The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

Copyright ©2002-2008 Derek M. Jones. All rights reserved.

6.8 Statements and blocks

statement syntax

1707

```
statement:
        labeled-statement
        compound-statement
        expression-statement
        selection-statement
        iteration-statement
        jump-statement
```

Commentary

statement header

A statement is the basic unit of executable code. The term *statement header* is sometimes used to refer to the keyword (e.g., **if**, **for**) and the immediately following sequence of tokens between, and including, parentheses in a *selection-statement* or *iteration-statement*.

C++

In C++ local declarations are classified as statements (6p1). They are called *declaration statements* and their syntax nonterminal is *declaration-statement*.

Other Languages

Imperative languages use statements to specify a programs execution time control flow, while *functional* languages use expressions. Other language families use fundamentally different building blocks, for instance *Logic*, or *constraint-based* languages specify relationships and a program *execution* is an attempt to find values that meet these relationships.

Common Implementations

A number of implementations[2, 12, 16, 28] have added extensions that provide support for either concurrent execution of groups of statements, or parallel operations on arrays of values. These extensions tend to be very dependent on particular hardware implementations (whose architecture is usually driven by the kinds of problems they have been designed to solve) and no single model of concurrency, or parallel execution, has achieved a significant market share (compared to the others).

Coding Guidelines

compound statement syntax
function definition syntax

The discussion in the following two subsections is based on drawing a parallel between natural language sentences and source code statements (i.e., statements are the sentences of a programs source code; sequences of them are used to create a compound statement (paragraph) and sequences of these used to build a function definition, a subsection).[1707.1] The reason for drawing this parallel is the desire to make use of some of the findings from research on human sentence processing (there being no equivalent studies performed on source code statements). The following is a broad summary of these findings:

- Over short time periods the syntax of what is read (i.e., words) is remembered, while over longer periods of time only the semantics (i.e., the meaning; which may differ between readers because of integration into their personal world model) is remembered.
- The measure of a persons working memory capacity that has the highest correlation with their performance in reading comprehension tasks is the reading span test.
- The closer previously obtained information is (i.e., the more recently the sentence containing it was read) to the point it is referenced, the higher the probability that readers will correctly integrate it into the information extracted from what they are currently reading.

reading span 1707

[1707.1] In practice a better parallel might be between statements and the *intonation units* commonly used in spontaneous spoken speech. Individual statements often contain a single piece of information and need to be considered in association with other statements, much like speech e.g., “hey . . . ya know that guy John . . . down the poolhall . . . he bought a Harley . . . if you can believe that.”.

What is remembered

Having read a sentence, what does the reader remember about it? Studies[3] have found that over short time periods the syntax (i.e., words) is remembered, while over longer periods of time only the semantics (i.e., the meaning) is remembered. Readers might like to try the following test (based on Jenkins[19]). Part 1: A line at a time, (1) read the sentence on the left, (2) look away and count to five, (3) answer the question on the right, and (4) repeat process for the next line.

```
The girl broke the window on the porch.                         Broke what?
The hill was steep.                                             What as?
The cat, running from the barking dog, jumped on the table.     From what?
The tree was tall.                                              Was what?
The old car climbed the hill.                                   What did?
The cat running from the dog jumped on the table.               Where?
The girl who lives next door broke the window on the porch.     Lives where?
The car pulled the trailer.                                     Did what?
The scared cat was running from the barking dog.                What was?
The girl lives next door.                                       Who does?
The tree shaded the man who was smoking his pipe.               What did?
The scared cat jumped on the table.                             What did?
The girl who lives next door broke the large window.            Broke what?
The man was smoking his pipe.                                   Who was?
The old car climbed the steep hill.                             The what?
The large window was on the porch.                              Where?
The tall tree was in the front yard.                            What was?
The car pulling the trailer climbed the steep hill.             Did what?
The cat jumped on the table.                                    Where?
The tall tree in the front yard shaded the man.                 Did what?
The car pulling the trailer climbed the hill.                   Which car?
The dog was barking.                                            Was what?
The window was large.                                           What was?
```

You have now completed part 1. Please do something else for a minute or so, before moving on to part 2 (which immediately follows).

Part 2: when performing this part, do not look at the sentences from part 1 above. Now, a line at a time, (1) read the sentence on the left, (2) if you think that sentence appeared as a sentence in part 1 write a number between one and five expressing your confidence (one expressing very little confidence and five expressive a lot of confidence in the decision) next to *old*, otherwise write a number representing your confidence level next to *new*, and (3) repeat process for the next line.

```
The car climbed the hill.                                         old___, new ___
The girl who lives next door broke the window.                    old___, new ___
The old man who was smoking his pipe climbed the steep hill.      old___, new ___
The tree was in the front yard.                                   old___, new ___
The window was on the porch.                                      old___, new ___
The barking dog jumped on the old car in the front yard.          old___, new ___
The cat was running from the dog.                                 old___, new ___
The old car pulled the trailer.                                   old___, new ___
The tall tree in the front yard shaded the old car.               old___, new ___
The scared cat was running from the dog.                          old___, new ___
The old car, pulling the trailer, climbed the hill.               old___, new ___
The girl who lives next door broke the large window on the porch. old___, new ___
The tall tree shaded the man.                                     old___, new ___
The cat was running from the barking dog.                         old___, new ___
The cat was old.                                                  old___, new ___
```

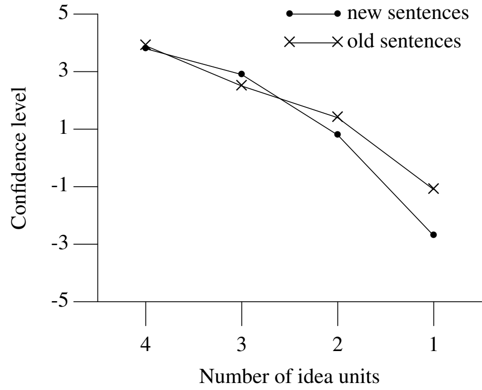


Figure 1707.1: Subject confidence level for having previously seen a sentence containing different numbers of idea units. Based on Bransford and Franks.[3]

```
The girl broke the large window.                    old___, new ___
The car climbed the steep hill.                     old___, new ___
The man who lives next door broke the window.       old___, new ___
The cat was scared.                                 old___, new ___
```

You have now completed part 2. Count the number of sentences you judged to be *old*.

The surprise is that all of the sentences are new.

While reading you abstracted and remembered the general ideas contained in the sentences (they are based on the four *idea sets*, (1) "The scared cat running from the barking dog jumped on the table.", (2) "The old car pulling the trailer climbed the steep hill.", (3) "The tall tree in the front yard shaded the man who was smoking his pipe.", and (4) "The girl who lives next door broke the large window on the porch.").

In a study by Bransford and Franks[3] each idea unit was broken down into sentences containing single ideas (e.g., "The cat was scared."), two ideas (e.g., "The scared cat jumped on the table."), three ideas (e.g., "The scared cat was running from the dog."), and four ideas (e.g., "The scared cat running from the barking dog jumped on the table."). The results (see Figure 1707.1) show that the greater the number of ideas units included in a sentence, the greater a subjects confidence that the sentence was previously seen (independently of whether it had been).

The issue of what people remember about what they have previously read is discussed in more detail elsewhere.

function definition syntax

Processing single sentences

propositional form

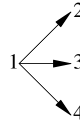
Individual sentence complexity[6] has a variety of effects on human performance. A study by Kintsch and Keenan[22] asked subjects to read single sentences, each containing the same number of words, but varying in the number of propositions they contained (see Figure 1707.2). The time taken to read each sentence and recall it (immediately after reading it) was measured.

The results (see Figure 1707.3) show that reading rate decreases as the number of propositions in a sentence is increased (with the total number of words remaining the same). A later study[23] found that reader performance was also affected by the number of word concepts in a sentence and the grammatical form of the propositions (subordinate or superordinate).

Developers are often told to break up a complex statement or expression into several simpler ones. However, in many cases the quantity of interest is the comprehension cost of all of the code (it is possible that in some cases readers may only need to comprehend a subset of the simpler statements). It is therefore necessary to ask whether reader comprehension effort is minimized by having a single complex statement or having several simpler statements? While it is not yet possible to answer this question, some of the issues are known and the following subsection discusses the integration of information between related sentences.

Romulus, the legendary founder of Rome, took the women of the Sabine by force.

1 (took, Romulus, women, by force)
2 (found, Romulus, Rome)
3 (legendary, Romulus)
4 (Sabine, women)



Cleopatra's downfall lay in her foolish trust in the fickle political figures of the Roman world.

1 (because, α, β)
2 $\alpha \rightarrow$ (fell down, Cleopatra)
3 $\beta \rightarrow$ (trust, Cleopatra, figures)
4 (foolish, trust)
5 (fickle, figures)
6 (political, figures)
7 (part of, figures, world)
8 (Roman, world)

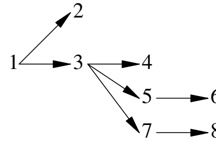


Figure 1707.2: Two sentences, one containing four and the other eight propositions, and their propositional analyses. Based on Kintsch and Keenan.[22]

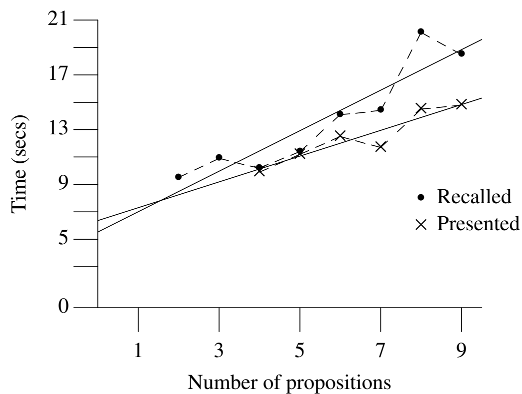


Figure 1707.3: Reading time (in seconds) and recall time for sentences containing different numbers of propositions (straight lines represent a least squares fit; for reading $t = 6.37 + .94P_{pres}$, and for recall $t = 5.53 + 1.48P_{rec}$). Adapted from Kintsch and Keenan.[22]

statements integrating information between

Integrating information between sentences

This subsection discusses peoples performance in integrating information between (or across) sentences they have read. Developers need to integrated information from different source code statements, to obtain a higher-level view of a programs behavior. It is assumed that the main factors affecting the performance of a reader of prose sentences are also the main factors, and in the same proportions, that affect the performance of readers of source code statements. The factors are working memory and the processing performed on the information it contains (see Just and Carpenter[20] for a capacity theory of comprehension).

The relative order in which a developer has to write many statements is dictated by dependencies between the operands they contain. However, there is often a degree of flexibility in the absolute order in which they occur. For instance, some developers initialize all locally declared objects at the start of a function, while others initialize them close to where they are used.

Although various theories of text comprehension have been proposed,[20,21] it is not yet possible to give reliable answers to detailed questions. For instance, in the following three assignments, would moving the assignment to x after the assignment to y reduce the cognitive effort needed to comprehend the value of the expression assigned to z?

```
1  x = ex_1 + ex_2;            /* Could be reordered to follow assignment to y.   */
2  y = complicated_expression; /* Contains no dependencies on previous statement. */
3  z = y + ex_1;
```

Optimizing statement ordering, in those cases where some flexibility is available, to minimize reader cognitive effort when integrating information between statements requires that the relationships between all statements within a function be taken into account. For instance, ex_2 may also appear prior to the assignment to x and there may be a greater benefit to this assignment appearing close to this usage, rather than close to the assignment to z.

Because there is no method of measuring adherence, no guideline recommendation, dealing with statement ordering, is given here.

reading span

memory developer

A study by Daneman and Carpenter[7] investigated the connection between various measures of subjects working memory span and their performance on a reading comprehension task. The two measures of working memory used were the *reading span* and *word span*. In the reading span test subjects have to read, out loud, sequences of sentences while remembering the last word of each sentence, which have to be recalled at the end of the sequence. The number of sentences in each sequence is increased until subjects are unable to successfully recall all the last words. The word span measure is a variant of the digit span test discussed elsewhere, using words rather than digits.

memory digit span

In the Daneman and Carpenter study the reading comprehension task involved subjects reading a narrative passage containing approximately 140 words and then answering questions about facts and events described in the passage. Various passages were constructed, where the distance between information needed to answer some of the questions was varied. For instance, the final sentence of the passage contained a pronoun (e.g., she, her, he, him, or it) that referred to a noun occurring in a previous sentence. Different passages contained the referenced noun in either the second, third, fourth, fifth, sixth, or seventh sentence before the last sentence.

In the excerpt: " . . . river clearing . . . The proceedings were delayed because the leopard had not shown up yet. There was much speculation as to the reason for this midnight alarm. Finally he arrived and the meeting could commence." the question "Who finally arrived?" refers to information contained in the last and third to last sentence. The question "Where was the meeting held?" requires the recall of a fact.

The results showed that there was little correlation between a subjects performance in the word span test and the reading comprehension test. However, there was a strong correlation between their performance in the reading span test and the reading comprehension test (see Figure 1707.4). A similar pattern of results was obtained when the task involved listening, rather than reading. A second experiment included controls to ensure that subjects processed the sentences in the reading span test (rather than simply looking for and remembering the last word) and that differences in rate of reading were not a factor. The pattern of results was unchanged. A study by Turner and Engle[29] found that having subjects verify simple arithmetic identities,

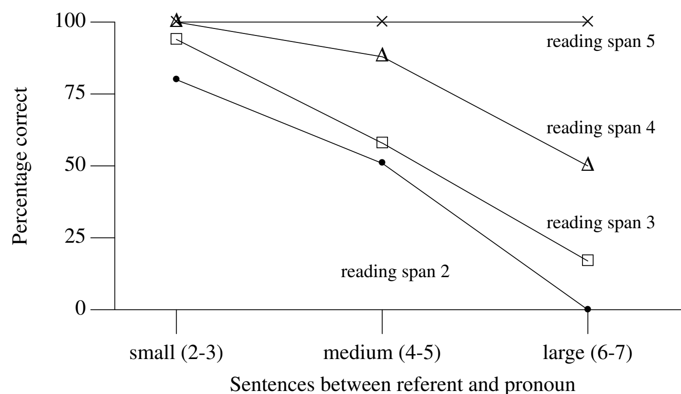


Figure 1707.4: Percentage of correct subject responses to the pronoun reference questions as a function of the number of sentences between the pronoun and the referent noun. Plotted lines are various subject reading spans. Adapted from Daneman and Carpenter.[7]

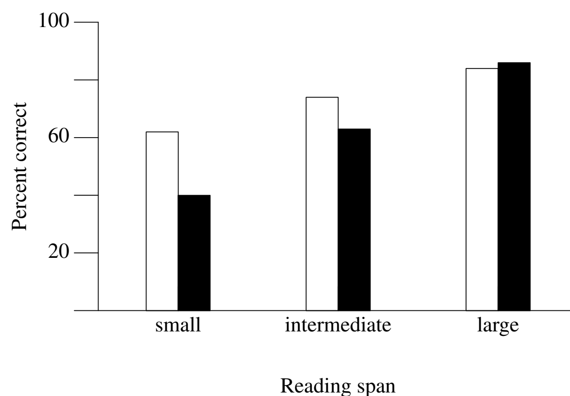


Figure 1707.5: Percentage of correct answers as a function of subject's reading span and the presence or absence of a sentence boundary. Adapted from Daneman and Carpenter.[8]

rather than a reading comprehension test, did not alter the results. However, altering the difficulty of the background task (e.g., using sentences that required more effort to comprehend) reduced performance.

The difference between word span and reading span as measures of working memory is that the first is purely a measure of memory usage while the second also involves processing information (the extent to which processing information consumes, interferes with, or competes with working memory resources is a hotly debated issue). Comprehension of prose involves integrating information within and between sentences. Other studies have found that reading span correlates with performance in a number other tasks, including:

- A study by Daneman and Carpenter[8] investigated subjects performance in integrating information within a single sentence and across two sentences. For instance, is there a difference in comprehension performance between the sentence "There is a sewer near our home who makes terrific suits" (this is an example of what is known as a *garden path* sentence) and "There is a sewer near our home. He makes terrific suits"? The results (see Figure 1707.5) show that a sentence boundary can affect comprehension performance. It was proposed that this difference in performance was caused by readers purging any verbatim information they held, in working memory, about a sentence on reaching its end. The availability of previously read words, in the single sentence case, making it easier to change the interpretation made on the basis of what has already been read.
- There are a number of structural components involved in the reading of prose. For instance, at the

microstructure level words are formed from character sequences and syntactic processing of words occurs, while at the macrostructure level information has to be integrated across sentences and the narrative has to be followed. A study by Graesser, Hoffman, and Clark[15] found that (1) more cognitive resources are allocated to macrostructure than microstructure processing, (2) differences in reading speed could be attributed to different rates of microstructure processing, and (3) variations in reader goals affect the rate at which the macrostructure is processed.

- A study by Glanzer, Fischer, and Dorfman[14] investigated the affects an interruption had on subjects performance, when reading prose. The results showed that when reading organized text (a sequence of sentences having dependencies between them) an interruption (requiring a different task to be performed) caused information on the last two sentences to be lost from working memory. This loss of information caused a reduced subjects performance in reading the remaining sentences (unless they were able to reread the previous two sentences). An interruption did not cause a reduction in performance when the sentences were not organized (i.e., they were not related to each other).
- A study by Gibson and Thomas[13] found that subjects were likely to perceive complex ungrammatical sentences as being grammatical. Subjects handling complex sentence that exceeded working memory capacity by *forgetting* parts of the syntactic structure of the sentence, resulting in a grammatically correct sentence.
- Text inference often involve more than integrating information between sentences. Knowledge about the real world is often required. A study by Singer and Ritchot[27] showed subjects pairs of sentences and asked them to answer questions about inferences that could be drawn from these sentences. For instance, the two sentences "Valerie left early for the birthday party. She spent an hour shopping in the mall." would be expected to activate reader's knowledge of bringing birthday presents to parties, while the sentences "Valerie left the birthday party early. She spent an hour shopping in the mall." would not be expected to activate such knowledge. The results showed that there was no correlation between reading span and knowledge access when making such bridging inferences. The result of this study implies that while reading span provides a good measure of a person's ability to integrate information between sentences, it does not provide a measure of their ability to integrate information they have just read with information held in long-term memory.

A study by Ehrlich and Johnson-Laird[9] asked subjects to draw diagrams depicting the spatial layout of everyday items specified by a sequence of sentences. The sentences varied in the extent to which an item appearing as the object (or subject, or not at all) in one sentence appeared as the subject (or object, or not at all) in the immediately following sentence. For instance, there is referential continuity in the sentence sequence "The knife is in front of the pot. The pot is on the left of the glass. The glass is behind the dish.", but not in the sequence "The knife is in front of the pot. The glass is behind the dish. The pot is on the left of the glass.".

The results found that when the items in the sentence sequences had referential continuity 57% of the diagrams were correct, compared to 33% when there was no continuity. Most of the errors for the non-continuity sentences were items missing from the diagram drawn and subjects reported finding it difficult to remember the items as well as the relation between them.

A study by Frase[11] found that the kind of deduction subjects had to perform (e.g., forward to backward chaining) also affects their performance. For instance, verifying the correctness of the deduction in "Some X are Y. Some Z are X. Therefore, some Z are Y." requires forward chaining, while verification of "Some Y are X. Some X are Z. Therefore, some Z are Y." requires backward chaining. The results showed that subjects were slower and more error prone when performing backward chaining.

statement visual layout

Visual layout of statements

Source code statements are generally written one per line, a practice that is different from prose sentences (which generally appear as a continuous stream of characters, with one sentence immediately following another on the same line, with line breaks decided by the space available for the next word). Statements are usually indented on the left to show block nesting level (this issue is discussed elsewhere).

reading kinds of visual skimming

```
#include <string.h>
#define v1 13
#define v2 0
#define v3 1
int v4(char v5[], int *v6) { int v7, v8; *v6=v2; v8=strlen(v5); if
(v8 > v1) { *v6=v3; } else { for (v7=0; v7 < v8; v7++) { if ((v5[v7]
< '0') || (v5[v7] > '9')) { *v6=v3; } } } }
```

The two cases discussed here are the practice of writing two or more statements on the same line and what to do when a statement will not fit on a single line. These cases are only issues that need discussion because source is not always read in detail, it is sometimes rapidly scanned visually. During rapid visual scanning of the source its left edge is often used as a reference point for the start of individual statements. This usage suggests the following:

reading
kinds of

- If multiple statements appears on the same line, readers using rapid visual scanning may only the first one may be noticed. For this reason some coding guideline documents recommend that a line contain at most one statement. However, there can be benefits to placing more than one statement on the same line. For instance, in the following fragment assigns values representing the four corners of a square:

  ```
  x1=1.0; y1=4.0; x2=3.0; y2=4.0;
  x3=1.0; y3=2.0; x4=3.0; y4=2.0;
  ```

  and the layout of the statements is suggestive of what they represent (other combinations are possible), and has the possibly benefits of reducing comprehension effort and being able to have more statements simultaneously visible on the screen. These benefits need to be balanced against the potential cost of readers failing to notice the assignment to other objects on the same line. Within a **switch** statement some of the labeled statements may only perform a single operation. It is possible to visually highlight this operation and contrast it with operations performed by other labeled statements by placing the *uninteresting* termination statement to the right of the statement, rather than underneath it.

  ```
  case 1: i--;            break;
  case 2: k++;            break;
  case 3: ++expr;ssion--; return;
  case 4: func();         break;
  ```

  These benefit need to be balanced against the potential cost of readers failing to notice a difference in termination statements, or treating two statement as one statement. The following guideline recommendation allows for some degree of flexibility.

Rev 1707.1

Multiple statements shall only appear on the same line, in a visible form, when the probable reduction in reader comprehension costs is greater than the probable increase in costs caused by mistakes made when quickly scanning the source visually.

- Readers are likely to treat the start of every line that appears immediately above or below adjacent lines, in a sequence of statements, as the start of a different statement. One way of reducing this possibility is to use a significant amount of right indentation on the second and subsequent lines relative to the start of the first line (which is what developers appear to do).

Studies have found that peoples memory for objects within their visual field of view is organized according to the relative positions of those objects. For instance, a study by McNamara, Hardy, and Hirtle[24] gave subjects two minutes to memorize the location of objects on the floor of a room (see Figure 1707.6). The objects were then placed in a box and subjects were asked to place the objects in their original position. The memorize/recall cycle was repeated, using the same layout, until the subject could place all objects in their correct position.

grouping
spatial location

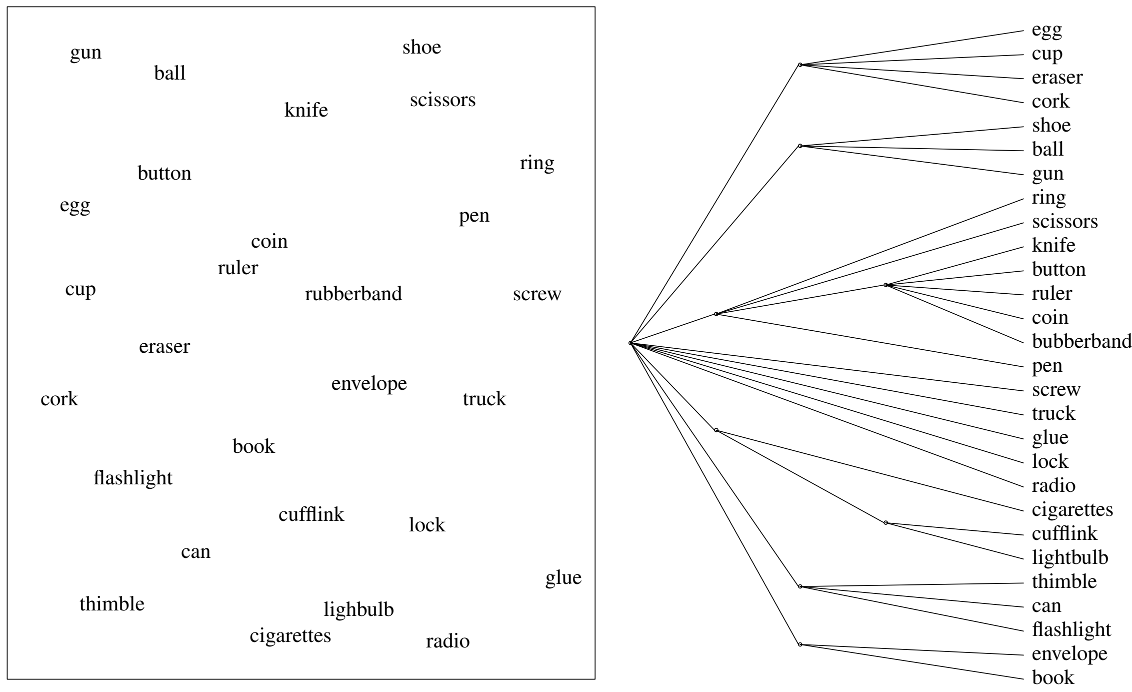


Figure 1707.6: Example of an object layout and the corresponding ordered tree for one of the subjects. Based on McNamara, Hardy, and Hirtle.[24]

The order in which each subject recalled the location of objects was used to create a hierarchical tree (one for each subject). The resulting trees (see Figure 1707.6 for an example) showed how subjects' spatial memory of the objects seen had a hierarchical organization, with the spatial distance between items being a significant factor in its structure.

Source code statements usually have a one-dimensional organization, down the display (indentation does not change the one-dimensional organization; multiple statements on the same line is not really fully two-dimensional).

If developer memory of statement sequences is hierarchical, with visual distance between statements being a significant factor in the formation of clusters, then ordering statements so that related ones are close to each other may improve recall performance. The extent to which statements can be reordered will depend on the relative order in which their side effects must occur.

metrics introduction

Many software engineering metrics use statements as the unit of measurement.

Usage

Of the approximately 2,204,000 statements in the visible form of the `.c` files 60.3% were *`expression-statement`*s, 21.3% *`selection-statement`*s, 15.0% *`jump-statement`*s, and 3.4% *`iteration-statement`*s. Of these 5.4% were *`labeled-statement`*s.

Semantics

---

statement

A *`statement`* specifies an action to be performed. 1708

Commentary

This defines the term *statement*. This definition might also be said to include those declarations that include initializers. However, *`statement`* is also a terminal of the C syntax. Like declarations, statements can also cause identifiers to be declared (e.g., the type of a cast operator). So the two do share some characteristics. In the case of the null statement no action is performed.

null statement

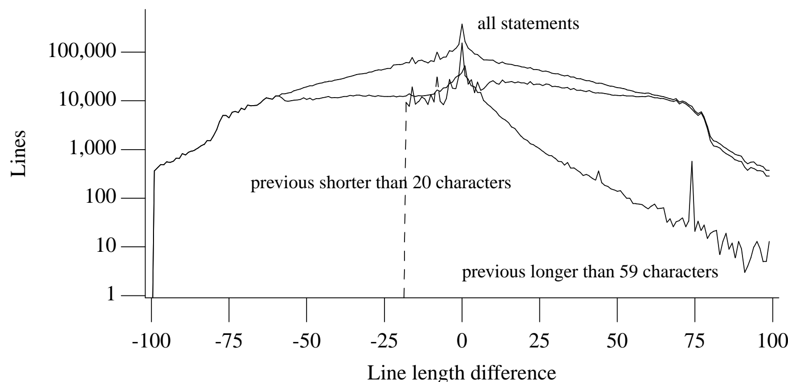


Figure 1707.7: Visible difference in offset of last non-space character on a line between successive lines, in the visible form of the `.c` files (horizontal tab characters were mapped to 8 space characters), for lines of various lengths, i.e., those whose previous line contained 60 or more characters, and those whose previous line contains less than 20 characters. There are ten times fewer lines sharing the same right offset as sharing the same left offset (see Figure 1707.8). Based on the visible form of the `.c` files.

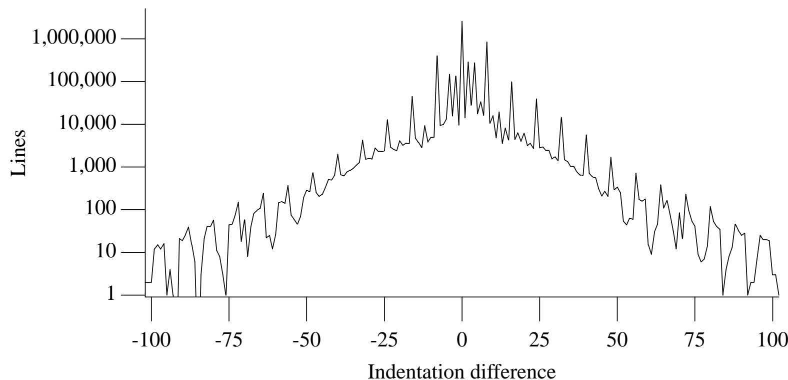


Figure 1707.8: Visible difference in relative indentation of first non-space character on a line between successive lines in the visible form of the `.c` files (horizontal tab characters were mapped to 8 space characters). The smaller peaks around zero are indentation differences of two characters. The wider spaced peaks have a separation of eight characters. Individual files had more pronounced peaks. Based on the visible form of the `.c` files.

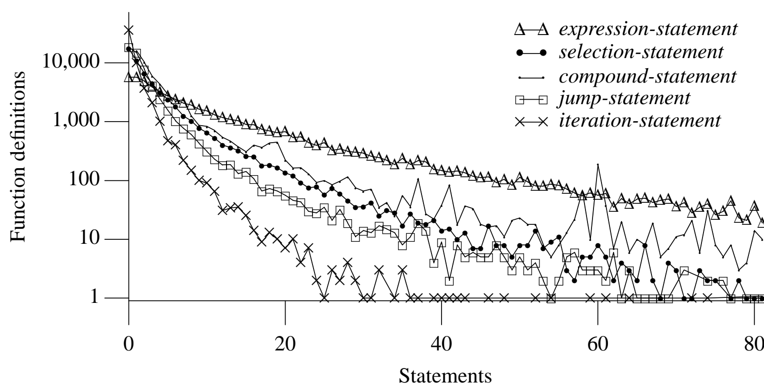


Figure 1707.9: Number of function definitions containing a given number of each kind of statement. Based on the translated form of this book's benchmark programs.

C++

The C++ Standard does not make this observation.

Other Languages

Some languages do not define things called *statements*, everything is an expression. While other languages do define them, and always allow them to return values, just like C expressions. Programs written in logic programming languages (e.g., Prolog) contain a list of facts, not statements, to execute. The user of such programs states a goal and the implementation attempts to prove, or disprove it, using the available facts.

Coding Guidelines

redundant code
dead code

It is possible to write statements whose action, when performed, does not affect the output of a program. This issue is discussed elsewhere. A sequence of one or more statements whose actions can never be performed is commonly known as *dead code*.

---

statement executed in sequence

Except as indicated, statements are executed in sequence. 1709

Commentary

The sequence referred to here is that obtained by starting with the first token in a source file and parsing to the end of that file. The exceptions are caused by statements which change the flow of control, causing a statement at the start of another sequence to be executed. The terms *order of execution*, or *execution order* are sometimes used to describe the order in which statement sequences are executed. This specification has been interpreted as having a wider meaning by the C committee (see the response to DR #087). It is taken to imply that the execution of two functions cannot overlap (i.e., occur in parallel). For instance, in the expression `g() + h()` one of the functions is chosen to be executed and that function must execute a **`return`** statement before the other function can start execution.

Performing the actions specified by statements is what most programs spend most of their time doing (issues such as idling, waiting for an I/O to complete, and system resource management, such as page swapping, are not considered here). In some applications it is important to be able to make accurate estimates of the time needed to execute a given sequence of statements. For instance, in realtime applications there may be a fixed time window within which certain calculations must be carried out, either because new data is arriving or the result of the calculation is needed.

processor pipeline
cache
iteration statement syntax

Accurately estimating the best and worst-case execution times (*BCET* and *WCET*) of a statement is not always as simple as summing the execution times of the sequence of machine instructions generated for that statement. Issues such as instruction pipelining, caching of data and instructions, and dependencies between instructions (that create interlocks) all complicate the calculation. Various tools have been developed for estimating the worst-case execution time of C source code. Englblom[10] reviews the available techniques and proposes worst-case execution time analysis method. The subproblem of estimating the number of iterations of a loop is discussed elsewhere.

Other Languages

Some languages (e.g., Algol 68) contain constructs that allow developers to specify that certain statements may be executed in parallel. Other languages (e.g., Java) support threads of execution where the function is the smallest unit of concurrent execution.

Common Implementations

basic block 1710

Translators that perform little or no optimization usually generate machine code on a statement by statement basis. This severely restricts the savings that can be made and considering sequences of statements provides an opportunity to make greater savings (the actual unit considered is usually a basic block). The OpenMP C++ API specification[2] defines directives that can be included in source code to support symmetric multiprocessing (SMP).

```
1   int f(void)
2   {
```

```
3   /* ... */
4      #pragma omp parallel
5      {
6      /*
7       * Code in this block potentially executed
8       * in another processor thread.
9       */
10     }
11  /* ... */
12  }
```

1710 A *block* allows a set of declarations and statements to be grouped into one syntactic unit.

block

Commentary

This defines the term *block*. The term *compound statement* and its association with block is discussed elsewhere. In various contexts a block might exist without there being a compound statement present.

compound statement syntax
block selection sub-statement

C90

> *A compound statement (also called a block) allows a set of statements to be grouped into one syntactic unit, which may have its own set of declarations and initializations (as discussed in 6.1.2.4).*

compound statement syntax

The term *block* includes compound statements and other constructs, that need not be enclosed in braces, in C99. The differences associated with these constructs is called out in the sentences in which they occur.

C++

The C++ Standard does not make this observation.

Common Implementations

basic block

The term *basic block* is often used by compiler writers to refer to a sequence of instructions that has one entry point and one exit point. The basic block is often the region of code over which many optimizers operate. More sophisticated optimizers consider the effects of multiple basic blocks. A block may also be a basic block, but if it contains any statements or operators that have conditional flow of control it will be a sequence of basic blocks. The implementation of C operators such as function call, logical-AND, and the comma operator all terminate one basic block and start another; as do the statements **goto** and **return** and labeled statements.

Table 1710.1: Occurrence of constructs that terminated execution of a basic block during execution of PostgreSQL processing the TPC-D benchmark. Adapted from Ramirez, Larriba-Pey, Navarro, Serrano, Torrellas, and Valero.[26]

Basic Block Type	Static Count (thousand)	Dynamic Count (billion)
Branch	54.026 (42.4%)	4.0 (50.2%)
Fall-through	31.120 (24.4%)	1.8 (22.4%)
Function return	32.052 (25.2%)	1.1 (13.7%)
Function call	10.228 (8 %)	1.1 (13.7%)

Writers of translators and processor designers want the average number of machine instructions executed in a basic block to be as large as possible. For translators long sequences of code with continuous flow of control increases the probability of common subexpressions occurring and the consequent reuse, rather than recalculation, of values. For processors large basic blocks enable them to keep their pipelines full, minimizing average instruction execution time. Function inlining can remove one of the constructs that cause basic blocks to end, function calls (see Table 1710.1), increasing the length of some basic blocks.

processor pipeline
inline suggests fast calls

for statement

Table 1710.2 lists average basic block lengths for some complete programs. The special case of iteration statements is discussed elsewhere.

Table 1710.2: Mean number of machine instructions executed per basic block (i.e., total number of instructions executed in a function divided by the total number of basic blocks executed in that function) for a variety of SPEC benchmark programs. *Leaf* refers to functions that do not call any other functions, while *Non-Leaf* refers to functions that contain calls to other functions. Based on Calder, Grunwald, and Zorn.[4]

Program	Leaf	Non-Leaf	Program	Leaf	Non-Leaf
burg	6.8	4.9	eqntott	9.1	5.4
ditroff	6.8	4.7	espresso	5.0	5.1
tex	10.4	8.5	gcc	5.2	5.7
xfig	4.8	5.3	li	2.9	5.7
xtex	7.3	5.8	sc	3.5	4.2
compress	18.4	5.7	Mean	7.3	5.5

function definition syntax

Just as optimizers moved up from working at the single statement level, they eventually moved up from working at the single basic block level (the issue of whole function optimization is discussed elsewhere). Translator output (machine code) for each block often occurs in the program image in the same sequential order as the basic blocks in the source code. However, processors can have a number of characteristics that can cause this ordering to be sub-optimal, including:

jump statement causes jump to

cache

- *Span dependent branch instructions.* This issue is discussed elsewhere.
- *Instruction caches.* The size of a function or basic block, the total size of the cache and the size of a cache line all need to be considered when working out the optimal layout of instruction sequences in memory.[18]

processor pipeline

basic block 1710

- *Instruction pipelines.* Basic blocks can be reordered to ensure that they start on alignments best suited to a processor.[30]

storage dividing up

- *Storage organized in fixed sized pages.* Having infrequently executed instructions in the same page as frequently executed instructions can increase execution overheads (through increased swapping of pages). Storing infrequently executed instructions together in different pages can increase overall performance (the overhead of the extra instruction to jump back, at the end of an else arm, is offset by the removal of the instruction that would otherwise be needed, in the if arm, to jump around the else).

The memory manager prefetches sequences of instructions to fill a complete cache line. Optimal use of this prefetching occurs if all instructions in a cache line are executed. In the example below, whichever arm of the **`if`** statement is executed, it is likely that some instructions from the non-executed arm will be fetched into a cache line.

```
1  if (x == 2)
2     {
3     /* Frequently executed code. */
4     }
5  else
6     {
7     /* Infrequently executed code. */
8     }
9  /* other code */
```

If it is known (for instance, through dynamic profiling information[25]) that one of the arms is executed more frequently than the other, it may be worthwhile reordering basic blocks in the program image. One technique is to place infrequently executed blocks in an area of storage reserved for such cases. The generated code being rewritten to jump to the infrequent block, and back again (the rewritten form of the above example is in C form below). This reorganization is likely to result in a higher percentage of the instructions contained in a cache line being executed.

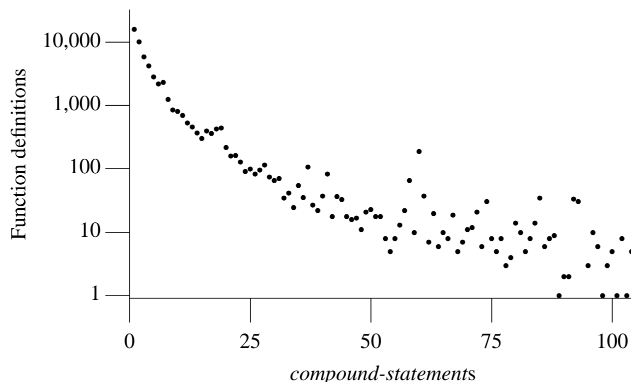


Figure 1710.1: Number of function definitions containing a given number of `compound-statement`s. Based on the translated form of this book's benchmark programs.

```
1   if (x != 2)
2      goto INFREQ_15_s;
3
4      {
5      /* Frequently executed code. */
6      }
7   INFREQ_15_e:
8   /* other code */
9
10
11  /* ... */
12  goto INFREQ_14_e;
13  INFREQ_15_s:
14     {
15     /* Infrequently executed code. */
16     }
17  goto INFREQ_15_e;
18  INFREQ_16_s:
19  /* ... */
```

The major problem associated with optimizing code layout at translation time is that information on frequency of basic block execution usage is needed. Developers may be unwilling or unable to gather this information, or it may be very difficult to obtain reliable information because of different end user usage patterns.

One solution to this problem is to perform what has been called *just in time code layout*.[5] The usage patterns of an executing program were monitored to decide when it is worthwhile moving code within the executing program image, to improve the instruction cache hit rate. Performance results comparable to those obtained for profile based methods have been obtained for programs running under Unix, but results for programs running under Windows NT were mixed (as they were for profile based code layout).

As processor power and available storage capacity has increased, the ability to consider code layout from more global perspectives has become possible. At first researchers built translators that reordered statements within basic blocks, and then moved on to reordering basic blocks within individual functions.[1] The break has now been made with source level constructs and the latest code layout algorithms are based purely on frequency of basic block execution over the entire program[17] (the function definitions to which the basic blocks might be said to belong have become irrelevant).

Usage

Usage information on block nesting is discussed elsewhere. limit block nesting

1711 The initializers of objects that have automatic storage duration, and the variable length array declarators of

object initializer evaluated when

ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.

Commentary

Both declarations and compound literals can create objects having automatic storage duration and an initializer. Storage for objects that have automatic storage duration, and non VLA type, will have been created at the start of their lifetime (when the block that declares them was entered). As well as performing initialization, the evaluation of a declaration for an object having a variable length array type also allocates storage for it.

automatic storage duration

VLA lifetime starts/ends

indeterminate value

The indeterminate value stored in an object is a conceptual value in the sense that no pattern of bits need be stored in the object by an implementation (because no predictable behavior is required to occur, should this value be accessed).

initialization in list order

In one particular case, overlapping initializers, the evaluation of the initializers need not occur as if they were a sequence of assignment statements.

C90

Support for variable length array types is new in C99.

C++

Support for variable length array types is new in C99 and they are not specified in the C++ Standard.

Other Languages

This initialization behavior is found in the majority of programming languages (although many do not support the mixing of statements are declarations).

Common Implementations

The indeterminate value stored in an object may be affected by the initialization of other objects defined in the same block. For instance, if a several objects are initialized to the same value it may be more efficient to initialize a block of storage, which may include space allocated to uninitialized objects, using a loop rather than individual stores.

Coding Guidelines

The evaluation of variable length array declarators can generate side effects. However, because the expression appearing between `[` and `]` is not a full expression there is no sequence point after its evaluation (there is a sequence point at the end of the full declarator that contains it). For this reason the guideline given for the evaluation of full expressions is not applicable here. However, until the use of variable length arrays becomes more common and the ordering of side effects in their evaluation becomes an issue worth addressing a guideline recommendation is not cost effective.

full expression sequence points 1712

full declarator sequence point

sequence points all orderings give same value ??

```
1  #include <stdio.h>
2
3  int glob;
4
5  void f(void)
6  {
7  int
8      /*
9       * Unspecified order of evaluation of expressions in a full declarator.
10      */
11     a_1[printf("Hello ")][printf("world\n")],
12
13     /*
14      * Same object modified more than once between sequence
15      * points -> undefined behavior.
16      */
17     a_2[glob++][glob++];
18 }
```

The extent to which developers might draw an, incorrect, parallel between a compound literal containing constant expressions only and string literals (which have static storage duration) is not known. One important difference is that any modification of the unnamed object, denoting the compound literal, will be overwritten when the original definition is executed again. Until more experience in developer use of compound literals has been gained there is no point in discussing this issue further.

string literal static storage duration

Example

```
1  #include <stdio.h>
2
3  extern int glob;
4  struct T {
5            int mem;
6           };
7
8  void f(void)
9  {
10 int i = 0;
11
12 START_LOOP:
13 if (i == 10)
14    goto END_LOOP;
15
16 i++;
17    int      loc_1 = glob;
18    struct T loc_2 = { 1 },             loc_3 = { glob },
19            *ploc_a = (struct T){ 1 }, *ploc_b = (struct T){ glob };
20
21             loc_1++;
22             loc_2.mem++;               loc_3.mem++;
23             ploc_a->mem++;             ploc_b->mem++;
24
25 goto START_LOOP;
26 END_LOOP:;
27
28 if (       (loc_1 != (glob+1))  ||
29            (loc_2.mem != 1+1)   ||    (loc_3.mem != (glob+1)) ||
30            (ploc_a->mem != 1+1) ||    (ploc_b->mem != (glob+1)))
31    printf("This is not a conforming implementation\n");
32 }
```

1712 A *full expression* is an expression that is not part of another expression or of a declarator.

full expression

Commentary

This defines the term *full expression*. While the term may only be used once outside of this C paragraph, it can be of use in disambiguating what is meant during conversations between developers. Developers often use the shorter term *expression*. However, this term might also be applied to individual arguments in a function call, or an array index. The terms *top-level expression* or *outer most expression* are sometimes used to refer to what the C Standard calls a full expression. A component of an expression is often referred to as a *subexpression*. An expression that is part of a declarator is included within the definition of a full declarator.

parenthesized expression nesting levels

full declarator

The subexpression x+y is said to be a *common subexpression* of x+y + a[x+y] (it may also be a CSE of other expressions, provided the values of x and y are unchanged).

common subexpression

C++

1.9p12

A full-expression is an expression that is not a subexpression of another expression.

The C++ Standard does not include declarators in the definition of a full expression. Source developed using a C++ translator may contain declarations whose behavior is undefined in C.

```
1  int i;
2
3  void f(void)
4  {
5  int a[i++][i++]; /* undefined behavior */
6                   // a sequence point between modifications to i
7  }
```

Each of the following is a full expression: 1713

Commentary

These are all of the contexts in which an expression, that needs to be evaluated during program execution, can occur.

C++

The C++ Standard does not enumerate the constructs that are full expressions.

Table 1713.1: Occurrence of full expressions in various contexts (as a percentage of all full expressions). Based on the translated form of this book's benchmark programs.

Context of Full Expression	Occurrence	Context of Full Expression	Occurrence
expression statement	65.9	**for** *expr-1*	1.6
if controlling expression	16.4	**for** controlling expression	1.5
return expression	6.2	**for** *clause-1*	1.5
object declaration initializer	4.2	**switch** controlling expression	0.6
while controlling expression	2.1		

---

full expression initializer

an initializer; 1714

Commentary

This is the complete expression appearing between braces, not the individual comma-separated expressions.

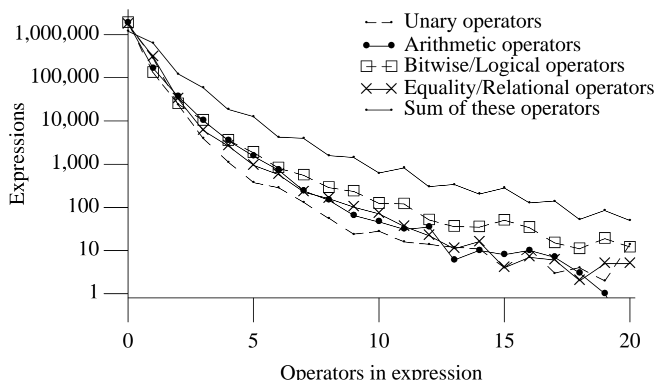


Figure 1712.1: Number of expressions containing a given number of various kinds of operator, plus a given number of all of these kinds of operators. Based on the visible form of the `.c` files.

1715 the expression in an expression statement;

full expression expression statement

Commentary

In an expression statement the entire statement is an expression.

expression statement syntax

C++

The following is not the same as saying that an expression statement is a full expression, but it shows the effect is the same:

All side effects from an expression statement are completed before the next statement is executed. 6.2p1

Other Languages

In many languages C's most common form of expression statement (i.e., simple assignment) consists of two expressions. These are the expressions appearing to the left and right of the assignment token, which are considered to be separate expressions.

1716 the controlling expression of a selection statement (**if** or **switch**);

full expression controlling expression

Commentary

This ensures consistent behavior for all control flows leading from the evaluation of the controlling expression. For instance, in the following sequence of statements:

```
1 if (i++)
2    j++;
3 k=i;
```

the value of `i` assigned to `k` does not depend on the sequence point contained within the arm of the **if** statement.

1717 the controlling expression of a **while** or **do** statement;

Commentary

The rationale here is the same as for selection statements.

1716 full expression controlling expression

1718 each of the (optional) expressions of a **for** statement;

Commentary

In C90 the semantics of the **for** statement were expressed in terms of an equivalent **while** statement. This equivalence required that each of the (optional) expressions be equivalent to a full expression.

for statement

Other Languages

In many languages the expressions in a **for** statement are evaluated once, prior to the first iteration of the loop, every time the statement is encountered during program execution.

1719 the (optional) expression in a **return** statement.

return expression sequence point

Commentary

A sequence point occurs just before a function is called. Having one occur just before the called function returns allows the execution of a function call to be treated as an indivisible operation (from the point of view of the code containing the call).

function call sequence point

1720 The end of a full expression is a sequence point.

full expression sequence point

sequence points

Commentary

The order in which the sequence points at the end of full expressions occur is fully defined by a programs flow of control. Unlike the ordering of sequence points within a full expression (assuming there is more than one), which occur in one of several possible orderings.

Coding Guidelines

sequence points all orderings give same value ??

The guideline recommendation dealing with sequence point evaluation ordering applies to sequence points within a full expression. The sequence point at the end of full expressions occurs in the same order for all implementations.

Forward references: expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the `return` statement (6.8.6.4). 1721

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1985.
2. Anon. OpenMP C and C++ application program interface. Technical Report Version 2.0, OpenMP Architecture Review Board, Mar. 2002.
3. J. D. Bransford and J. J. Franks. The abstraction of linguistic ideas. *Cognitive Psychology*, 2:331–350, 1971.
4. B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1995.
5. J. B. Chen and B. D. D. Leupen. Improving instruction locality with just-in-time code layout. In USENIX, editor, *The USENIX Windows NT Workshop 1997*, pages 25–32, Berkeley, CA, USA, Aug. 1997. USENIX.
6. H. Cheung and S. Kemper. Competing complexity metrics and adults' production of complex sentences. *Applied Psycholinguistics*, 13:53–76, 1992.
7. M. Daneman and P. A. Carpenter. Individual differences in working memory and reading. *Journal of Verbal Learning and Verbal Behavior*, 19:450–466, 1980.
8. M. Daneman and P. A. Carpenter. Individual differences in integrating information between and within sentences. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 9(4):561–584, 1983.
9. K. Ehrlich and P. N. Johnson-Laird. Spatial descriptions and referential continuity. *Journal of Verbal Learning and Verbal Behavior*, 21:296–306, 1982.
10. J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Apr. 2002.
11. L. T. Frase. Associative factors in syllogistic reasoning. *Journal of Experimental Psychology*, 76(3):407–412, 1968.
12. N. Gehani and W. D. Roome. *The Concurrent C Programming Language*. Silicon Press, 1989.
13. E. Gibson and J. Thomas. Memory limitations and structured forgetting: The perception of complex ungrammatical sentences as grammatical. *Language and Cognitive Processes*, 14(3):225–248, 1999.
14. M. Glanzer, B. Fischer, and D. Dorfman. Short-term storage in reading. *Journal of Verbal Learning and Verbal Behavior*, 23:467–486, 1984.
15. A. C. Graesser, N. L. Hoffman, and L. F. Clark. Structural components of reading time. *Journal of Verbal Learning and Verbal Behavior*, 19:135–151, 1980.
16. S. Graphics. *Cray Standard C/C++ Reference Manual*. Silicon Graphics, Inc, Mountain View, CA, USA, 3.6 edition, June 2002.
17. R. E. Hank. *Region-based compilation*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.
18. A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. Technical Report Research Report 96/3, Compaq Western Research Laboratory, 1996.
19. J. J. Jenkins. Remember that old theory of memory? Well, forget it! *American Psychologist*, 29(11):785–795, 1974.
20. M. A. Just and P. A. Carpenter. A capacity theory of comprehension: Individual differences in working memory. *Psychological Review*, 99(1):122–149, 1992.
21. W. Kintsch. *Comprehension: A paradigm for cognition*. Cambridge University Press, 1998.
22. W. Kintsch and J. Keenan. Reading rate and retention as a function of the number of propositions in the base structure of sentences. *Cognitive Psychology*, 5:257–274, 1973.
23. W. Kintsch, E. Kozminsky, W. J. Streby, G. McKoon, and J. M. Keenan. Comprehension and recall of text as a function of content variables. *Journal of Verbal Learning and Verbal Behavior*, 14:196–214, 1975.
24. T. P. McNamara, J. K. Hardy, and S. C. Hirtle. Subjective hierarchies in spatial memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 15(2):211–227, 1989.
25. 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 *LNCS*, pages 340–359. Springer, 2000.
26. A. Ramírez, J.-L. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Code reordering of decision support systems for optimized instruction fetch. In *IEEE International Conference on Parallel Processing (ICPP99)*, 1999.
27. M. Singer and K. F. M. Ritchot. The role of working memory capacity and knowledge access in text inference processing. *Memory & Cognition*, 24(6):733–743, 1996.
28. Sun. *C User's Guide*. Sun Microsystems, Inc, Palo Alto, CA, USA, revision a edition, May 2000.
29. M. L. Turner and R. W. Engle. Is working memory capacity task dependent? *Journal of Memory and Language*, 28:127–154, 1989.
30. C. Young, D. S. Johnson, D. R. Karger, and M. D. Smith. Near-optimal intraprocedural branch alignment. *SIGPLAN Notices*, 32(5):183–193, May 1997.