The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.5.17 Comma operator

comma operator syntax

1313

expression:
assignment-expression
expression , *assignment-expression*

Commentary

This operator is frequently seen in automatically generated source code. It is usually used to simplify the analysis that needs to be performed by the generator in deducing what it has to do. For instance, when generating an expression it is not necessary to look ahead to see if some other expressions need to be evaluated first. If such an expression is encountered, it can occur as the left operand of a comma expression (which can occur in any context an expression can occur in).

Other Languages

The comma operator is unique to C (and C++). However, some languages provide a mechanism for grouping multiple statements into an expression that returns a value.

Common Implementations

compound expression

gcc supports what it calls *compound expressions*. The following example— the compound expression is delimited by ({ and }) and its value is that of the last expression in the compound— could be written using comma and conditional operators (however, it is not possible to define local objects within these operators).

```
1  extern int foo(void);
2
3  void f(void)
4  {
5  int loc;
6
7  loc =({
8       int x = foo (),
9           y;
10      if (x > 0)
11         y = x;
12      else
13         y = -x;
14      y;
15      });
16 }
```

Compound expressions do not offer much benefit in the visible source, except for automatically generated code. However, the ability to define objects can be useful in macro definitions. The following definition is written so that its arguments are only evaluated once.

```
1  #define MAXINT(x,y) ({int _x = (x), _y = (y); _x > _y ? _x : _y; })
```

Coding Guidelines

The comma token is one of the visibly smallest characters (in terms of display area occupied) and usually appears among other characters (unlike the semicolon token, which usually appears at the end of a line). Although specialist fonts have been designed for a variety of domains (e.g., mathematics, linguistics), none have yet become generally available for displaying source code. Increasing the visible size of the comma character is only one of several display issues that could be addressed.

Most occurrences of the comma token in the visible source are as a punctuator (e.g., an argument separator in function calls, or a list of identifiers in a declaration), not as an operator. The C syntax requires that any use of this token as an operator in an argument expression be enclosed in parentheses.

EXAMPLE 1319 comma operator

Semantics

1314 The left operand of a comma operator is evaluated as a void expression;

comma operator
left operand

Commentary

The intent is for the left operand to generate side effects, but for any value it might have, not to take part in forming the result of the expression that contains it.

Coding Guidelines

The purpose of the left operand is to generate side effects without contributing to the value of the containing expression. As such it requires readers to make a cognitive task switch between comprehending the expression and updating their mental model (based on the consequences of the side effects). Is the cost/benefit to using a comma operator more worthwhile than separating its components over two statements (the semicolon after the first statement provides a sequence point)? There are three contexts in which an expression can appear:

cognitive switch

1. *An initializer.* Initialization is a construct where C99 offers a solution not available in C90.

```
1  extern int glob;
2
3  void f(void)
4  {
5  int loc_1 = (--glob , glob + 3);
6
7  /*
8   * Only possible in C99.
9   */
10 glob--;
11 int loc_2 = glob + 3;
12
13 /*
14  * A possible rewriting in C90.
15  */
16 int dummy = glob--;
17 int loc_3 = glob + 3;
18 }
```

   Recommending that a comma operator in an initializer be replaced by a statement and a declaration limits portability by requiring a C99 translator. Also given that developers are not yet accustomed to seeing statements before declarations, it is possible that such statements will be overlooked.

2. *An expression statement.* There are no obvious benefits to using a comma operator in an expression statement. The single sequence point guarantee can be obtained using other constructs (although developers sometimes make incorrect assumptions about the sequencing of other operand evaluations).

1315 comma operator sequence point

3. *A controlling expression.* The use of the comma operator in this context is discussed under the cases in which they can occur, selection statements and iteration statements.

selection statement syntax
iteration statement syntax

Until C99 translators become more generally available, the following guideline recommendation limits itself to expression statements.

Cg 1314.1 The comma operator shall not appear in the visible form of an expression statement.

If the left operand of the comma operator does not generate a side effect, it is redundant code.

redundant code

1315 there is a sequence point after its evaluation.

comma operator
sequence point

Commentary

This sequence point guarantees that any side effects that occur in the evaluation of the left operand will have taken place before the right operand is evaluated. There is no other guarantee given for the order of evaluation of any other operands that may occur within the full expression containing the comma operator.

C++

5.18p1 *All side effects (1.9) of the left expression, except for the destruction of temporaries (12.2), are performed before the evaluation of the right expression.*

function call
sequence point

The discussion on the function-call operator is applicable here.

Common Implementations

Sequence points cut down on the opportunities for optimization, or at least make them harder to find. Long stretches of source code without sequence points provide an optimizer maximum flexibility in ordering the sequence of generated machine code. For this reason translators prefer to evaluate the left operand of comma operators as soon as possible (in some case they may evaluate it as late as possible, but that is less commonly optimal). In:

```
1  extern int glob_1,
2             glob_2;
3  extern int g(void);
4
5  void f(void)
6  {
7  int loc[4];
8  /* ... */
9  loc[++glob_1] = (glob_2 * loc[0]) + (g() , glob_2) + (loc[3] * 3);
10 }
```

the only requirement placed on the call to g is that it occurs before the right operand of the comma operator is evaluated. Calling g after any of the other operands had been evaluated would require saving their intermediate results. Looking at the visible source code, there are obvious optimization advantages to calling g before any other operands are evaluated. But then a more detailed analysis may show that the other operands are already available in registers, and ready to be used, in which case it may be more efficient to call g as late as possible.

Coding Guidelines

An incorrect assumption sometimes made by developers is to assume that all operands to the left of a comma operator will be evaluated (and side effects have occurred) before any of the operands to its right are evaluated. This issue is covered by the guideline recommendation dealing with sequence points.

sequence ?? points
all orderings give same value

Then the right operand is evaluated; 1316

Commentary

The word *then* is sometimes incorrectly interpreted to mean that the right operand is evaluated immediately after the evaluation of the left operand; that is, no other operand is evaluated between the left and right operand evaluations. As the following example illustrates, other operands may be evaluated between the evaluations of the two operands of a comma operator.

```
1  extern int glob_1,
2             glob_2;
3  extern int g(void),
4             h(void); /* Writing extern int h(void); would be idiomatic. */
5
6  void f(void)
7  {
8  int loc_1;
9  /*
10  * After glob_1 is incremented there is no guarantee that g will be
```

```
 * called before glob_2 is incremented, or that after glob_2 is
 * incremented that h will be called before glob_1 is incremented.
 */
loc_1 = (glob_1++ , g()) + (glob_2++ , h());
}
```

1317 the result has its type and value.[95)]

Commentary

The right operand has the same type and value as if it had not occurred as an operand of the comma operator.

1318 If an attempt is made to modify the result of a comma operator or to access it after the next sequence point, the behavior is undefined.

Commentary

This specification is consistent with the right operand being the result of a binary operator. However, it would also have been possible to treat this operand as if it had not been operated on, but the Committee chose not to make that decision.

C90

This sentence did not appear in the C90 Standard and had to be added to C99 because of a change in the definition of the term lvalue. lvalue

C++

The C++ definition of lvalue is the same as C90, so this wording is not necessary in C++. lvalue

Common Implementations

Implementations are unlikely to use a temporary to store the value of the right operand. They will refer directly to the right operand.

Coding Guidelines

The code that needs to be written to generate this behavior is sufficiently obscure and unlikely to occur that no guideline recommendation is given here.

Example

```
struct {
        int m[10];
       } x;

void f(void)
{
int *p = &((x , x).m[2]);

*p = 1;
p = &(x.m[2]);
*p = 1;
}
```

1319 EXAMPLE EXAMPLE comma operator

As indicated by the syntax, the comma operator (as described in this subclause) cannot appear in contexts where a comma is used to separate items in a list (such as arguments to functions or lists of initializers). On the other hand, it can be used within a parenthesized expression or within the second expression of a conditional operator in such contexts. In the function call

```
f(a, (t=3, t+2), c)
```

the function has three arguments, the second of which has the value 5.

Commentary

controlling expression
if statement

Using an assignment operator in contexts other than an expression statement is discussed elsewhere.

---

footnote 95
comma operator
lvalue

95) A comma operator does not yield an lvalue. 1320

Commentary

This is the one significant difference between an occurrence of the right operand on its own and as an operand of a comma operator. The former case could be an lvalue while the latter is not.

lvalue
converted to value

C++

5.18p1 *. . . ; the result is an lvalue if its right operand is.*

```
#include <stdio.h>

void DR_188(void)
{
char arr[100];

if (sizeof(0, arr) == sizeof(char *))
   printf("A C translator has been used\n");
else
   if (sizeof(0, arr) == sizeof(arr))
      printf("A C++ translator has been used\n");
   else
      printf("Who knows why we got here\n");
}

void f(void)
{
int loc;

(2, loc)=3; /* constraint violation */
            // conforming
}
```

Forward references: initialization (6.7.8). 1321

References