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.

6.8.5.3 The for statement

for statement

The statement 1774

for (*clause-1* **;** *expression-2* **;** *expression-3* **)** *statement*

behaves as follows:

Commentary

Rationale

In C89, for loops were defined in terms of a syntactic rewrite into **while** loops. This introduced problems for the definition of the **continue** statement; and it also introduced problems when the operands of cast and **sizeof** operators contain declarations as in:

```
enum {a, b};
{
    int i, j = b;
    for (i = a; i < j; i += sizeof(enum {b, a}))
        j += b;
}
```

not being equivalent to:

```
enum {a, b};
{
    int i, j = b;
    i = a;
    while (i < j) {
        j += b;                   // which b?
        i += sizeof(enum {b, a});  // declaration of b moves
    }
}
```

because a different b is used to increment i in each case. For this reason, the syntactic rewrite has been replaced by words that describe the behavior.

C90

Except for the behavior of a **continue** *statement in the loop body, the statement*

```
for ( expression-1 ; expression-2 ; expression-3 ) statement
```

and the sequence of statements

```
expression-1 ;
while (expression-2) {
statement ;
expression-3 ;
}
```

are equivalent.

C++

Like the C90 Standard, the C++ Standard specifies the semantics in terms of an equivalent **while** statement. However, the C++ Standard uses more exact wording, avoiding the possible ambiguities present in the C90 wording.

Other Languages

In most other languages the ordering of expressions puts the controlling expression last. Or to be more exact, an upper or lower bound for the loop control variable appears last. Most other languages do not support having anything other than the loop control variable tested against a value that is known at translation time. Some languages (e.g., Ada, Algol 68, and Pascal) do not allow the loop control variable to be modified by the body of the loop.

1774 loop control variable

Common Implementations

loop unrolling

Loop unrolling is the process of decreasing the number of iterations a loop makes by duplicating the statement in the loop body.[1] For instance:

```
for (loop = 0; loop < 10; loop++)
   {
   a[loop] = loop;
   }
/*
 * Can be unrolled to the following equivalent form:
 */
for (loop = 0; loop < 10; loop+=2)
   {
   a[loop] = loop;
   a[loop+1] = loop+1;
   }
```

Loop unrolling reduces the number of jumps performed (which can be a significant saving when the loop body is short) and by increasing the number of statement in the loop body creates optimization opportunities (which, in the above example, could result in two loop bodies executing in less time than twice the time for a single iteration). When the iteration count is not exactly divisible by the loop body unrolling factor copies of the loop body may need to occur before the start, or after the end, or the loop statement.

At the minimum, loop unrolling requires knowing the number of loop iterations and the amount by which the loop control variable is incremented, at translation time. Implementations often place further restrictions on loops before that they unroll (requiring the loop body to consist of a single basic block is a common restriction).

Arbitrary amounts of loop unrolling (e.g., iterating 10 times over 100 copies of a loop body where the original is known to iterate 1000 times) does not necessarily guarantee improved performance. Duplicating the loop body increases code size, which decreases the probability that all of the loop body instructions will fit within the processor's instruction cache. Unless optimizers take into account the size of a processor's instruction cache when evaluating the cost effectiveness of loop unrolling they can end up reducing, rather than increasing, program performance.[2]

iteration statement syntax

Jinturkar[3] analyzed the loops in a set of benchmarks to determine the complexity and size of loop bodies, and the nature of the loop bounds. The results were that in 50% of loops the iteration count could be deduced at translation time and that the generated machine code for the loop bodies of each loop that were unrolled was smaller than 256 bytes.

Coding Guidelines

Writers of coding guideline documents often regard the components of a **for** statement as having attributes that other loop statements don't have (e.g., they have an associated loop control variable). While it can be argued that many of these authors have simply grafted onto C concepts that only exist in other languages (or perhaps the encapsulation all of the loop control information in one visually delimited area of the source

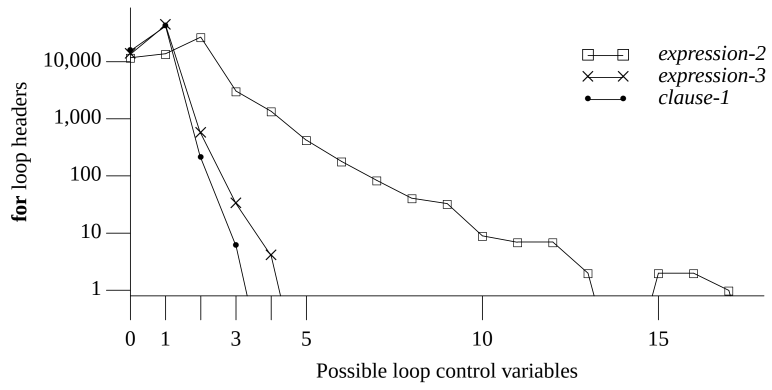

Figure 1774.1: Number of possible loop control variables appearing in *`expression-2`* (square-box) after filtering against the objects appearing in *`expression-3`* (cross) and after filtering against the objects appearing in *`clause-1`* (bullet). Based on the visible form of the `.c` files.

triggers a cognitive response that triggers implicit assumptions in readers), if a sufficient number of developers associate these attributes with **`for`** statements then they become part of the culture of C and need to be considered here. Other loop conceptualization issues are discussed elsewhere.

while statement

loop control variable

This subsection discusses one attribute commonly associated with **`for`** statements that is not defined by the C Standard, the so-called *loop control variable* (or simply *loop variable*, or alternatively *loop counter*). A loop control variable is more than simply a concept that might occur during developer discussion, many coding guideline documents make recommendations about its use (e.g., a loop control variable should not be modified during execution of the body of the loop, or have floating-point type). Which of the potentially four different objects that might occur, for instance, in the most common form of loop header (see Table **??**) is the loop control variable?

MISRA

```
for (lcv_1=0; lcv_2 < lcv_3; lcv_4++)
```

The following algorithm frequently returns an answer that has been found to be acceptable to developers (it is based on the previous standard and has not been updated to reflect the potential importance of objects declared in *`clause-1`*). Note that the algorithm may return zero, or multiple answers; a union or structure member selection operator and its two operands is treated as a single object, but both an array and any objects in its subscript are treated as separate objects and therefore possible loop control variables:

1. list all objects appearing in *`expression-2`* (the controlling expression). If this contains a single object, it is the loop control variable (33.2% of cases in the `.c` files),
2. remove all objects that do not appear in *`expression-3`* (which is evaluated on every loop iteration). If a single object remains, that is the loop control variable (91.8% of cases in the `.c` files),
3. remove all objects that do not appear in *`clause-1`* (which is only evaluated once, prior to loop iteration). If a single object remains, that is the loop control variable (86.2% of cases in the `.c` files).

Unlike the example given above, in practice the same object often appears as an operand somewhere within all three components (see Figure 1774.1).

Because the controlling expression is evaluated on every iteration of the loop, the loop control variable can appear in contexts that are not supported in other languages (because most evaluate the three loop components only once, prior to the first iteration). For instance:

```
for (lcv_1=0, lcv_2=0; a1[lcv_1] < a2[lcv_2]; lcv_1++, lcv_2+=2)
```

Experience shows that developers often assume that, in a **for** statement, modification of any loop control variables only occurs within the loop header. This leads to them forming beliefs about properties of the loop, for instance, *it loops 10 times*. There tend to be fewer assumptions made about the use of **while** statements (which might not even be thought to have a loop control variable associated with them) and the following guideline is likely to cause developers to use this form of looping construct.

Cg 1774.1 A loop control variable shall not be modified during the execution of the body of a **for** statement.

Some coding guideline documents recommend that loop control variables not have floating-point type. It might be thought that such a recommendation only makes sense in languages where the loop termination condition involves an equality test (in C this case is covered by the guideline recommendation dealing with the type of the operands of the quality operators). However, the controlling expression in a C **for** statement can contain relational operators, which can also have a dependence on the accuracy of floating-point operations. For instance, it is likely that the author of the following fragment expects the loop to iterate 10 times. However, it is possible that 10 increments of `i` result in it having the value 9.9999, and loop termination not occurring until after the eleventh iteration.

?? equality operators not floating-point operands

```
for (float i=0.0; i < 10.0; i++)
```

A possible developer response to a guideline recommendation that loop control variables not have floating point type is to use a **while** statement (which are not covered by the algorithm for deducing loop control variables). Some of the issues associated with the finite accuracy of operations on floating-point values can be addressed with guideline recommendations. However, the difficulty of creating wording for a recommendation dealing with the use of floating-point values to control the number of loop iterations is such that none is attempted here.

Table 1774.1: Occurrence of sequences of components omitted from a **for** statement header (as a percentage of all **for** statements). Based on the visible form of the `.c` files.

Components Omitted	%
clause-1	3.8
clause-1 expr-2	0.1
clause-1 expr-2 expr-3	2.5
clause-1 expr-3	0.1
expr-2	0.8
expr-2 expr-3	0.2
expr-3	1.6

1775 The expression *>expression-2* is the controlling expression that is evaluated before each execution of the loop body.

controlling expression for statement

Commentary

The loop body of a **for** statement may be executed zero or more times.

Other Languages

In many languages the termination condition of a **for** statement, specified in the header of the loop body, is fixed prior to the first iteration of the loop body (every time the loop statement is encountered). The commonly used termination condition being that the loop counter be equal to the value of a specified expression.

Coding Guidelines

The controlling expression in a **for** statement is sometimes written so that its evaluation also has the side effect of modifying the value of the loop control variable, removing the need for *expression-3*. A developer may have any of a number of reasons for using such an expression, from use of an idiom to misplaced concern for efficiency (many of the issues associated with side effects within the controlling expression are the same as those that apply to **while** statements). while statement

---

The expression *expression-3* is evaluated as a void expression after each execution of the loop body. 1776

Commentary

The common usage is for the evaluation of this expression to modify the value of the loop control variable.

Other Languages

In many languages the value used to increment/decrement the loop counter of a **for** statement is fixed (every time the **for** statement is encountered) prior to the first iteration of the loop body.

If *clause-1* is a declaration, the scope of any ~~variables~~ identifiers it declares is the remainder of the declaration and the entire loop, including the other two expressions; 1777

Commentary

loop body

The phrase *entire loop* means *expression-2*, *expression-3*, and the loop body.

The wording was changed by the response to DR #292.

C90

Support for this functionality is new in C99.

Other Languages

In some languages (e.g., Ada and Algol 68) the occurrence of an identifier as the loop control variable also acts as a definition of that identifier (its type is that of the controlling expressions).

Coding Guidelines

Some coding guideline documents warn of the dangers of accessing loop control variables outside of the loops they control. One of the reasons for this is that some languages do not define the value of this variable once the loop has terminated (which makes such accesses equivalent to those of an uninitialized variable). Loop control variables in C do not have this behavior and some form of related guideline recommendation is not required.

Declaring the loop control variable via *clause-1* has the benefit of localizing the visual context over which it is referenced. Possible costs include having to modify existing habits (e.g., looking for the declaration at the start of the function body) and possible lack of support for constructs new in C99 by ancillary tools. block iteration statement

Example

```
void three_different_objects_named_loc(void)
{
int loc = 20;

for (int loc = 0; loc < 10; loc++)
   int loc = -8;
}
```

---

it is reached in the order of execution before the first evaluation of the controlling expression. 1778

Commentary

This requirement is necessary because it is intended that objects declared in *clause-1* appear in the controlling expression.

1779 If *clause-1* is an expression, it is evaluated as a void expression before the first evaluation of the controlling expression.[134)]

Commentary

This evaluation occurs once, every time the **for** statement head of the loop is encountered in the flow of control.

1780 Both *clause-1* and *expression-3* can be omitted.

Commentary

Saying in words what is specified in the syntax. A **for** statement loop header is essentially a means of visually highlighting the various components of a loop.

C++

The C++ Standard does not make this observation, that can be deduced from the syntax.

Other Languages

Being able to omit the specification for the initial value of a loop counter (i.e., *clause-1*) is unique to C (and C++). Most languages allow their equivalent of *expression-3* to be omitted and use a default value (usually either 1 or -1).

Coding Guidelines

Why would a developer choose to omit either of these constructs in a **for** statement, rather than using a **while** statement? This issue is discussed elsewhere.

iteration statement syntax
while statement

1781 An omitted *expression-2* is replaced by a nonzero constant.

Commentary

Specifying that an omitted *expression-2* is replaced by a nonzero constant allows a more useful meaning to be given to those cases where *clause-1* or *expression-3* are present, than by replacing it by the constant 0. Omitting *expression-2* creates a loop that can never terminate via a condition in the loop header. Executing a **break**, **goto**, or **return** statement (or a call to the longjmp library function) can cause execution of the loop to terminate. The term *infinite loop* is often used to describe a **for** statement where the controlling expression has been omitted. In some freestanding environments the main body of a program consists of an infinite loop that is only terminated when electrical power to the processor is switched off.

Other Languages

Most languages require that the loop termination condition be explicitly specified. In Ada the loop header is optional (a missing header implies an infinite loop).

Common Implementations

The standard describes an effect that most implementations do not implement as stated. A comparison that is unconditionally true can be replaced by an unconditional jump.

Coding Guidelines

There is an idiom that omits *expression-2* when an infinite loop is intended (usually omitting the other two expressions as well). Does such an idiom have a more worthwhile cost/benefit than using a **while** statement, with a nonzero constant as the controlling expression? Existing source code contains both usages (see Table **??**, Table **??**) and given practice readers will learn to automatically recognize both forms. However, such automatic recognition takes time to learn and a **while** statement whose controlling expression is a nonzero constant probably requires less effort to comprehend (because it is not an implicit special case) for less experienced developers.

Example

```
#define TRUE 1

void f(void)
{
for (;;)
   { /* ... */ }

while (TRUE)
   { /* ... */ }
}
```

References

1. J. W. Davidson and S. Jinturkar. An aggressive approach to loop unrolling. Technical Report CS-95-26, University of Virginia, June 1995.
2. S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, July 1999.
3. S. Jinturkar. *Data-Specific Optimizations*. PhD thesis, University of Virginia, May 1996.