The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.8.5 Iteration statements

iteration statement syntax

iteration-statement:

while (*expression*) *statement*
do *statement* **while** (*expression*) ;
for ($expression_{opt}$; $expression_{opt}$; $expression_{opt}$) *statement*
for (*declaration* $expression_{opt}$; $expression_{opt}$) *statement*

Commentary

The terms *loop header* or *head of the loop* are sometimes used to refer to the source code location containing the controlling expression of a loop (in the case of a **for** statement it might be applied to all three components bracketed by parentheses).

It is often claimed that programs spend 90% of their time executing 10% of their code. This characteristic is only possible if the time is spent in a subset of the programs iteration statements, or a small number of functions called within those statements. While there is a large body of published research on program performance, there is little evidence to back up this claim (one study[14] found that 88% of the time was spent in 20% of the code, while analysis[19] of some small embedded applications found that 90% of the time was spent in loops). It may be that researchers are attracted to applications which spend their time in loops because there are often opportunities for optimization. Most of existing, published, execution time measurements are based on engineering and scientific applications, for database oriented applications[13] and operating systems[16] loops have not been found to be so important.

The **;** specified as the last token of a **do** statement is not needed to reduce the difficulty of parsing C source. It is simply part of an adopted convention.

C90

Support for the form:

for (*declaration* $expr_{opt}$; $expr_{opt}$) *statement*

is new in C99.

C++

The C++ Standard allows local variable declarations to appear within all conditional expressions. These can occur in **if**, **while**, and **switch** statements.

Other Languages

Many languages require that the lower and upper bounds of a **for** statement be specified, rather than a termination condition. They usually use keywords to indicate the function of the various expressions (e.g., Modula-2, Pascal):

```
1  FOR I=start TO end BY step
```

Some languages (e.g., BCPL, Modula-2) require `step` to be a translation time constant. Both Ada or Pascal require **for** statements to have a step size of one. Ada uses the syntax:

```
1  for counter in 1..10
2     loop
3     ...
4  for counter in reverse 1..10
5     loop
6     ...
```

which also acts as the definition of `counter`.

Cobol supports a **PERFORM** statement, which is effectively a **while** statement.

```
1          PERFORM UNTIL quantity > 1000
2         * some code
3          END-PERFORM
```

The equivalent looping constructs In Fortran is known as a **do** statement. A relatively new looping construct, at least in the Fortran Standard, is **FORALL**. This is used to express a looping computation in a form that can more easily be translated for parallel execution. Some languages (e.g., Modula-2, Pascal) use the keywords **repeat/until** instead of **do/while**, while other languages (e.g., Ada) do not support an iteration statement with a test at the end of the loop.

A few languages (e.g., Icon[3] which uses the term *generators*) have generalized the looping construct to provide what are commonly known as *iterators*. An iterator enumerates the members of a set (a mechanism for accessing each enumerated member is provided in the language), usually in some unspecified order, and has a loop termination condition.

Common Implementations

Many programs spend a significant percentage of their time executing iteration statements. The following are some of the ways in which processor and translator vendors have responded to this common usage characteristic:

Measuring implementations

- Translator vendors wanting to optimize the quality of generated machine code have a number of optimization techniques available to them. A traditional loop optimization is strength reduction[4] (which replaces costly computations by less expensive ones), while more ambitious optimizers might perform hoisting of loop invariants and loop unrolling. Loop invariants are expressions whose value does not vary during the iteration of a loop; such expressions can be hoisted to a point just outside the start of the loop. Traditionally translators have only performed loop unrolling on **for** statements. (Translation time information on the number of loop iterations and step size is required; this information can often be obtained by from the expressions in the loop header, i.e., the loop body does not need to be analyzed.) More sophisticated optimizations include making use of data dependencies to order the accesses to storage. As might be expected with such a performance critical construct, a large number of other optimization techniques are also available.

  translator optimizations; loop unrolling; data dependency

- Processor vendors want to design processors that will execute programs as quickly as possible. Holding the executed instructions in a processor's cache saves the overhead of fetching them from storage and most processors cache both instructions and object values. Some processors (usually DSP) have what is known as a *zero overhead loop buffer* (effectively a software controlled instruction cache). The sequence of instructions in such a loop buffer can be repetitively executed with zero loop overhead (the total loop count may be encoded in the looping instruction or be contained in a register). Because of their small size (the Agere DSP16000[1] loop buffer has a limit of 31 instructions) and restrictions on instructions that may be executed (e.g., no instructions that change the flow of control) optimizers can have difficulty making good of such buffers.[17] The characteristics of loop usage often means that successive array elements are accessed on successive loop interactions (i.e., storage accesses have spatial locality). McKinley and Temam[10] give empirical results on the effect of loops on cache behavior (based on Fortran source). Some CISC processors support a decrement/increment and branch on nonzero instruction;[5,9] ideal for implementing loops whose termination condition is the value zero (something that can be arranged in handwritten assembler, but which rarely happens in loops written in higher-level languages— Table 1763.1). The simplifications introduced by the RISC design philosophy did away with this kind of instruction; programs written in high-level languages did not contain enough loops of the right kind to make it cost effective supporting such an instruction. However, one application domain where a significant amount of code is still written in assembler (because of the comparatively poor performance of translator generated machine code) is that addressed by DSP processors, which often contain such decrement (and/or increment) branch instructions (the SC140 DSP core[11] includes hardware loop counters that support up to four levels of loop nesting). The

  cache; translator performance vs. assembler; DSP processors

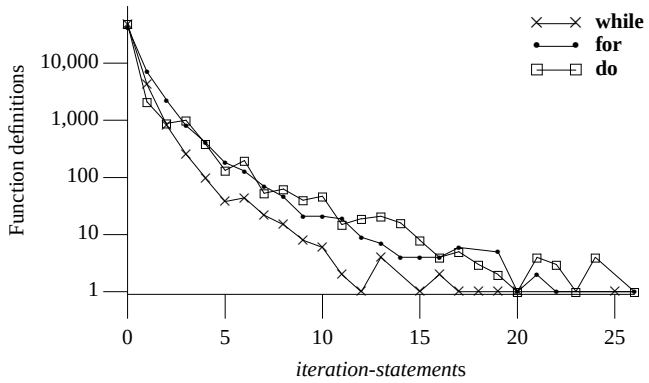

Figure 1763.1: Number of function definitions containing a given number of *iteration-statement*s. Based on the translated form of this book's benchmark programs.

C compiler for the Unisys e-@ction Application Development Solutions[18] uses the JGD processor instruction to optimize the loop iteration test. However, this usage limits the maximum number of loop iterations to $2^{35} - 2$, a value that is very unlikely to be reached in a commercial program (a trade-off made by the compiler implementors between simplicity and investing effort to handle very rare situations).

Obtaining an estimate of the execution time of a sequence of statements may require estimating the number of times an iteration statement will iterate. Some implementations provide a mechanism for the developer to provide iteration count information to the translator. For instance, the translator for the TMS320C6000[15] supports the following usage:

```
1  #pragma MUST_ITERATE (30) /* Will loop at least 30 times. */
```

Another approach is for the translator to deduce the information from the source.[8]

Program loops may not always be expressed using an *iteration-statement* (for instance, they may be created using a **goto** statement). Ramalingam[12] gives an algorithm for identifying loops in almost linear time.

Example

```
1  #include <stdio.h>
2
3  int f(unsigned char i, unsigned char j)
4  {
5  do
6  while (i++ < j)
7     ;
8  while (i   > j++)
9     ;
10
11 if (j != 0)
12    printf("Initial value of i was greater than initial value of j\n");
13 }
```

Usage

A study by Bodík, Gupta, and Soffa[2] found that 11.3% of the expressions in SPEC95 were loop invariant.

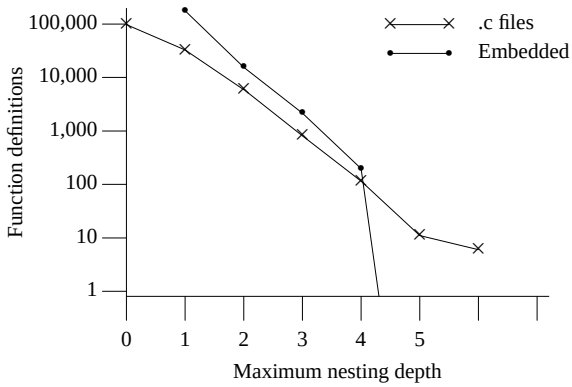

Figure 1763.2: Number of functions containing *iteration-statement*s nested to the given maximum nesting level; for embedded C[7] (whose data was multiplied by a constant to allow comparison) and the visible form of the `.c` files (zero nesting depth denotes functions not containing any *iteration-statement*s).

Table 1763.1: Occurrence of various kinds of **for** statement controlling expressions (as a percentage of all such expressions). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`); *assignment* is an assignment expression, *integer-constant* is an integer constant expression, and *expression* denotes expressions that contain arithmetic and shift operators. Based on the visible form of the `.c` files.

Abstract Form of **for** loop header	%
assignment **;** identifier < identifier **;** identifier **v++**	33.2
assignment **;** identifier < *integer-constant* **;** identifier **v++**	11.3
assignment **;** identifier **;** assignment	7.0
assignment **;** identifier < expression **;** identifier **v++**	3.3
assignment **;** identifier < identifier **;** **++v** identifier	2.7
; ;	2.5
assignment **;** identifier **!=** identifier **;** assignment	2.5
assignment **;** identifier **<=** identifier **;** identifier **v++**	2.2
assignment **;** identifier **>=** *integer-constant* **;** identifier **v--**	1.6
assignment **;** identifier < function-call **;** identifier **v++**	1.4
assignment **;** identifier < identifier **;** identifier **v++** , identifier **v++**	1.4
others	31.1

Table 1763.2: Occurrence of various kinds of **while** statement controlling expressions (as a percentage of all **while** statements). Where *object* is a reference to a single object, which may be an identifier, a member (e.g., `s.m`, `s->m->n`, or `a[expr]`); *assignment* is an assignment expression, *integer-constant* is an integer constant expression, and *expression* denotes expressions that contain arithmetic and shift operators. Based on the visible form of the `.c` files.

Abstract Form of Control Expression	%	Abstract Form of Control Expression	%
others	43.5	expression	2.2
object	12.2	***v** object	2.0
object **!=** object	7.0	assignment	1.8
integer-constant	6.2	**!** object	1.6
object < object	4.7	**!** function-call	1.3
function-call	4.4	object **!=** *integer-constant*	1.2
object > *integer-constant*	4.0	object **v--** > *integer-constant*	1.1
object **v--**	3.2	**!** expression	1.0
assignment **!=** object	2.4		

Constraints

1764 The controlling expression of an iteration statement shall have scalar type.

controlling expression
iteration statement

if statement
controlling expression
scalar type

Commentary

The issues here are the same as for controlling expressions in **if** statements.

Other Languages

Many languages do not support loop control variables having a pointer type (invariably because they do not support any form of pointer arithmetic).

Coding Guidelines

loop control variable

The concept of *loop control variable* is discussed elsewhere.

---

for statement
declaration part

The declaration part of a **for** statement shall only declare identifiers for objects having storage class **auto** or **register**. 1765

Commentary

loop control variable

The intent is to support the declaration of an identifier that is used as a loop control variable. There are many people who believe that limiting the scope over which such control variables can be modified is a *good thing*. Another coding guideline related issue is that of declaration of identifiers occurring close to where they are used in statements (this issue is discussed elsewhere).

declaration
syntax
statement
syntax

Problem

DR #277

Consider the code:

```
for (enum fred { jim, sheila = 10 } i = jim; i < sheila; i++)
     // loop body
```

Proposed Committee Response

The intent is clear enough; `fred`, `jim`, *and* `sheila` *are all identifiers which do not denote objects with* **auto** *or* **register** *storage classes, and are not allowed in this context.*

C90

Support for this functionality is new in C99.

C++

6.4p2 *The* `declarator` *shall not specify a function or an array. The* `type-specifier-seq` *shall not contain* **typedef** *and shall not declare a new class or enumeration.*

```
1  void f(void)
2  {
3  for (int la[10]; /* does not change the conformance status of the program */
4                   // ill-formed
5       ; ;)
6     ;
7  for (enum {E1, E2} le; /* does not change the conformance status of the program */
8                          // ill-formed
9       ; ;)
10    ;
11 for (static int ls; /* constraint violation */
12                     // does not change the conformance status of the program
13      ; ;)
14    ;
15 }
```

Other Languages

In some languages (e.g., Ada and Algol 68) the identifier used as a loop control variable in a **for** statement is implicitly declared to have the appropriate type (based on the type of the expressions denoting the start and end values).

Coding Guidelines

The ability to declare identifiers in this context is new in C99 and at the time of this writing there is insufficient experience with its use to know whether any guideline recommendation is worthwhile.

Semantics

1766 An iteration statement causes a statement called the *loop body* to be executed repeatedly until the controlling expression compares equal to 0.

iteration statement executed repeatedly loop body

Commentary

This defines the term *loop body*. The term *loop* is commonly used as a noun by developers to refer to constructs associated with iteration statements (which are rarely referred to as *iteration statements* by developers). For instance, the terms *loop statement*, or simply a *loop* are commonly used by developers.

Execution of the loop may also terminate because a **break**, **goto**, or **return** statement is executed. The discussion on the evaluation of the controlling expression in an **if** statement is applicable here.

if statement operand compare against 0

It is often necessary to access a block of storage (e.g., to copy it somewhere else, or to calculate a checksum of its contents). For anything other than the smallest of blocks the overhead of a loop can be significant.

Duff's Device

```
1  void send(register unsigned char *to,
2            register unsigned char *from,
3            register int count)
4  {
5  do
6      *to++ = *from++;
7  while (--count > 0);
8  }
```

The above loop requires a comparison after ever item copied. Unrolling the loop would reduce the number of comparisons per items copied. However, because `count` is not known at translation time an optimizer is unlikely to perform loop unrolling. The loop can be unrolled by hand, making sure that code also handles the situation where the number of items being copied is not an exact multiple of the loop unroll factor. A technique proposed by Tom Duff[6] (usually referred to as *Duff's device*) is (the original example used `*to`, i.e., the bytes were copied to some memory mapped serial device):

```
1  void send(register unsigned char *to,
2            register unsigned char *from,
3            int count)
4  {
5  register int n = (count+7)/8;
6
7  switch (count % 8)
8     {
9     case 0: do{   *to++ = *from++;
10    case 7:       *to++ = *from++;
11    case 6:       *to++ = *from++;
12    case 5:       *to++ = *from++;
13    case 4:       *to++ = *from++;
14    case 3:       *to++ = *from++;
15    case 2:       *to++ = *from++;
16    case 1:       *to++ = *from++;
17                } while (--n > 0);
18    }
19 }
```

C++

The C++ Standard converts the controlling expression to type **bool** and expresses the termination condition in terms of true and false. The final effect is the same as in C.

Other Languages

In many other languages the model of a *for loop* involves a counter being incremented (or decremented) from a start value to an end value, while the model of a *while loop* (or whatever it is called) being something that iterates until some condition is met. There is considerable overlap between these two models (it is always possible to rewrite one form of loop in terms of the other). The differences between the two kinds of loop are purely conceptual ones, created by developer loop classification models. Loop classification is often based on deciding whether a loop has the attributes needed to be considered a *for loop* (e.g., the number of iterations being known before the first iteration starts), all other loops being classified as *while loop*s. Early versions of Fortran performed the loop termination test at the end of the loop. This meant that loops always iterated at least once, even if the test was false on the first iteration.

Coding Guidelines

Some coding guideline documents recommend that loop termination only occur when the condition expressed in the controlling expression becomes equal to zero. A number of benefits are claimed to accrue from adhering to this recommendation. These include, readers being able to quickly find out the conditions under which the loop terminates (by looking at the loops controlling expression; which might only be a benefit for one form of reading) and the desire not to jump across the control flow. It is always possible to transform source code into a form where loop termination is decided purely by the control expression. However, is there a worthwhile cost/benefit to requiring such usage? The following example illustrates a commonly seen need to terminate a loop early:

reading
kinds of
jump state-
ment
syntax

```
1  #define SPECIAL_VAL 999
2  #define NUM_ELEMS 10
3
4  extern int glob;
5  static int arr[NUM_ELEMS];
6
7  void f_1(void)
8  {
9  for (int loop = 0; loop < NUM_ELEMS; loop++)
10    {
11    /* ... */
12    if (glob < NUM_ELEMS/2)
13       {
14       glob++;
15       if (arr[loop] == SPECIAL_VAL)
16          break;
17       }
18    else
19       arr[loop] = glob;
20    /* ... */
21    }
22 }
23
24 void f_2(void)
25 {
26 for (int loop = 0; loop < NUM_ELEMS; loop++)
27    {
28    /* ... */
29    if (glob < NUM_ELEMS/2)
30       {
31       glob++;
32       if (arr[loop] == SPECIAL_VAL)
33          loop = NUM_ELEMS;
```

```
34          }
35       else
36          arr[loop] = glob;
37
38       if (loop != NUM_ELEMS)
39          { /* ... */ }
40       }
41    }
42
43    void f_3(void)
44    {
45    _Bool terminate_early = 0;
46
47    for (int loop = 0; (loop < NUM_ELEMS) && !terminate_early; loop++)
48       {
49       /* ... */
50       if (glob < NUM_ELEMS/2)
51          {
52          glob++;
53          if (arr[loop] == SPECIAL_VAL)
54             terminate_early = 1;
55          }
56       else
57          arr[loop] = glob;
58
59       if (!terminate_early)
60          { /* ... */ }
61       }
62    }
```

Looking at the controlling expression in `f_1` and `f_2` it appears to be easy to deduce the condition under which the loop will terminate. However, in both cases the body of the loop contains a test that also effectively terminates the loop (in the case of `f_2` the body of the loop has increased in complexity by the introduction of an **`if`** statement). The function `f_3` handles the case where guidelines recommend against modifying the loop control variable in the loop body.

loop control variable

Any guideline recommendation needs to be based on a comparison of the costs and benefits of the loop constructs in these functions (and other cases). Your author knows of no studies that provide the information needed to make such a comparison. For this reason this coding guideline subsection is silent on the issue of how loops might terminate. A loop where it is known, at translation time, that the number of iterations is zero, is a loop containing redundant code. The issue of redundant code is discussed elsewhere.

redundant code

1767 The repetition occurs regardless of whether the loop body is entered from the iteration statement or by a jump.[DR268]

Commentary

This is a requirement on the implementation.

This sentence was added by the response to DR #268.

Other Languages

Many languages (e.g., Pascal, Ada) treat loop bodies as indivisible entities and do not permit a jump into them (although it is usually possible to jump out of them).

Coding Guidelines

Some coding guideline documents recommend against jumping into the body of a loop. One argument is that a reader of the source may not notice that a loop could be entered in this way and makes a modification that fails to take this case into account (i.e., introduces a fault).

There are a variety of situations where jumping into the body of a loop may result in code that is less likely to contain faults and be less costly to maintain (see the example given for the **`goto`** statement).

goto EXAMPLE

Jumping into the body of a loop is rare and no data is available on the kinds of faults in which it plays a significant contributing factor. For this reason a no guideline recommending is given.

---

block iteration statement

An iteration statement is a block whose scope is a strict subset of the scope of its enclosing block. 1768

Commentary

block selection statement

The rationale for this specification is the same as that given for the block implicitly created for a selection statement.

---

block loop body

The loop body is also a block whose scope is a strict subset of the scope of the iteration statement. 1769

Commentary

block selection sub-statement

The rationale for this specification is the same as that given for the block implicitly created for the substatements of a selection statement.

Example

In the following example the lifetime of the compound literal starts and terminates on every iteration of the loop.

```
1  struct S {
2           int mem_1;
3           int mem_2;
4           };
5
6  extern void g(struct S);
7
8  void f(void)
9  {
10 for (int i = 0; i < 10; i++)
11    g((struct S){.mem_1 = i, .mem_2 = 42});
12 }
```

DR268) Code jumped over is not executed. 1770

Commentary

EXAMPLE case fall through

This is a requirement on the implementation and is consistent with other situations where code is jumped over.

This sentence was added by the response to DR #268.

In particular, the controlling expression of a **for** or **while** statement is not evaluated before entering the loop body, nor is *clause-1* of a **for** statement. 1771

Commentary

iteration statement 1763 syntax

iteration statement 1766 executed repeatedly

While any expressions in the loop header are not executed when the loop body is entered, the controlling expression evaluation that occurs at start of the next, and any subsequent, iterations of the loop is executed.

This sentence was added by the response to DR #268.

References

1. Agere Systems. *DSP16000 Digital Signal Processor Core Information Manual*. Agere Systems, mn02-026winf edition, June 2002.
2. R. Bodík, R. Gupta, and M. L. Soffa. Complete removal of redundant computations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 1998.
3. T. W. Christopher. *Icon Programming Language Handbook*. Thomas W. Christopher, 1996.
4. K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 23(5):603–625, Sept. 2001.
5. Data General Corporation. *Programmer's Reference Manual: Nova Line Computers*. Data General Corporation, 2 edition, Sept. 1975.
6. T. Duff. Unwinding loops. *Newsgroup: netlang.c*, 1984-05-07 07:19:21 PST.
7. J. Engblom. Static properties of commercial embedded real-time and embedded systems. Technical Report ASTEC Technical Report 98/05, Uppsala University, Sweden, Nov. 1998.
8. C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, pages 121–148, May 2000.
9. IBM. *z/Architecture: Principles of Operation*. International Business Machines, first edition, Dec. 2000.
10. K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, Nov. 1999.
11. Motorola, Inc. *SC140 DSP Core Reference Manual*. Motorola, Inc, 2 edition, Apr. 2001.
12. G. Ramalingam. Identifying loops in almost linear time. *Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):175–188, 1999.
13. 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.
14. D. C. Suresh, S. R. Mohanty, W. A. Najjar, L. N. Bhuyan, and F. Vahid. Loop level analysis of security and network applications. In *Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-03)*, Feb. 2003.
15. Texas Instruments. *TMS320C6000 Programmer's Guide*. Texas Instruments, Inc, spru196d edition, Mar. 2000.
16. J. Torrellas, C. Xia, and R. L. Daigle. Optimizing the instruction cache performance of the operating system. *IEEE Transactions on Computers*, 47(12):1363–1381, 1998.
17. G.-R. Uh, Y. Wang, D. Whalley, S. Jinturkar, C. Burns, and V. Cao. Techniques for effectively exploiting a zero overhead loop buffer. In *Proceedings of the International Conference on Compiler Construction*, pages 157–172, Mar. 2000.
18. Unisys Corporation. *C Compiler Programming Reference Manual Volume 1: C Language and Library*. Unisys Corporation, 7831 0422■006, release level 8R1A edition, 2001.
19. J. Villarreal, R. Lysecky, S. Cotteral, and F. Vahid. A study of the loop behavior of embedded programs. Technical Report UCR-CSE-01-03, University of California, Riverside, Dec. 2001.