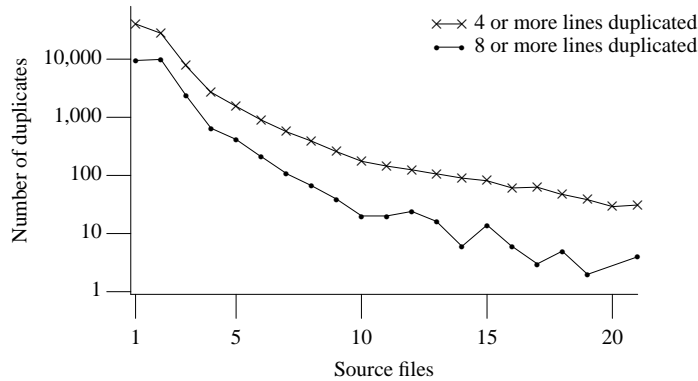


Figure 1821.3: Number of instances of duplicate physical lines, where a given duplicate line sequence is contained within a single source file or more than one source file (ignoring comments and blank lines) for sequences having at least 4 and 8 lines. Data created by processing the .c files (for each of the book's program's complete source tree) using Simian.^[35]

Duplicate source code

The same code may occur in several places of the same program (the source does not have to be character for character identical to be considered *the same*, commenting or layout may differ, some identifiers may have different spellings, or one or more lines may have been added or deleted). The terms *duplicate code* or *clones* is often used to describe instances of similar code.

duplicate code

The following are some of the reasons why duplicate code may exist in a program:

- The functionality required is very similar to that implemented by some existing source. The developer copies this existing source to use as a template for the new functionality, which may only involve a few changes to the original.
- Enhancing performance. For instance, replacing a function call, within a loop, by a copy of the body of the function.
- C's lack of support for generic types. For instance, a function defined to perform some operation on objects of type X and a duplicate definition that performs the same operation on objects having type Y. One solution here is to use call backs (e.g., the `qsort` library function has no need of any knowledge of the type of the objects being compared because it calls a developer supplied function to perform comparisons).
- Oversight. On large development efforts it is not unknown for two developers to implement functions having the same functionality.
- Coincidence. In any large program there are likely to be sequences of statements that are very similar, even though their purpose is completely different.

Duplication of source code not only increases the size of programs, it also increases the possibility of faults being created by modifications to existing code.^[19] For instance, when one sequence of statements is modified, but a duplicate sequence in another part of the program is left unmodified (the unmodified statements may not have needed modification; however, experience suggests that there is a very real likelihood that they did).

Merging duplicate sequences of statements into a single textual occurrence ensures that any changes that need to be made only need to be made in one place. Whether this one place is a function (inline or not) or a macro is a developer decision.

It is to be expected that programs will contain individual lines that are identical. Similarity only becomes noteworthy when significant amounts of code have a high degree of similarity. Judgments of what constitutes significant can vary. Detecting duplicate code can be a computationally expensive process and a number of different algorithms have been proposed, including:

- Baker^[1] built a C based tool `dup` that looked for either exact matches (ignoring white space) or parameterized matches (called *p-matches*, where the spelling of identifiers and constant literals could be different). Running `dup` on 714,479 lines of the X Window System it found 2,487 matches (representing 976 groups, each instances of code that had been copied and edited) of at least 30 lines (representing 19% of the code).

Table 1821.2: Number of clones (the same sequence of 30 or more tokens, with all identifiers treated as equivalent) detected by CCFinder between three different operating systems (Linux, FreeBSD, and NetBSD). Adapted from Kamiya, Kusumoto, and Inoue.^[21]

O/S pairs	Number of Clone Pairs	% of Lines Included in a Clone	% of Files Containing a Clone
FreeBSD/Linux	1,091	FreeBSD (0.8) Linux (0.9)	FreeBSD (3.1) Linux (4.6)
FreeBSD/NetBSD	25,621	FreeBSD (18.6) NetBSD (15.2)	FreeBSD (40.1) NetBSD (36.1)
Linux/NetBSD	1,000	Linux (0.6) NetBSD (0.6)	Linux (3.3) NetBSD (2.1)

- Code that has been cut-and-pasted may subsequently have small changes made to it. A number of researchers have attempted to detect plagiarism,^[34,41] where an attempt has been made to hide the origins of the code (e.g., in student assignments).
- It is often possible for different sequences of the same set of statements to have the same effect. However, the dependencies between statements will not be affected by any differences in ordering and comparisons based on a dependence graph will be able to find duplicates.^[23]
- Other studies have used similarity measures based on the abstract syntax tree,^[3] software metrics,^[26] and the visible source (after white space and comments had been removed).^[9]

Analysis of programs, that have been developed over a period of time, shows that they contain a surprisingly large amount of duplicate code, and that the percentage of clone functions stays approximately the same over multiple releases of a product (6–8% clones over 6 product releases as the number of function grew from 170,00 to 206,000^[24]).

Not all applications are made up of a single program. For large applications it can be worthwhile to have a number of smaller programs, each performing a specific function. Once the decision is made to have separate programs there is the real possibility that source code will be cut-and-pasted between different programs, rather than a common set of library functions created.

A study by Tonella, Antoniol, Fiutem, and Calzolari^[40] describes an application containing 4.7 M lines of code making up 402 programs (there were that many functions called `main`) linking against 815 compiled translation units (libraries). A metric taxonomy^[26] was used to detect function clones. Out of 7,277 functions; 1,016 had the same name and metric values, 609 had different names but the same metric values, and 621 functions differed by a single metric. No results were given for how many functions were actually removed from the complete application suite. They were under considerable time pressure which meant it was not possible to consider all function clones.

Use of functions invariably causes developers to consider efficiency issues. The efficiency issues associated with the call/return overhead are discussed elsewhere as are the issue of passing parameters versus using global objects.

Usage

A study of over 3,000 C functions by Harrold, Jones, and Rothermel^[15] found that the size of a functions control dependency graph was linear in the number of statements (the theoretical worst-case is quadratic in the number of statements).

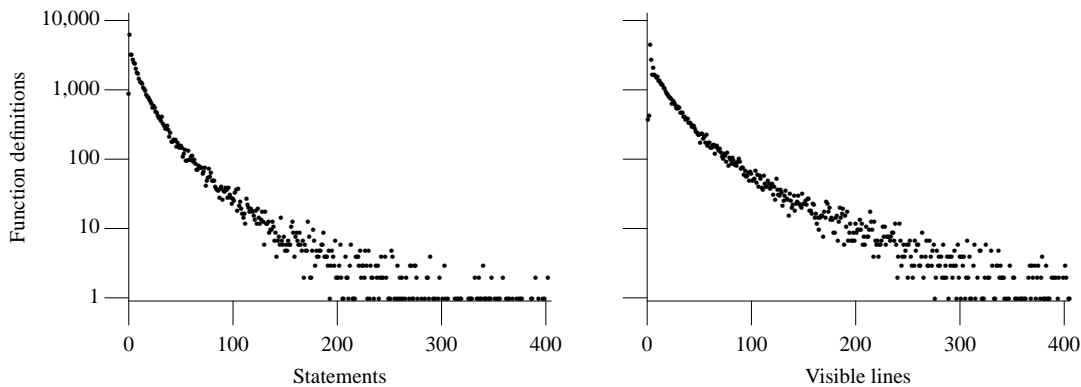


Figure 1821.4: Number of function definitions containing a given number of statements and visible source lines. Based on the translated form of this book's benchmark programs.

A study by Neamtiu, Foster, and Hicks^[30] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 81% of releases one or more existing function definitions had their argument signature changed, while one or more function definitions had their return type changed in 42% of releases and one or more function definitions had their name changed in 49% of releases.^[29]

Table 1821.3: Static count of number of functions and uncalled functions in SPECint95. Adapted from Cheng.^[6]

Benchmark	Lines of Code	Number of Functions	Uncalled Functions	Benchmark	Lines of Code	Number of Functions	Uncalled Functions
008.espresso	14,838	361	46	126.gcc	205,583	2,019	187
023.eqntott	12,053	62	2	130.li	7,597	357	1
072.sc	8,639	179	8	132.jpeg	29,290	477	16
085.cc1	90,857	1,452	51	134.perl	26,874	276	13
124.m88ksim	19,092	252	13	147.vortex	67,205	923	295

How many instructions are executed, on average, in a function definition? It will depend on the characteristics of the translator and host processor (see Table 1821.4).

translation
technology

Table 1821.4: Mean number of instructions executed per function invocation. Based on Calder, Grunwald, and Zorn.^[5]

Program	Mean	Leaf	Non-Leaf	Program	Mean	Leaf	Non-Leaf
burg	61.6	30.6	142.8	eqntott	386.8	402.8	294.2
ditroff	58.6	72.3	56.3	espresso	244.9	151.3	526.5
tex	173.2	44.3	205.4	gcc	96.4	30.1	123.5
xfig	61.9	38.6	74.8	li	42.5	31.9	44.2
xtex	114.9	93.9	136.5	sc	71.1	49.4	80.1
compress	368.4	1,360.2	367.5	Mean	152.8	209.6	186.5

Table 1821.5 gives a breakdown of the overall control flow characteristics of function bodies. One explanation for the larger number of SPECint benchmark functions containing iteration statements is that these programs were selected on the basis of primarily being cpu bound. The only practical way of using lots of cpu time is to iterate and hence this benchmark is biased in favour functions that iterate a lot.

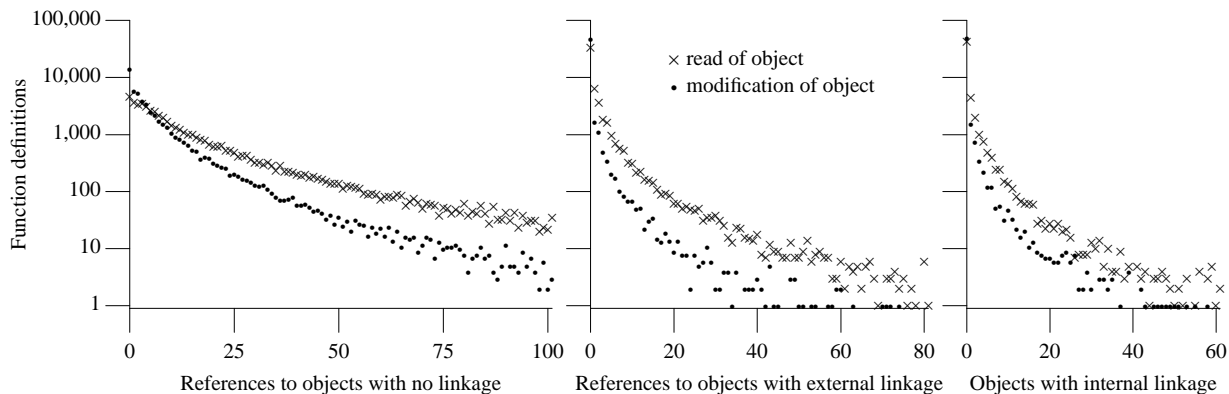


Figure 1821.5: Number of function definitions containing a given number of references (i.e., an access or modification) to all objects, having various kinds of linkage. Based on the translated form of this book’s benchmark programs.

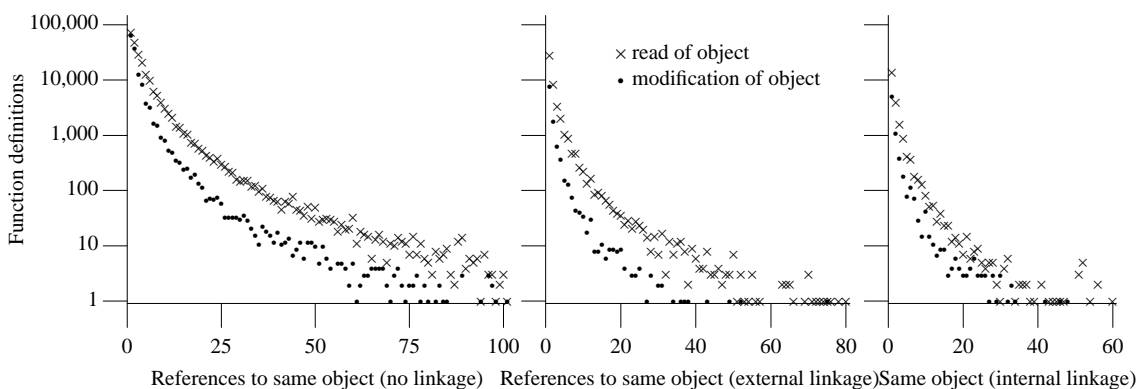


Figure 1821.6: Number of function definitions containing a given number of references (i.e., an access or modification) to the same object, having various kinds of linkage. Based on the translated form of this book’s benchmark programs.

Table 1821.5: Contents of function bodies (as a percentage of all bodies) for embedded .c source,^[10] SPECint95, and the translated form of this book’s benchmark programs.

	Embedded	SPECint95	Book benchmarks
Trivial (one basic block)	32.7	16.2	57.1
Non-looping	47.9	48.1	18.1
Looping	19.4	35.7	24.8

Usage information on the number of objects defined within a function definition is given elsewhere (see Figure ??).

Constraints

The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.¹³⁸⁾ 1822

Commentary

The syntax for function definitions requires a declarator, which means that the following:

```

1  int x
2  { }
```

is syntactically valid. However, it violates the above constraint. In the following declarations:

```
1 typedef int F(void);
2 F f { /* ... */ }
```

the declarator `f` does not syntactically specify a function type (see the footnote for further discussion).

1830 [footnote](#)
138

C++

The C++ Standard specifies the syntax (which avoids the need for a footnote like that given in the C Standard):

The declarator in a function-definition shall have the form

8.4p1

D1 (*parameter-declaration-clause*) *cv-qualifier-seq_{opt}* *exception-specification_{opt}*

Common Implementations

Many translators use the requirements specified in this constraint to simplify the language grammar they need to process. Consequently the diagnostic message they issue for a violation of this constraint often refers to a violation of syntax.

1823 The return type of a function shall be `void` or an object type other than array type.

Commentary

This wording is slightly different from that specified for function declarators, but the set of return types supported is the same. While function types are not included, pointers to functions are object types and are therefore permitted.

function
definition
return type

function
declarator return
type

Array types are converted to pointer types in many contexts. Given the occurrence of these implicit conversions, writing an expression (e.g., in a `return` statement) that has an array type is relatively involved. A special case dealing with objects having an array type in a `return` statement would add complications to the language for little obvious benefit (it is possible to declare a function to return a structure type, which has a member having an array type).

array
converted to
pointer

C++

Types shall not be defined in return or parameter types.

8.3.5p6

The following example would cause a C++ translator to issue a diagnostic.

```
1 enum E {E1, E2} f (void) /* does not change the conformance status of program */
2                               // ill-formed
3 {
4   return E1;
5 }
```

Other Languages

A number of languages support functions returning array types and some languages (e.g., those in the functional family of languages) support functions returning function types or even partially evaluated functions. However, some languages further restrict the return type to being a scalar type.

Example

```
1 int (*g)(void) /* Constraint violation. */
2 { /* ... */ }
3
4 int (*h(void))[2] /* Constraint violation. */
5 { /* ... * }
```

Usage

Usage information on function return types in the .c files is given elsewhere (see Table ??).

Table 1823.1: Occurrence of function return types (as a percentage of all return types; signedness and number of bits appearing in value representation form) appearing in the source of embedded applications (5,597 function definitions) and the SPECint95 benchmark (2,713 function definitions). A likely explanation of the greater use of type **void** is the perceived performance issues associated with returning values via the stack causing developers to return values via objects at file scope. Adapted from Engblom.^[10]

Type/Representation	Embedded	SPECint95	Type/Representation	Embedded	SPECint95
void	59.4	31.2	ptr-to . . .	2.0	17.1
unsigned 32 bit	0.5	2.2	signed 32 bit	0.3	48.4
unsigned 16 bit	3.3	0.0	signed 16 bit	1.6	0.2
unsigned 8 bit	31.6	0.5	signed 8 bit	0.8	0.0

The storage-class specifier, if any, in the declaration specifiers shall be either **extern** or **static**.

1824

Commentary

In the context of function declarations, storage-class specifiers are used to specify linkage. The two chosen where **extern** and **static** (representing external and internal linkage respectively), making the appearance of any other storage-class specifier meaningless.

C++

7.1.1p2 *The **auto** or **register** specifiers can be applied only to names of objects declared in a block (6.3) or to function parameters (8.4).*

A C++ translator is not required to issue a diagnostic if these storage-class specifiers appear in other contexts. Source developed using a C++ translator may contain constraint violations if processed by a C translator.

Common Implementations

One possible interpretation that could be given to the **register** storage-class appearing in a function definition, is as a hint to translators that the machine code for the function body be kept in a processor's instruction cache. However, support for explicit program control of the cache is only just starting to appear in commercially available processors.

cache

Coding Guidelines

The coding guideline issues associated with function declarations that include the storage-class specifier **static**, **extern**, or no storage-class are discussed elsewhere.

static
internal linkage
extern
identifier
linkage same as
prior declaration
function
no storage-class

If the declarator includes a parameter type list, the declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting of a single parameter of type **void**, in which case there shall not be an identifier.

1825

Commentary

The syntax for a parameter type list in a function definition is the same as that used for function declarations, where no identifiers need be specified. However, in the case of a function identifiers are required, otherwise there is no mechanism for accessing any argument value passed in that position (use of the ellipsis does not involve the explicit specification of type information)

ellipsis
supplies no
information

C++

An identifier can optionally be provided as a parameter name; if present in a function definition (8.4), it names a parameter (sometimes called “formal argument”). [Note: in particular, parameter names are also optional in function definitions and . . .

[Note: unused parameters need not be named. For example,

8.4p5

```
void print(int a, int)
{
    printf("a = %d\n", a);
}
```

—end note]

Source developed using a C++ translator may contain unnamed parameters, which will cause a constraint violation if processed by a C translator.

1826 No declaration list shall follow.

Commentary

A declaration list can only follow when the form of the function declarator is the old style.

1827 If the declarator includes an identifier list, each declaration in the declaration list shall have at least one declarator, those declarators shall declare only identifiers from the identifier list, and every identifier in the identifier list shall be declared.

identifier list
declare at least
one declarator

Commentary

The declaration list declares the identifiers that appear in the parameter list and only those identifiers. Standard support for *old style* function definitions was needed because of the existence of a large body source code containing them. This constraint does not prohibit some declarations of doubtful utility. For instance:

```
1 void f_1(p)
2 enum {x, y /* x & y appear in a declaration-specifier, not in a declarator list. */
3         } p;
4 { /* ... */ }
5
6 void f_2(p)
7 struct {
8     int z; /* z is declared, but is not in the declarator list. */
9     } *p;
10 { /* ... */ }
```

C90

The requirement that every identifier in the identifier list shall be declared is new in C99. In C90 undeclared identifiers defaulted to having type **int**.

Source files that translated without a diagnostic being issued by a C90 translator may now result in a diagnostic being generated by a C99 translator.

C++

The declaration list form of function definitions is not supported in C++.

Common Implementations

It is likely that many C99 implementations will offer a C90 compatibility option, that will successfully translate source files containing functions whose parameter definition includes identifiers that are not explicitly declared in the declaration list.

Example

Fortran allows function parameters to be referred to before they are defined and it is established practice to declare an array parameter before the length specified in its array bounds. Translating such Fortran source into C required either that the parameter order be reversed, or old style definitions be used.

```

1  extern void f_proto(int, double [*][*]);
2  extern void f_old();
3
4  /*
5   * C requires identifiers to be declared before they are referenced.
6   */
7  void f_proto(int p_length, double p_1[p_length][p_length])
8  { /* ... */ }
9
10 void f_old(p_1, p_length) /* Parameter order follows Fortran conventions. */
11 int p_length;
12 double p_1[p_length][p_length];
13 { /* ... */ }
14
15 /*
16  * Hanging on to a bit more type information...
17  */
18 void f_alternative(double *p_1, int p_length)
19 {
20  double (*loc_p_1)[p_length] = (double (*)[p_length])p_1;
21 }

```

An identifier declared as a typedef name shall not be redeclared as a parameter.

1828

Commentary

Traditionally C source has been syntactically processed using a parser that only examined the next token in the input stream (i.e., the grammar can be processed using a LALR(1) tool, such as yacc and bison). This C constraint means that translators are not required to process the following code:

```

1  typedef int I;
2
3  int f(I) /* Constraint violation. */
4  int I;
5  { /* ... */ }

```

which requires more than one token lookahead to answer the question: is `f` an old style function talking a parameter called `I`, or is `f` a declaration of a prototype taking a parameter of type `int`, with no identifier name given?

C++

The form of function definition that this requirement applies to is not supported in C++.

function dec-
laration list
storage-class
specifier

The declarations in the declaration list shall contain no storage-class specifier other than **register** and no initializations. 1829

Commentary

Parameters have automatic storage duration and are initialized by the value of the corresponding argument.

C++

The form of function definition that this requirement applies to (i.e., old-style) is not supported in C++.

parameter
automatic stor-
age duration
function call
preparing for

Other Languages

Those languages (e.g., Ada) that support the use of default values on parameters, as a method of providing default values, usually require that they appear in the function declaration rather than its definition (so they are visible at the points in the source where calls to them occur).

1830 138) The intent is that the type category in a function definition cannot be inherited from a typedef:

```
typedef int F(void);           // type F is "function with no parameters
                              //                               returning int"
F f, g;                       // f and g both have type compatible with F
F f { /* ... */ }            // WRONG: syntax/constraint error
F g() { /* ... */ }          // WRONG: declares that g returns a function
int f(void) { /* ... */ }    // RIGHT: f has type compatible with F
int g() { /* ... */ }        // RIGHT: g has type compatible with F
F *e(void) { /* ... */ }     // e returns a pointer to a function
F *((e))(void) { /* ... */ } // same: parentheses irrelevant
int (*fp)(void);             // fp points to a function that has type F
F *Fp;                       // Fp points to a function that has type F
```

footnote
138

Commentary

This requirement significantly simplifies the creation of grammars that can be processed by commonly available parser generators without any syntactic production rule conflicts occurring.

C++

The C++ Standard specifies this as a requirement in the body of the standard (8.3.5p7).

Semantics

1831 The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters.

Commentary

Saying in words what is specified in the syntax. The return type is given by some of the *declaration-specifiers* and parts of the *declarator*.

C++

The C++ Standard does not explicitly make this association about function definitions (8.4).

Usage

Information on argument types is given elsewhere (see Table ??).

Table 1831.1: Occurrence of parameter types in function definitions (as a percentage of the parameters in all function definitions). Based on the translated form of this book's benchmark programs.

Type	%	Type	%	Type	%	Type	%
struct *	44.4	void *	3.4	long	1.6	struct * *	1.2
int	14.7	union *	3.1	int *	1.5	enum	1.2
other-types	6.8	unsigned long	2.7	unsigned char *	1.4	const char *	1.1
unsigned int	5.1	unsigned int *	2.0	char * *	1.3	long *	1.0
char *	4.7	unsigned char	1.6	unsigned short	1.2		

1832 If the declarator includes a parameter type list, the list also specifies the types of all the parameters;

Commentary

Each parameter has the type specified in this list, not the type of the corresponding parameter of any composite type (that may have been created because of the occurrence of previous declarations).

C++

If the parameter list is empty the C++ Standard defines the function as taking no arguments (8.3.5p2).

Example

```

1  extern int glob;
2  extern void f(int);
3
4  void f(const int p)
5  { /* p has type const int in this function body. */ }
6
7  void g(void)
8  {
9  f(glob); /* Composite type of f is function taking a parameter of type int. */
10 }

```

function prototype such a declarator also serves as a function prototype for later calls to the same function in the same translation unit. 1833

Commentary

composite type

If there are any previous declarations of the same function (perhaps from an included header), their types along with the function prototype of the function definition are used to form a composite type. It is this composite type that is used for later calls.

If the declarator includes an identifier list,¹⁸²⁷ the types of the parameters shall be declared in a following declaration list. 1834

Commentary

identifier list declare at least one declarator

This requirement on the program (also covered by a constraint) associates each identifier in the identifier list with the type of the corresponding identifier declared in the declaration list.

C++

The identifier list form of function definition is not supported in C++.

parameter type adjusted In either case, the type of each parameter is adjusted as described in 6.7.5.3 for a parameter type list; 1835

Commentary

array type adjust to pointer to function type adjust to pointer to

The ordering of this and the following C sentence is important. Both array and function types are converted to pointers to the appropriate respective types before the requirement on being an object type applies.

the resulting type shall be an object type. 1836

Commentary

parameter parameter linkage object type complete by end

This sentence clarifies the relative order in which adjustment of parameter types and checking for an object type occurs. Requirements in other parts of the standard specify that a parameter (which is an object with no linkage) have a complete type by the end of its declarator; so declaring a parameter to have an incomplete type would be a constraint violation.

C++

The only difference, in parameter types, between a function declaration and a function definition specified by the C++ Standard is:

The type of a parameter or the return type for a function declaration that is not a definition may be an incomplete class type.

- 1837 If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behavior is undefined.

Commentary

The behavior is undefined if there are either fewer arguments or more arguments, passed in a call, than parameters in the function definition. This C sentence is not specific to the form of function definition (i.e., it could be old-style or prototype) and it is not necessary to access any of the parameters within the function definition. Simply calling the function is sufficient to cause the undefined behavior.

Developers did define functions accepting variable numbers of arguments prior to the availability of prototypes and the ellipsis notation (first specified in the C90 Standard). One technique used to access the additional argument values was to take the address of the last declared parameter and perform pointer arithmetic on it (knowledge of stack layout, such as order of arguments pushed and their alignment, was required to make it point at where the additional arguments were placed).

C++

This C situation cannot occur in C++ because it relies on the old-style of function declaration, which is not supported in C++.

Common Implementations

Because of the unknown, assumed to be significant, amount of existing code that passes variable number of arguments to functions defined using the old-style notation, implementations invariably play safe and do not use any special argument passing conventions (i.e., they are invariably passed on the stack) for such definitions. In the case of function defined using prototype notation, implementations invariably assume that the number of arguments passed equals the number of parameters defined. If these numbers differ the results can be completely unpredictable.

Coding Guidelines

This usage may occur in existing code. The cost/benefit issues involved in modifying existing code are discussed elsewhere. If the guideline recommendation specifying the use of prototypes is followed the usage described by this C sentence will not occur in newly written code.

external
declaration
syntax
?? function
declaration
use prototype

- 1838 Each parameter has automatic storage duration.

Commentary

This specification can be deduced from other wording in the standard, i.e., a parameter is an object, whose identifier has no linkage and its declaration may not include the storage-class specifier **static**, therefore it has automatic storage duration.

Coding Guidelines

Some coding guideline documents recommend that function parameters (which are defined to have block scope) be treated as being different from other block scope objects. The recommended differences in treatment arise from conceptual ideas about what parameters represent. For instance, considering them as input only values (which implies that they should not be modified in the function body). The rationale for this view of parameters is often derived from other languages where a pass by address method of argument passing is supported (in this case modifications of a parameters value will affect the value of the corresponding object passed as the argument). There is no obvious benefit in having such a guideline recommendation in C.

parameter
automatic stor-
age duration

parameter
parameter
linkage
parameter
storage-class
1829 function dec-
laration list
storage-class
specifier
automatic
storage duration
1839 parameter
scope begins at

- 1839 Its identifier is an lvalue, which is in effect declared at the head of the compound statement that constitutes the function body (and therefore cannot be redeclared in the function body except in an enclosed block).

parameter
scope begins at

Commentary

With the introduction of support for variable length arrays in C99, this description (which was in a footnote in the C90 Standard), is no longer completely accurate.

```
1 void f(int p_length, double p_1[p_length][p_length])
2 { /* ... */ }
```

C++

The C++ Standard does not explicitly specify the fact that this identifier is an lvalue. However, it can be deduced from clauses 3.10p1 and 3.10p2.

3.3.2p2 *The potential scope of a function parameter name in a function definition (8.4) begins at its point of declaration. If the function has a function try-block the potential scope of a parameter ends at the end of the last associated handler, else it ends at the end of the outermost block of the function definition. A parameter name shall not be redeclared in the outermost block of the function definition nor in the outermost block of any handler associated with a function try-block.*

The layout of the storage for parameters is unspecified.

1840

Commentary

In the past there existed a relatively common practice of accessing the values of different parameters by calculating their addresses using pointer arithmetic (this usage causes undefined behavior because pointer arithmetic is only guaranteed to deliver defined results while it remains within the same pointed-to object, or one past the end of the same object). This specification of behavior (there is no equivalent wording for other objects) is intended to explicitly point out to developers that the standard does not guarantee any particular storage layout for parameters.

pointer
arithmetic
defined if
same object
storage
layout

C++

The C++ Standard does not explicitly specify any storage layout behavior for parameters.

Other Languages

Those languages (e.g., Ada and CHILL) that do provide a mechanism for specifying how objects are laid out in storage do not usually provide support for specifying the relative layout of parameters. BCPL explicitly specifies that the parameters form a contiguous array. However, it is implementation-defined whether the first parameter can be accessed using the lowest or highest subscript.

Common Implementations

Most implementations do the obvious and parameters are laid out in a contiguous area of storage, with addresses either allocated high to low, or low to high. However, some implementations speed up parameter passing by using registers and may not even allocate storage if it is not needed. The alignment of storage used for parameters can be influenced by factors that do not affect object storage layout in other contexts. For instance, the processor may support instructions that manipulate the stack in fixed-sized units (e.g., the natural size of type `int`) and the choice of alignment used for various types is driven by the requirements of these instructions (which do not affect alignment choice in other contexts). Storage allocation issues for parameters are also discussed elsewhere.

alignment
storage
layout

stack

Coding Guidelines

Making use of storage layout information requires that other unspecified or undefined behavior be used, for instance incrementing pointer values so that they no longer point within the original object. The guideline recommendations dealing with making use of representation information is applicable.

pointer
arithmetic
undefined
representation
information
using

- 1841 On entry to the function, the size expressions of each variably modified parameter are evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment.

function entry
parameter
type evaluated

Commentary

This requirement, on implementations, creates a dependency between the evaluation of an argument whose corresponding parameter is used in the evaluation of the size expression of a parameter having a variably modified type and the evaluation of the size expression of that variably modified parameter. For instance, the required evaluation order in the following call to `f`:

```

1  extern int glob_1,
2      glob_2[10][2];
3
4  void f(int n, int a[n][2])
5  { /* ... */ }
6
7  void g(void)
8  {
9  f(glob_1, glob_2);
10 }
```

is to assign the argument `glob_1`, in the call to `f`, to the parameter `n`, evaluate the type of `a`, and then assign the argument `glob_2` to the parameter `a`.

The order of evaluation of the size expressions of parameters having variably modified types is unspecified. For instance, a strictly conforming program has to make worst-case assumptions about the number of elements required in the arrays passed as arguments to the functions `f_1` or `f_2` in the follow example:

```

1  extern int glob;
2
3  int g(void)
4  {
5  return glob++;
6  }
7
8  /*
9  * The order of evaluation of the following variably modified
10 * parameters is not defined, but there is a sequence point
11 * between the two modifications of glob.
12 */
13 void f_1(int a_1[const g()], int a_2[const g()])
14 { /* ... */ }
15
16 /*
17 * The evaluation of the following variably modified parameters causes
18 * undefined behavior, because the same object is modifies more than
19 * once between sequence points (a full declarator is a sequence point.
20 */
21 void f_2(int a_1[const glob++], int a_2[const glob++])
22 { /* ... */ }
```

If a function prototype is visible at the point of call the values of the arguments will already have been converted to the types of the parameters, before being passed. There is no actual assignment performed on function entry.

If the only visible declaration is an old style function declaration the arguments will be subject to the default argument promotions, which means they may not have the same type as the parameter. When the type of the parameter is not compatible with its promoted type it is necessary for the value of the argument to be converted, as if by assignment, to the type of the parameter.

default ar-
gument
promotions

```

1  #include <stdio.h>
2
3  int f(p)
4  unsigned char p;
5  {
6  /* Implementations acts as if argument passed is assigned to p here. */
7  printf("argument passed = %d\n", p);
8  }
9
10 int main(void)
11 {
12 /* Output from first three calls should be the same as from the second three. */
13 f(0x42);           f(0x0142);           f(0x2142);
14 f((unsigned char)0x42); f((unsigned char)0x0142); f((unsigned char)0x2142);
15 }

```

function call
preparing for

The standard specifies elsewhere that each parameter is assigned the value of the corresponding argument.

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

Common Implementations

Some implementations can optimize away the need to perform an assignment because of the way that they load values from storage, into registers. For instance, a load byte instruction may clear all of the other bits in the register as well as loading a byte into it, removing the need to convert the value passed as an argument (any other, more significant, bits are always ignored).

```

1  #include <stdio.h>
2
3  int f(p)
4  unsigned char p;
5  {
6  /*
7  * Accessing the storage allocated to p plus some bytes immediately after it may
8  * access the value of the argument actually passed (provided the implementation
9  * does not perform the implicit assignment). However, such an access also
10 * makes use of undefined behavior, so in theory any result could be returned.
11 */
12 printf("parameter value = %d, argument passed may have been = %d\n",
13         p,                               *(int *)&p);
14 }

```

If the argument value is not representable in the parameter type the behavior of most implementations is the same as that for the same situation in an assignment.

Coding Guidelines

Support for variably modified types is new in C99 and the issue of side effects in their evaluation is one of developer education as well as potentially being coding guideline related. Variable length array declarations can generate side effects, a known problem area. The coding guideline issues for full declarators are discussed elsewhere.

object
initializer eval-
uated when

(Array expressions and function designators as arguments were converted to pointers before the call.)

1842

Commentary

These conversions mirror those that occur for the parameter types.

array
converted
to pointer
function
designator
converted to type
parameter-1835

- 1843 After all parameters have been assigned, the compound statement that constitutes the body of the function definition is executed.

Commentary

That is, after all the parameters have been assigned the value of the corresponding arguments.

The compound body associated with a function definition is treated the same way as a compound body within a nested inner block.

This C statement assumes that the execution environment has sufficient resources to allocate storage for the objects defined in the called function. The C Standard does not provide any mechanism to check whether there are sufficient resources available to call and execute the designated function. Neither does it say anything about what might happen if there are insufficient resources to execute the function definition.

C90

The C90 Standard does not explicitly specify this behavior.

C++

The C++ Standard does not explicitly specify this behavior.

- 1844 If the `}` that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.

function termination reaching }

Commentary

This case deals with an implicit **return** statement without an expression (omitting an expression from an explicit **return** statement in a function returning an object type is a constraint violation). There is a body of existing code that omits an explicit **return** statement because it is known that the caller does not access a returned value. Rather than specifying this case as a constraint violation, requiring translators to issue a diagnostic, rendering a (allegedly large) body of existing code non-conforming the Committee specified undefined behavior (this exact wording also appears in the C90 Standard).

return without expression

C++

*Flowing off the end of a function is equivalent to a **return** with no value; this results in undefined behavior in a value-returning function.*

6.6.3p2

The C++ Standard does not require that the caller use the value returned for the behavior to be undefined; this behavior can be deduced without any knowledge of the caller.

```

1  int f(int p_1)
2  {
3  if (p_1 > 10)
4      return 2;
5  }
6
7  void g(void)
8  {
9  int loc = f(11);
10
11 f(2); /* does not change the conformance status of the program */
12     // undefined behavior
13 }
```

Other Languages

This behavior is common to many programming languages.

Common Implementations

The usual behavior is for the value of the function call to be whatever happens to be held in the storage location used to hold the return value, when the terminating `}` is reached.

Coding Guidelines

The `}` that terminates a function is reached for one of two reasons:

1. it was not intended to occur and the author of the function body has made a mistake. These coding guidelines are not intended to recommend against the use of constructs that are obviously faults,
2. it is intended to occur in certain situations. This intended usage creates a dependency between the calls that do not access the value returned and the control flow in the function definition that results in no defined value being returned in some situations.

Adhering to a guideline recommending that the terminating `}` in a function not be reached may require adding a **return** statement, with an associated expression. Adding such a statement raises various questions, including the following:

- What value should be returned by this statement? On the basis that the statement is never intended to be executed any value will suffice. A case can be made for zero in that this value is more likely to be within the bounds of expected return values and may not have any significant affect on subsequent execution, and a case can be made for returning a value that is likely to have a noticeable affect on subsequent program execution. Returning a fixed value is at least more likely to ensure consistent behavior.
- How will subsequent readers of the function source treat this **return** statement? Without additional information they are likely to assume the value (e.g., a comment or giving it a symbolic name such as `DUMMY_VALUE`) was intended to be returned. The presence of what is essentially a **return** statement has possible costs as well as possible benefits.

Without any obvious cost/benefit for using a **return** statement no guideline recommendation is made here. However, guidelines having other aims (e.g., defensive programming) may recommend the presence of such a statement.

Usage

In the translated source of this book's benchmark programs 0.7% of function definitions contained both `return;` (or the flow of control reached the terminating `}`) and `return expr;`.

EXAMPLE 1 In the following:

```
extern int max(int a, int b)
{
    return a > b ? a : b;
}
```

extern is the storage-class specifier and **int** is the type specifier; **max(int a, int b)** is the function declarator; and

```
{ return a > b ? a : b; }
```

is the function body. The following similar definition uses the identifier-list form for the parameter declarations:

```
extern int max(a, b)
int a, b;
{
    return a > b ? a : b;
}
```

Here **int a, b;** is the declaration list for the parameters. The difference between these two definitions is that the first form acts as a prototype declaration that forces conversion of the arguments of subsequent calls to the function, whereas the second form does not.

Commentary

These two forms of parameter declaration are similar because the type of `a` and `b` is compatible with its promoted types

C++

The second definition of `max` uses a form of function definition that is not supported in C++.

1846 139) See “future language directions” (6.11.7).

footnote
139

1847 EXAMPLE 2 To pass one function to another, one might say

```
int f(void);
/* ... */
g(f);
```

Then the definition of `g` might read

```
void g(int (*funcp)(void))
{
    /* ... */
    (*funcp)(); /* or funcp() ... */
}
```

or, equivalently,

```
void g(int func(void))
{
    /* ... */
    func(); /* or (*func)() ... */
}
```

Other Languages

The second declaration of `g` is the form often used by other languages.

Coding Guidelines

While the second form of declaration of `g` may appear more intuitive to many developers, this may be because they have relatively little experience using function pointers. The first declaration of `g` uses the form needed to declare an object as a pointer to a function type and developers that regularly use pointer to function types will be practiced in reading it.

References

1. B. S. Baker. On finding duplication and near-duplication in large software systems. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Second Working Conference on Reverse Engineering*, pages 86–95, Los Alamitos, California, July 1995. IEEE Computer Society Press.
2. C. Y. Baldwin and K. B. Clark. *Design Rules. Volume 1. The Power of Modularity*. The MIT Press, 2000.
3. I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In T. M. Koshgoftaar and K. Bennett, editors, *Proceedings of the International Conference on Software Maintenance (ICSM’98)*, pages 368–378. IEEE Computer Society Press, 1998.
4. G. H. Bower, J. B. Black, and T. J. Turner. Scripts in memory for text. *Cognitive Psychology*, 11:177–220, 1979.
5. B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1995.
6. B.-C. Cheng. *Compile-Time Memory Disambiguation for C Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.
7. B. Di Martino. Specification and automatic recognition of algorithmic concepts within programs. Technical Report Technical Report 97-14, University of Vienna, Nov. 1997.
8. D. J. Dooling and R. E. Christiaansen. Episodic and semantic aspects of memory for prose. *Journal of Experimental Psychology: Human Learning and Memory*, 3(4):428–436, 1977.
9. S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicate code. In H. Yang and L. White, editors, *Proceedings International Conference on Software Maintenance ICSM’99*, pages 109–119. IEEE Computer Society Press, Sept. 1999.
10. J. Engblom. Why SpecInt95 should not be used to benchmark embedded systems tools. *ACM SIGPLAN Notices*, 34(7):96–103, July 1999.
11. N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(3):675–689, 1999.
12. M. Frappier, S. Matwin, and A. Mili. Software metrics for predicting maintainability. Technical Report Technical Memorandum 2, Canadian Space Agency, Jan. 1994.
13. A. C. Graesser, S. B. Woll, D. J. Kowalski, and D. A. Smith. Memory for typical and atypical actions in scripted activities. *Journal of Experimental Psychology: Human Learning and Memory*, 6(5):503–515, 1980.
14. M. A. K. Halliday and R. Hasan. *Cohesion in English*. Pearson Education Limited, 1976.
15. M. J. Harrold, J. A. Jones, and G. Rothermel. Empirical studies of control dependence graph size for C. *Empirical Software Engineering Journal*, 3(2):203–211, Mar. 1998.
16. J. Hartman. *Automatic control understanding for natural programs*. PhD thesis, University of Texas at Austin, May 1991.
17. L. Hatton. Reexamining the fault density-component size connection. *IEEE Software*, 14(2):89–97, Mar. 1997.
18. IAR Systems. *PICmicro C Compiler: Programming Guide*, icpic-1 edition, 1998.
19. L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering*, pages 55–64, Sept. 2007.
20. W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Morgan Kaufmann Publishers, Inc, 1986.
21. T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
22. W. Kintsch, T. S. Mandel, and E. Kozminsky. Summarizing scrambled stories. *Memory & Cognition*, 5(5):547–552, 1977.
23. J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering (WCRE 2001)*, pages 301–309, Oct. 2001.
24. B. Laguë, D. Proulx, E. M. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings: 1997 International Conference on Software Maintenance*, pages 314–321. IEEE Computer Society Press, 1997.
25. J. M. Mandler and N. S. Johnson. Remembrance of things parsed: Story structure and recall. *Cognitive Psychology*, 9:111–151, 1977.
26. J. Mayrand, C. Leblanc, and E. M. Merlo. Automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–254. IEEE Computer Society Press, Nov. 1996.
27. R. Muth, S. Watterson, and S. Debray. Code specialization based on value profiles. In *Proceedings 7th International Static Analysis Symposium (SAS 2000)*, volume 1824 of *LNCIS*, pages 340–359. Springer, 2000.
28. J. Nandigam. *A Measure for Module Cohesion*. PhD thesis, University of Louisiana, May 1995.
29. I. Neamtiu. Detailed break-down of general data provided in paper^[30] kindly supplied by first author. Jan. 2008.
30. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, May 2005.
31. A. J. Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *The Journal of Systems and Software*, 20(3):295–308, 1993.
32. R. E. Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR-20, Software Engineering Institute, Sept. 1992.
33. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
34. L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical Report Technical Report 2000-1, Universität Karlsruhe, Fakultät für Informatik, 2000.
35. RedHill Consulting, Pty. Simian - similarity analyser, v 2.0.2. www.redhillconsulting.com.au, 2004.
36. R. C. Shank and R. P. Abelson. *Scripts, Plans, Goals, and Understanding: An Enquiry into Human Knowledge Structures*. Lawrence Erlbaum Associates, 1977.
37. M. B. Siff. *Techniques for software renovation*. PhD thesis, University of Wisconsin–Madison, 1998.
38. W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 38(2-3):231–256, 1999.

39. K. Sullivan, Y. Cai, B. Hallen, and W. G. Griswold. The structure and value of modularity in software design. Technical Report Technical Report CS-2001-13, University of Virginia, 2001.
40. P. Tonella, G. Antoniol, R. Fiutem, and F. Calzolari. Reverse engineering 4.7 million lines of code. *Software—Practice and Experience*, 30(2):129–150, Feb. 2000.
41. K. L. Verco and M. J. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In *Proceedings of First Australian Conference on Computer Science Education*, 1996.
42. A. H. Watson and T. J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. Technical Report NIST Special Publication 500-235, National Institute of Standards and Technology (NIST), Sept. 1996.
43. L. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT, July 1992.
44. S. Woods. *A Method of Program Understanding using Constraint Satisfaction for Software Reverse Engineering*. PhD thesis, University of Waterloo, 1996.
45. S. Woods and Q. Yang. Approaching the program understanding problem: Analysis and A heuristic solution. In *Proceedings of the 18th International Conference on Software Engineering (ICSE '96)*, Mar. 1996.