

Mastering Advanced C

Rex Jaeschke

Copyright © 1985–1994 Rex Jaeschke. All rights reserved.
Edition: 2.0
Printing: November 25, 1999

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means whatsoever, except in the case of brief quotations embodied in critical reviews and articles.

The information in this book is subject to change without notice and should not be construed as a commitment by the author or the publisher. Although every precaution has been taken in the preparation of this book, the author and publisher assume no responsibility for errors or omissions.

Printed in the United States of America.

The training materials associated with this book are available for license. These materials include a manuscript master, overhead transparencies, quick reference guides, instructor guide, and lab solutions in electronic format. Interested parties should contact the author at the address below.

Please address comments and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
(703) 860-0091
rex@RexJaeschke.com

This book was typeset by the author using the T_EX typesetting package, L^AT_EX macros, and Makeindex indexing tool.

Contents

I	Environment	1
1	Command-Line Argument Processing	3
1.1	Introduction	3
1.2	Accessing Command-Line Arguments	3
1.3	Spawning Programs	5
1.4	Environment Variables	6
2	Numerical Limits	9
2.1	Introduction	9
2.2	limits.h	10
2.3	float.h	11
II	Language	13
3	Operators and the Precedence Table	15
3.1	The Precedence Table	15
3.2	Primary Operators	17
3.3	Unary Operators	18
3.4	Other Operators	19
3.5	Miscellaneous Issues	20
4	Sequence Points	21
4.1	Introduction	21
4.2	Full Expressions	23
4.3	Sequence Point Operators	24
4.4	Conclusion	26
5	Lvalues	27
5.1	Introduction	27
5.2	Two Kinds of Lvalues	28
5.3	Operators That Generate Lvalues	28
5.4	Operators That Need Modifiable Lvalues	29
6	The Comma Operator	31
6.1	Syntax and Semantics	31
6.2	An Application	32
6.3	A Function Trace Facility	35
7	Type Qualifiers	43
7.1	The <code>const</code> Qualifier	43
7.2	The <code>volatile</code> Qualifier	47
7.3	Using <code>const</code> and <code>volatile</code> Together	48
7.4	Miscellaneous Issues	48

8	Generic Pointers	51
8.1	Typed Pointers	51
8.2	Pointer Representation	52
8.3	The Generic Pointer Type	52
8.4	Standard Headers	54
8.5	Type Qualifiers	55
8.6	Tracking Different Pointer Types	55
9	Pointers to Functions	59
9.1	Introduction	59
9.2	An Example	60
9.3	Jump Tables	61
9.4	Miscellaneous Issues	62
10	Pointers to Arrays	65
10.1	Introduction	65
10.2	Dynamic Allocation of Multidimensional Arrays	66
10.3	An Abstraction Tool	68
11	Enumerated Data Types	71
11.1	Introduction	71
11.2	Internal Representation	71
11.3	Lack of Type/Range Checking	73
11.4	Miscellaneous Issues	73
12	Type Synonyms	75
12.1	Defining Synonyms	75
12.2	Standard Type Synonyms	77
12.3	Abstract Types and I/O	77
12.4	<code>typedef</code> versus <code>#define</code>	78
13	Function Prototypes	81
13.1	Introduction	81
13.2	Assignment Compatibility	83
13.3	Pointers to Functions	84
13.4	Argument Names	84
13.5	Prototypes and Macros	85
13.6	Type Qualifiers	85
13.7	New-Style Function Definitions	86
13.8	Argument Widening	86
14	Mastering Declarations	89
14.1	Introduction	89
14.2	Base Types	89
14.3	Deriving From a Base Type	89
14.4	Deriving from Derived Types	90
14.5	Precedence of Punctuators	91
14.6	Forcing Punctuator Precedence	91
14.7	Writing Declarations	92
14.8	Reading a Declaration	93
14.9	Using Type Information	94
III	Preprocessor	97
15	Preprocessor Operators and Directives	99
15.1	Operators	99
15.2	Directives	101

16 Header Design and Management	103
16.1 Header Categories	103
16.2 Header Names	104
16.3 Header Contents	104
16.4 Protecting Header Contents	104
16.5 Conditional Inclusion	106
16.6 Planning for Debugging and Maintenance	106
16.7 Concatenating Headers	107
16.8 Conclusion	107
IV Library	109
17 Signal Handling	111
17.1 Introduction	111
17.2 An Example	113
17.3 Atomic and Non-Atomic Objects	114
17.4 Portability and Extensions	115
17.5 Reentrancy	115
17.6 SIG_DFL Handling	116
17.7 SIG_IGN Handling	116
17.8 Handler Requirements and Limitations	116
17.9 Critical Sections	117
18 Program Termination	119
18.1 Introduction	119
18.2 <code>abort</code> versus <code>exit</code>	119
18.3 Registering an Exit Handler	120
18.4 Framework for an Application	122
18.5 Intercepting Aborts	124
18.6 Miscellaneous Issues	125
19 Non-Local Jumps	127
19.1 Introduction	127
19.2 Some Examples	127
19.3 Program Context	133
19.4 Miscellaneous Issues	133
20 Variable-Length Argument Lists	135
20.1 Introduction	135
20.2 Implementing a Maximum Function	136
20.3 The <code>va_*</code> Routines	138
21 Sorting and Searching	141
21.1 Sorting	141
21.2 Searching	149
V Miscellaneous	153
22 Mixed-Language Programs	155
22.1 Introduction	155
22.2 Environmental Issues	156
22.3 Data Representation	157
22.4 Inter-Procedural Communication	159
22.5 I/O	161
22.6 Library Issues	161

23 Lab Exercises	163
23.1 Problem #1:	163
23.2 Problem #2:	163
23.3 Problem #3:	164
23.4 Problem #4:	164
23.5 Problem #5:	164
23.6 Problem #6:	166
23.7 Problem #7:	167
23.8 Problem #8:	167
23.9 Problem #9:	168
23.10 Problem #10:	168
23.11 Problem #11:	168
23.12 Problem #12:	169
23.13 Problem #13:	169
VI Appendices	171
A Operator Precedence	173
B Solutions to Exercises	175
B.1 Problem #1:	175
B.2 Problem #2:	178
B.3 Problem #3:	182
B.4 Problem #4:	183
B.5 Problem #5:	184
B.6 Problem #6:	186
B.7 Problem #7:	187
B.8 Problem #8:	188
B.9 Problem #9:	189
B.10 Problem #10:	190
B.11 Problem #11:	191
B.12 Problem #12:	193
B.13 Problem #13:	194
Index	197

Chapter 4

Sequence Points

Many expressions in C contain side effects and programmers are often confused about exactly when these side effects are actually executed by a compiler. In this chapter we shall learn just when these occur.

4.1 Introduction

Most operators in C have only one purpose, to compute a result. And in the process of computing that result, the operands themselves remain unchanged. For example, in the expression:

```
a + b - c * d
```

the value of each variable is used to compute a result. However, the value of each variable stays the same. One case in which this is not true, is assignment. Consider the commonly-used piece of code to read characters until end-of-file is reached:

```
if ((c = getchar()) != EOF)
```

Unlike most other languages, C defines assignment via an operator, not a punctuator. And because operators produce results, an assignment expression has a type and value. By definition, the type of such an expression is the type of its left operand and the value is that of the left operand after the assignment has taken place. So the assignment operator, be it a simple or compound assignment, actually has two purposes: It results in a value being computed and it causes the value of the right operand to be stored into the location designated by the left operand. The storing of the value is called a *side effect*. That is, when we perform an assignment, not only do we get an answer, we also get a change in the program's environment along the way. Let's look at a more common example of an expression containing a side effect:

```
a = b;
```

In this case, the value computed is not used; we evaluate the expression simply for the purpose of getting the side effect. This is very common in C programs. For example:

```
/*1*/ for (i = 0; exp ; i++) { /* ... */ }
/*2*/ ++i;
/*3*/ j++;
/*4*/ printf("Hello\n");
```

In case 1, the first and third expression are evaluated only for their side effects. Similarly, in cases 2 and 3 the resulting values are also discarded. As a result, it is purely a matter of style whether we use pre- or postfix notation because, in either case, the side effects are the same. In case 4, the `int` value returned is discarded.

One commonly used form of statement is called an *expression statement*. This is simply an expression followed by a semicolon. The following are valid expression statements:

```
6;
i + 6;
i/j;
sqrt(1.2);
isupper(c);
```

However, none of these expressions contains a side effect. And since we don't use the values computed, the statements are completely useless. Whether our implementation draws our attention to this is a matter of "quality of implementation;" There certainly is no error as far as Standard C is concerned. The rule then is that if an expression has no side effects and its value is never used, it is a useless expression.

C has a number of operators that always produce side effects. They are: all assignment operators, ++, and --. Calling a function that changes the environment by doing I/O, clearing the screen, generating an interrupt, or modifies a static object, also produces a side effect. However, many functions do not produce a side effect. For example, the character testing functions in `<ctype.h>` simply produce a result.

Most languages do not have operators that can produce a side effect. As a result many constructs permissible in C simply cannot exist in other languages. For example:

```
x[i] = y[i++]
a = i + ++i
a = b/(--b + c)
```

The question now becomes "Just when are these side effects done? Are the outcomes of these expressions predictable and useful?" To answer these questions we need to know about sequence points.

The C standard contains the following relevant statements:

"The semantic descriptions in this Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant."

"Accessing a `volatile` object, modifying an object, modifying a file, or calling a function that does any of those operations are all 'side effects', which are changes in the state of the execution environment. Evaluation of an expression may produce side effects. At certain specified points in the execution sequence called 'sequence points', all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place."

"An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object)."

"When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on. Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet."

What does this mean to mere mortals? Simply stated, a *sequence point* is a place in a program at which the compiler has actually done what it promised to do. That is, all side effects promised up to that point have actually been completed. Note though a compiler is permitted to not perform side effects if it can deduce they are not needed. This comes under the broader rule established by the C Standard. An implementation can perform any optimization it wants provided it produces the same result as that required by the Standard. That is, as if it had not performed the optimization. This is known as the *as if* rule.

Consider the following example:

```
main()
{
    int *f1(void);
    int f2(void);
    int f3(void);

    *f1() = f2() + f3();
}
```

There is a sequence point at the end of the assignment expression. That is, at the semicolon, we are guaranteed that all three functions have been called and that the object pointed to by `f1()` has been modified. However, the

order in which the functions are called is not guaranteed¹. The reason for this is that the order of evaluation of operands of both the addition and assignment operators, is undefined. So even though the implied grouping of terms in the assignment is:

```
(*f1() = (f2() + f3()));
```

that tells us nothing about the order of evaluation of each term. In this regard the optimizer has complete flexibility. If the three functions are some how inter-related—perhaps they communicate through global variables—we may well care about the order in which they are called. To completely define this as being left-to-right, for example, we would have to rewrite the assignment as follows:

```
main()
{
    int *f1(void);
    int f2(void);
    int f3(void);
    int *pi;

    pi = f1();
    *pi = f2();
    *pi += f3();
}
```

Many languages guarantee the order of evaluation of operands for each operator. However, in doing so they force the compiler to do it “their way” always, whether or not that is optimal. In C, if we don’t care about the order the optimizer can do the best it can and, if we do care, we have a way to specify so.

4.2 Full Expressions

Earlier, it was stated there was a sequence point at the end of the assignment expression. That was because that expression was a full expression. A *full expression* is an expression that is not part of another expression. Therefore, all expression statements—expressions followed by a statement-terminating semicolon—have sequence points at their end. For the most part, this should be intuitive since we would be unpleasantly surprised if in `i = 10; i` did not actually get the value 10 assigned to it before the next statement—`j = i;`, for example—were executed. Note though, that `i` need not be assigned any value if the compiler can determine `i` is not used later in the program.

The complete set of full expressions is: an initializer; the expression in an expression statement; the controlling expression of a `do`, `if`, `switch`, or `while` statement; each of the three optional expressions of a `for` statement; and the optional expression in a `return` statement. Examples of each of these follow:

¹There is also a sequence point before each function is called but that is of no importance here, as we shall explain later.

```

test()
{
    int f(int);
    extern int i, j;
/*1*/  int k = f(6), m = k;

/*2*/  i--;

/*3*/  if (i++)
        j = f(i);

/*4*/  for (i = 5; (f(i) + i) < j--; i++)
        /* ... */;

/*5*/  k++;

/*6*/  return (i++ + f(j));
}

```

Admittedly, this example is contrived and is not intended to do anything useful. However, it adequately demonstrates the point.

In case 1, `f` must be called before `m` can be initialized so `m` takes on a predictable value. In case 2, `i` must be decremented before we move on to the next statement. In case 3, whether or not `i` tests true, `i` must be incremented before we drop into the statement body or to the next statement.

In case 4, there is a sequence point at the end of each of the three full expressions. Therefore, `i` must be assigned the value 5 before the controlling expression is evaluated. Then, `f` must be called and `j` decremented before we drop into the statement body or through to the next statement. And at the end of a loop iteration, `i` must be incremented before the controlling expression is reevaluated.

Case 5 can be discarded altogether because `k` is not used beyond this point and, being an automatic object, it is reinitialized when `test` is next called. The compiler could also discard the variables `k` and `m` completely as they are never used. However, it would still have to evaluate the initializer for `k` as it might contain a side effect.

Finally, in case 6, `i` must be incremented and `f` must be called before we return to the calling function.

`i` and `j` are both global variables. If `f` actually relies on the values of these when it executes—either by reading and/or updating them—the outcome of this program would be undefined since the order of evaluation in two critical expressions, is undefined. For example:

```

/*4*/  for (i = 5; (f(i) + i) < j--; i++)
        /* ... */;

/*6*/  return (i++ + f(j));

```

In the controlling expression in case 4, the order of evaluation of the operands of the less-than operator is undefined. Therefore, `j` might or might not be decremented before function `f` is called. Also, if function `f` modifies `i`, the value of the expression `f(i) + i` is undefined. Similarly, in the return statement the order of evaluation of operands to the addition, is undefined and `i` might or might not be incremented before `f` is called.

4.3 Sequence Point Operators

While most of C's operators give no guarantee about the order of evaluation of their operands, a few do and these will be identified and discussed here.

The logical AND operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand tests false, the second operand is not evaluated. Therefore, the following expressions are well defined:

```

(i = f()) && (i++ < j)
(i++ - 10) && (i != f())
f() && g()

```

The logical OR operator also guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand tests true, the second operand is not evaluated. Therefore, the following expressions are well defined:

```
(i = f()) || (i++ < j)
(i++ - 10) || (i != f())
f() || g()
```

The conditional operator (`?:`) has a sequence point after the first operand is evaluated. Therefore, the following expressions are well defined:

```
++i ? f(i) : j = i
g() ? i++ : h()
```

However, the expression:

```
((i > j) ? i++ : i = 3) + i
```

is not since there is no sequence point at the end of the second and third operand.

The comma operator has a sequence point after the first operand is evaluated so the following expressions are well defined.

```
f(), g()
i++, (i + 10)
(i = f()), (i > k)
```

Note that even though the four operators just discussed do contain sequence points, undefined behavior is still possible if any of their operands involves an operator which does not have a sequence point. For example:

```
(f() + g()) && (i++ > j)
(i + f()) || (i + i++)
(f() + g()) ? (a[i] = i++) : h()
(f() + g()), h()
```

The order of evaluation of the operands to the addition operator is undefined. All that the `&&` sequence point guarantees is that both functions will be called before the second operand is evaluated—assuming the first tests true. If either of the functions `f` or `g` modifies `i` or `j` and the other reads them, the outcome is undefined. Each of the other three cases also contain undefined behavior.

The only other operator having a sequence point is the function call. The order of evaluation of the function designator and its argument list, and the order of evaluation of arguments within the list, is unspecified, but there is a sequence point before the function is actually called. Therefore, the following expressions produce unspecified behavior:

```
f1(i, i++)
f2(i = 10, i)
f3(g(), h()) /* assuming g and h interact */
f4(g() + h()) /*      "           "           */
(*table[i])(i++)
```

In the last case, `i` is used in both the argument list and in the function designator. `table` is an array of function pointers but it is undefined which function is actually being called since `i` might or might not have been incremented before `table` was subscripted.

Let's revisit the expressions we saw earlier:

```
x[i] = y[i++]
a = i + ++i
a = b/(--b + c)
```

Based on what we have learned, each has unspecified behavior.

4.4 Conclusion

If you are relatively new to C and you come from a language that guarantees left-to-right evaluation you will likely be in for some unpleasant surprises unless you take particular care. Most of the side effect problems possible in C are avoided in other languages simply because those other languages don't have equivalents to ++ and --, and they don't permit embedded assignments. However, many languages do allow procedures that return values and calling one of these can generate a side effect.

Even if you somehow determine the order of evaluation of a particular “undefined” expression for a given compiler, you are strongly urged to not rely on this information since it might change in a future release. And since the order of evaluation is undefined it can even change for the *exact same* expression elsewhere in the program during the *same* compilation.

One of the most important skills we can learn with C is which properties are defined by the language, which are implementation-defined—and consequently are required to be documented by the implementation—and which are undefined or unspecified. Not only is this skill necessary if we are to write reliable code, it is absolutely mandatory if we plan to write portable code.

Chapter 5

Lvalues

Numerous compilation error messages involve the term ‘lvalue.’ Unfortunately, this term is not well understood and very often inadequately described—if described at all—in compiler manuals. In this chapter we will learn what an lvalue is and which operators produce one.

5.1 Introduction

Consider the following example:

```
/* 1*/ void f()
/* 2*/ {
/* 3*/     int i, j = 5, k;
/* 4*/     const char c = 'a';
/* 5*/     int g(void);
/* 6*/
/* 7*/     c = 'b';
/* 8*/
/* 9*/     for (i = 0; i < 5; ++i)
/*10*/         /* ... */;
/*11*/
/*12*/     g() = 6;
/*13*/
/*14*/     k = i+++++j;
/*15*/ }
```

```
Error line 7: l-value specifies const object
Error line 9: '++' needs l-value
Error line 12: '=' : left operand must be l-value
Error line 14: '++' needs l-value
```

According to the C standard, “An *lvalue* is an expression (with an object type or an incomplete type other than void) that designates an object. When an object is said to have a particular type, the type is specified by the lvalue used to designate the object.” The term ‘lvalue’ then, is an abbreviation for ‘lvalue expression’ and comes from the notion of the properties required of expressions involved in assignment. For example, in the expression `a = b` we are taking the value of `b` and storing it into the location designated by `a` whereas with `b = a` we are taking the value of `a` and storing it in the location designated by `b`. That is, depending on the context, we are interested in using the value of the expression or the address which it designates. The term ‘lvalue’ was used to refer to an expression that was permitted to be on the left-hand side of an assignment. Similarly, the term ‘rvalue’ was used to refer to an expression that was permitted to be on the right-hand side of an assignment.

Not only are expressions on the left and right side of assignment operators treated differently, ones permitted on the left must have a special property. Consider the following expressions:

```

b
10.345
10 + a * c
printf("Hi")

```

These expressions can be used on the right side of an assignment (that is, as rvalues) since each has a value. The only requirement of an rvalue is that it have a value. And the only type of expression in C that doesn't have a value is a `void` expression. As such, any non-`void` expression is an rvalue and, consequently, the term 'rvalue' has been dropped from the modern C vocabulary.

Except for `b`, these expressions do not designate a memory location. As such, they are not lvalues and cannot be used in any context where an lvalue is required.

5.2 Two Kinds of Lvalues

Clearly, the name of an array designates a location in memory. That is, the name of an array is an lvalue. However, the following is prohibited:

```

char c1[10], c2[10];

c1 = c2;          /* error */

```

Another example of an lvalue that cannot be used directly on the left-hand side of an assignment involves the `const` type qualifier¹. For example:

```

const char c[10];

c[2] = 'a';      /* error */

```

To handle these situations, we need two kinds of lvalues: modifiable and non-modifiable. Both designate memory locations but in the case of a non-modifiable lvalue, we cannot store into that location using that particular designator. Therefore, the expressions `c1` and `c[2]` above, are non-modifiable lvalues.

5.3 Operators That Generate Lvalues

Only a very few operators are capable of generating lvalues and only three always do so.

Operators that Produce Lvalues

<i>Operator</i>	<i>Example</i>	<i>Lvalue</i>	<i>Modifiable Lvalue</i>
[<code>a[i]</code>	yes	possibly
*	<code>*pc</code>	yes [†]	possibly
->	<code>pst->m</code>	yes	possibly
.	<code>st.m</code>	possibly	possibly
Others	—	never	never

[†] `*pc` designates an object provided `pc` is not a pointer to a function nor a pointer to an array of unknown size.

Since only objects can be contained in an array, `[]` designates one of those objects. Similarly, when a pointer to an object is dereferenced, we are referring to the underlying object. And, since all members of structures or unions must be objects, `->` always designates an object. The dot operator almost always generates an lvalue. The only case where it does not is in the case of `f().m`. This construct is permitted since a function can return a structure or union by value. However, since the function call operator `()` does not produce an lvalue, any subordinate member is also not an lvalue.

Any lvalue is also a modifiable lvalue provided it does not designate an array or a `const`-qualified object.

¹See §TYPEQUAL, "Type Qualifiers" for more information

5.4 Operators That Need Modifiable Lvalues

There are only three operators that always need modifiable lvalues: `++`, `--`, and the assignment operators (left-hand operand only).

Let's revisit the original example and look at each error. In line 7, `c` is an lvalue but not a modifiable lvalue; therefore it cannot be modified. In line 9, the `1` should have been `i`, non an uncommon mistake. Since the function call operator does not produce an lvalue, line 12 is rejected.

The problem in line 14 has to do with how tokens are recognized by the compiler. What the compiler is really seeing is not `i++ + ++j`, but rather, `i++++ + j`. The error arises since the operand to second `++` is not an lvalue.