CHAPTER 2: **TYPES, OPERATORS AND EXPRESSIONS**

Variables and constants are the basic data objects manipulated in a pro­
  
gram. Declarations list the variables to be used, and state what type they
  
have and perhaps what their initial values are. Operators specify what is to
  
be do'ne to them. Expressions combine variables and constants to produce
  
new values. These are the topics of this chapter.

**2.1 Variable Names**

Although we didn't come right out and say so, there are some restric­
  
tions on variable and symbolic constant names. Names are made up of
  
letters and digits; the first character must be a letter. The underscore "\_"
  
counts as a letter; it is useful for improving the readability of long variable
  
names. Upper and lower case are different; traditional C practice is to use
  
lower case for variable names, and all upper case for symbolic constants.

Only the first eight characters of an internal name are significant,
  
although more may be used. For external names such as function names
  
and external variables, the number may be less than eight, because external
  
names are used by various assemblers and loaders. Appendix A lists details.
  
Furthermore, keywords like if, else, int, float, etc., are *reserved:* you
  
can't use them as variable names. (They must be in lower case.)

Naturally it's wise to choose variable names that mean something, that
  
are related to the purpose of the variable, and that are unlikely to get mixed
  
up typographically.

**2.2 Data Types and Sizes**

There are only a few basic data types in C:

33

34 THE C PROGRAMMING LANGUAGE CHAPTER 2

char a single byte, capable of holding one character
  
in the local character set.

int an integer, typically reflecting the natural size
  
of integers on the host machine.

float single-precision floating point.
  
double double-precision floating point.

In addition, there are a number of qualifiers which can be applied to
  
int's: short, long, and unsigned. short and long refer to different
  
sizes of integers, unsigned numbers obey the laws of arithmetic modulo
  
2, where *n* is the number of bits in an int; unsigned numbers are
  
always positive. The declarations for the qualifiers look like

short int x;
  
long int y;

unsigned int z;

The word int can be omitted in such situations, and typically is.

The precision of these objects depends on the machine at hand; the

table below shows some representative values.

DEC PDP-11 Honeywell 6000 IBM 370 Interdata 8/32

ASCII ASCII EBCDIC ASCII

char 8 bits 9 bits 8 bits 8 bits

int 16 36 32 32

short 16 36 16 16

long 32 36 32 32

float 32 36 32 32

double 64 72 64 64

The intent is that short and long should provide different lengths of
  
integers where practical; int will normally reflect the most "natural" size
  
for a particular machine. As you can see, each compiler is free to interpret
  
short and long as appropriate for its own hardware. About all you should
  
count on is that short is no longer than long.

**2.3 Constants**

int and float constants have already been disposed of, except to note
  
that the usual

123.456e-7

or

0.12E3

"scientific" notation for float's is also legal. Every floating point constant

CHAPTER 2 TYPES, OPERATORS AND EXPRESSIONS 35

is taken to be double, so the "e" notation serves for both float and
  
double.

Long constants are written in the style 1 23L. An ordinary integer con­
  
stant that is too long to fit in an **int** is also taken to be a long.

There is a notation for octal and hexadecimal constants: a leading 0
  
(zero) on an **int** constant implies octal; a leading Ox or OX indicates hexa­
  
decimal. For example, decimal 31 can be written as 037 in octal and 0 x1 f
  
or 0 X1 **F** in hex. Hexadecimal and octal constants may also be followed by
  
L to make them long.

A *character constant* is a single character written within single quotes, as
  
in'x'. The value of a character constant is the numeric value of the char­
  
acter in the machine's character set. For example, in the ASCII character set
  
the character zero, or'O', is 48, and in EBCDIC 'O' is 240, both quite
  
different from the numeric value 0. Writing 0 instead of a numeric value
  
like 48 or 240 makes the program independent of the particular value.
  
Character constants participate in numeric operations just as any other
  
numbers, although they are most often used in comparisons with other char­
  
acters. A later section treats conversion rules.

Certain non-graphic characters can be represented in character constants
  
by escape sequences like \n (newline), \t (tab), \0 (null), \\ (backslash),
  
\' (single quote), etc., which look like two characters, but are actually only
  
one. In addition, an arbitrary byte-sized bit pattern can be generated by
  
writing

*'\ddd*

where *ddd* is one to three octal digits, as in

**#define FORMFEED '\014' /\* ASCII form feed \*/**

The character constant \ represents the character with value zero.

' \0' is often written instead of 0 to emphasize the character nature of
  
some expression.

A *constant expression* is an expression that involves only constants. Such
  
expressions are evaluated at compile time, rather than run time, and accord­
  
ingly may be used in any place that a constant may be, as in

**#define MAXLINE 1000
  
char line[MAXLINE+1];**

or

**seconds = 60 \* 60 \* hours;**

A *string constant* is a sequence of zero or more characters surrounded by
  
double quotes, as in

**"I am a string"**

36 THE C PROGRAMMING LANGUAGE CHAPTER 2

or

/\* a null string \*/

The quotes are not part of the string, but serve only to delimit it. The same
  
escape sequences used for character constants apply in strings; \" represents
  
the double quote character.

Technically, a string is an array whose elements are single characters.
  
The compiler automatically places the null character \O at the end of each
  
such string, so programs can conveniently find the end. This representation
  
means that there is no real limit to how long a string can be, but programs
  
have to scan one completely to determine its length. The physical storage
  
required is one more location than the number of characters written
  
between the quotes. The following function **strlen ( s )** returns the length
  
of a character string **s,** excluding the terminal \O.

strlen(s) /\* return length of s \*/
  
char s[];

int i;

i = 0;

while (s[i]

++i;

return(i);

}

Be careful to distinguish between a character constant and a string that
  
contains a single character: x' is not the same as "x". The former is a
  
single character, used to produce the numeric value of the letter x in the
  
machine's character set. The latter is a character string that contains one
  
character (the letter x) and a \0.

**2.4 Declarations**

All variables must be declared before use, although certain declarations
  
can be made implicitly by context. A declaration specifies a type, and is fol­
  
lowed by a list of one or more variables of that type, as in

int lower, upper, step;
  
char c, line[1000];

Variables can be distributed among declarations in any fashion; the lists
  
above could equally well be written as

CHAPTER 2 TYPES, OPERATORS AND EXPRESSIONS 37

**int lower;**

**int upper;**

**int step;**

**char c;**

**char line [1000] ;**

This latter form takes more room, but is convenient for adding a comment
  
to each declaration or for subsequent modifications.

Variables may also be initialized in their declaration, although there are
  
some restrictions. If the name is followed by an equals sign and a constant,
  
that serves as an initializer, as in

**char backslash =**

**int i = 0;**

**float eps = 1.0e-5;**

If the variable in question is external or static, the initialization is done
  
once only, conceptually before the program starts executing. Explicitly ini­
  
tialized automatic variables are initialized each time the function they are in
  
is called. Automatic variables for which there is no explicit initializer have
  
undefined (i.e., garbage) values. External and static variables are initialized
  
to zero by default, but it is good style to state the initialization anyway.

We will discuss initialization further as new data types are introduced.

**2.5 Arithmetic Operators**

The binary arithmetic operators are +, \*, /, and the modulus opera-

tor %. There is a unary —, but no unary +.

Integer division truncates any fractional part. The expression

**x % y**

produces the remainder when **x** is divided by y, and thus is zero when y
  
divides **x** exactly. For example, a year is a leap year if it is divisible by 4
  
but not by 100, except that years divisible by 400 *are* leap years. Therefore

**if (year % 4 == 0 && year % 100 != 0 I I year % 400 == 0)**

*it's a leap year*

**else**

*it's not*

The % operator cannot be applied to **float** or **double.**

The + and — operators have the same precedence, which is lower than
  
the (identical) precedence of \*, /, and %, which are in turn lower than
  
unary minus. Arithmetic operators group left to right. (A table at the end
  
of this chapter summarizes precedence and associativity for all operators.)
  
The order of evaluation is not specified for associative and commutative
  
operators like \* and +; the compiler may rearrange a parenthesized compu­
  
tation involving one of these. Thus **a+ (b+c)** can be evaluated as

38 THE C PROGRAMMING LANGUAGE CHAPTER 2

(a+b) +c. This rarely makes any difference, but if a particular order is
  
required, explicit temporary variables must be used.

The action taken on overflow or underflow depends on the machine at
  
hand.

**2.6 Relational and Logical Operators**

The relational operators are

> >= < <=

They all have the same precedence. Just below them in precedence are the
  
equality operators:

== !=

which have the same precedence. Relationals have lower precedence than
  
arithmetic operators, so expressions like ± < im-1 are taken as

**< ) ,** as would be expected.

More interesting are the logical connectives && and I I . Expressions
  
connected by && or I **I** are evaluated left to right, and evaluation stops as
  
soon as the truth or falsehood of the result is known. These properties are
  
critical to writing programs that work. For example, here is a loop from the
  
input function **getline** which we wrote in Chapter 1.

**for (1=0; i<lim-1 && (c=getchar()) != '\n' && c I= EOF; ++i)** 
  
**s[i] = c;**

Clearly, before reading a new character it is necessary to check that there is
  
room to store it in the array s, so the test i<lim-1 *must* be made first.
  
Not only that, but if this test fails, we must not go on and read another
  
character.

Similarly, it wOUld be unfortunate if **c** were tested against **EOF** before
  
**getchar** was called: the call must occur before the character in **c** is tested.

The precedence of fs,,s, is greater than that of **I** I, and both are lower
  
than relational and equality operators, so expressions like

**&& (c = getchar()) != '\n' && c != EOF**

need no extra parentheses. But since the precedence of ! = is higher than
  
assignment, parentheses are needed in

(c **= getchar())**

to achieve the desired result.

The unary negation operator ! converts a non-zero or true operand into
  
0, and a zero or false operand into 1. A common use of ! is in construc­
  
tions like

CHAPTER 2 TYPES, OPERATORS AND EXPRESSIONS 39

if (!inword)

rather than

if (inword == 0)

It's hard to generalize about which form is better. Constructions like
  
! inword read quite nicely ("if not in word"), but more complicated ones
  
can be hard to understand.

**Exercise 2-1.** Write a loop equivalent to the for loop above without using
  
&&. LI

**2.7 Type Conversions**

When operands of different types appear in expressions, they are con­
  
verted to a common type according to a small number of rules. In general,
  
the only conversions that happen automatically are those that make sense,
  
such as converting an integer to floating point in an expression like f + i.
  
Expressions that don't make sense, like using a float as a subscript, are
  
disallowed.

First, char's and it's may be freely intermixed in arithmetic expres­
  
sions: every char in an expression is automatically converted to an int.
  
This permits considerable flexibility in certain kinds of character transforma­
  
tions. One is exemplified by the function atoi, which converts a string of
  
digits into its numeric equivalent.

atoi(s) /\* convert s to integer \*/
  
char s[];

int i, n;

n = 0;

for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)

n = 10 \* n + s[i] - '0';

return (n);

As we discussed in Chapter 1, the expression
  
s[i] - '0'

gives the numeric value of the character stored in s [ii because the values
  
of'0', 1 , etc., form a contiguous increasing positive sequence.

Another example of char to int conversion is the function lower
  
which maps a single character to lower case *for the ASCII character set only.* 
  
If the character is not an upper case letter, lower returns it unchanged.

40 THE C PROGRAMMING LANGUAGE CHAPTER 2

**lower(c) /\* convert c to lower case; ASCII only \*/** 
  
**int c;**

**if (c >= 'A' && c <= 'Z')
  
return(c + 'a' - 'A');
  
else**

**return(c);**

This works for ASCII because corresponding upper case and lower case
  
letters are a fixed distance apart as numeric values and each alphabet is con­
  
tiguous — there is nothing but letters between *A* and *Z.* This latter obser­
  
vation is *not* true of the EBCDIC character set (IBM 360/370), so this code
  
fails on such systems — it converts more than letters.

There is one subtle point about the conversion of characters to integers.
  
The language does not specify whether variables of type **char** are signed or
  
unsigned quantities. When a **char** is converted to an **int,** can it ever pro­
  
duce a *negative* integer? Unfortunately, this varies from machine to
  
machine, reflecting differences in architecture. On some machines (PDP-11,
  
for instance), a **char** whose leftmost bit is 1 will be converted to a negative
  
integer ("sign extension"). On others, a **char** is promoted to an int by
  
adding zeros at the left end, and thus is always positive.

The definition of C guarantees that any character in the machine's stan­
  
dard character set will never be negative, so these characters may be used
  
freely in expressions as positive quantities. But arbitrary bit patterns stored
  
in character variables may appear to be negative on some machines, yet
  
positive on others.

The most common occurrence of this situation is when the value —1 is
  
used for **EOF.** Consider the code

**char c;**

**c = getchar();
  
if (c == EOF)**

On a machine which does not do sign extension, c is always positive
  
because it is a **char,** yet **EOF** is negative. As a result, the test always fails.
  
To avoid this, we have been careful to use **int** instead of **char** for any
  
variable which holds a value returned by **getchar.**

The real reason for using int instead of **char** is not related to any
  
questions of possible sign extension. It is simply that **getchar** must return
  
all possible characters (so that it can be used to read arbitrary input) and, in
  
addition, a distinct **EOF** value. Thus its value *cannot* be represented as a
  
**char,** but must instead be stored as an int.

CHAPTER 2 TYPES, OPERATORS AND EXPRESSIONS 41

Another useful form of automatic type conversion is that relational
  
expressions like i > j and logical expressions connected by && and I **I** are
  
defined to have value 1 if true, and 0 if false. Thus the assignment

**isdigit = c >= ' 0 ' && c <= '9';**

sets **isdigit** to 1 if **c** is a digit, and to 0 if not. (In the test part of **if,** 
  
**while, for,** etc., "true" just means "non-zero.")

Implicit arithmetic conversions work much as expected. In general, if
  
an operator like + or \* which takes two operands (a "binary operator") has
  
operands of different types, the "lower" type is *promoted* to the "higher"
  
type before the operation proceeds. The result is of the higher type. More
  
precisely, for each arithmetic operator, the following sequence of conversion
  
rules is applied.

**char** and **short** are converted to **int,** and **float** is converted to
  
**double.**

Then if either operand is **double,** the other is converted to
  
**double,** and the result is **double.**

Otherwise if either operand is long, the other is converted to
  
long, and the result is long.

Otherwise if either operand is unsigned, the other is converted to
  
unsigned, and the result is unsigned.

Otherwise the operands must be int, and the result is **int.**

Notice that all **float's in** an expression are converted to **double;** all float­
  
ing point arithmetic in C is done in double precision.

Conversions take place across assignments; the value of the right side is
  
converted to the type of the left, which is the type of the result. A charac­
  
ter is converted to an integer, either by sign extension or not, as described
  
above. The reverse operation, int to **char, is well-behaved —** excess
  
high-order bits are simply discarded. Thus in

**int i;
  
char c;**

I = **c;
  
c = i;**

the value of **c** is unchanged. This is true whether or not sign extension is

involved.

If **x is float and i is int, then**

x= i

**and**

42 THE C PROGRAMMING LANGUAGE CHAPTER 2

i =**x**

both cause conversions; float to int causes truncation of any fractional
  
part. double is converted to float by rounding. Longer it's are con­
  
verted to shorter ones or to **char's** by dropping the excess high-order bits.

Since a function argument is an expression, type conversions also take
  
place when arguments are passed to functions: in particular, **char** and
  
**short** become **int,** and **float** becomes **double.** This is why we have
  
declared function arguments to be **int** and double even when the func­
  
tion is called with **char** and **float.**

Finally, explicit type conversions can be forced ("coerced") in any
  
expression with a construct called a *cast.* In the construction

*(type-name) expression*

the *expression* is converted to the named type by the conversion rules above.
  
The precise meaning of a cast is in fact as if *expression* were assigned to a
  
variable of the specified type, which is then used in place of the whole con­
  
struction. For example, the library routine **sqrt** expects a double argu­
  
ment, and will produce nonsense if inadvertently handed something else.
  
So if n is an integer,

**sqrt ( (double) n)**

converts n to double before passing it to **sqrt.** (Note that the cast pro­
  
duces the *value* of n in the proper type; the actual content of n is not
  
altered.) The cast operator has the same precedence as other unary opera­
  
tors, as summarized in the table at the end of this chapter.

**Exercise 2-2.** Write the function **htoi (s ) ,** which converts a string of
  
hexadecimal digits into its equivalent integer value. The allowable digits are
  
0 through **9, a** through **f, and A** through **F. 1=1**

**2.8 Increment and Decrement Operators**

C provides two unusual operators for incrementing and decrementing
  
variables. The increment operator ++ adds 1 to its operand; the decrement
  
operator -- subtracts 1. We have frequently used ++ to increment vari­
  
ables, as in

**if (c == \n')
  
++nl;**

The unusual aspect is that ++ and -- may be used either as prefix
  
operators (before the variable, as in **++n),** or postfix (after the variable:
  
n++). In both cases, the effect is to increment n. But the expression **++n** 
  
increments **n** *before* using its value, while **n++** increments **n** *after* its value
  
has been used. This means that in a context where the value is being used,

CHAPTER 2 TYPES, OPERATORS AND EXPRESSIONS **43**

not just the effect, **++n** and n++ are different. If n is 5, then

x **= n++;**

sets x to 5, but

x **= ++n;**

sets x to 6. In both cases, n becomes 6. The increment and decrement
  
operators can only be applied to variables; an expression like **x=(i+j)++** is
  
illegal.

In a context where no value is wanted, just the incrementing effect, as
  
in

**if (c ==
  
nl++;**

choose prefix or postfix according to taste. But there are situations where
  
one or the other is specifically called for. For instance, consider the func­
  
tion **squeeze (s, c)** which removes all occurrences of the character **c** 
  
from the string **s.**

**squeeze(s, c) /\* delete all c from s \*/**

**char s[];**

**int c;**

**{**

**int i, j;**

**for (i = j = 0; s[i] != '\0'; i++)**

**if (s[i] != c)**

s [j++] = s [i] ;

s[j] = 'VP;

)

Each time a non-c occurs, it is copied into the current j position, and only
  
then is j incremented to be ready for the next character. This is exactly
  
equivalent to

**if (s[i] != c) (
  
s[j] = s[i];
  
j++;**

}

Another example of a similar construction comes from the **getline** 
  
function which we wrote in Chapter I, where we can replace

**if (c ==
  
s[i] = c;**

)

by the more compact

44 THE C PROGRAMMING LANGUAGE CHAPTER 2

if (c ==
  
s[i++] = c;

As a third example, the function strcat(s) t) concatenates the
  
string t to the end of the string s. strcat assumes that there is enough
  
space in s to hold the combination.

strcat(s, t) /\* concatenate t to end of s \*/

char s[], t[]; /\* s must be big enough \*/

{

int i, j;

i = j = 0;

while (s[i] != '\0') /\* find end of s \*/

i++;

while ((s[i++] = t[j++]) != '\0') /\* copy t \*/

;

)

As each character is copied from t to s, the postfix ++ is applied to both i
  
and j to make sure that they are in position for the next pass through the
  
loop.

**Exercise 2-3.** Write an alternate version of squeeze (s1, s2) which
  
deletes each character in s1 which matches any character in the *string* s2.
  
ID

**Exercise 2-4.** Write the function any (s1, s2) which returns the first
  
location in the string s1 where any character from the string s2 occurs, or
  
—1 if s1 contains no characters from s2. El

**2.9 Bitwise Logical Operators**

C provides a number of operators for bit manipulation; these may not
  
be applied to float or double.

& bitwise AND

I bitwise inclusive OR

A bitwise exclusive OR

« left shift

» right shift

one's complement (unary)

The bitwise AND operator & is often used to mask off some set of bits; for
  
example,

c = n & 0177;

sets to zero all but the low-order 7 bits of n. The bitwise OR operator I is
  
used to turn bits on:

CHAPTER 2 TYPES, OPERATORS AND EXPRESSIONS **45**

**x = x I MASK;**

sets to one in **x** the bits that are set to one in **MASK.**

You should carefully distinguish the bitwise operators & and I from the
  
logical connectives && and I I, which imply left-to-right evaluation of a
  
truth value. For example, if **x** is 1 and **y** is 2, then **x & y** is zero while
  
x && y is one. (Why?)

The shift operators « and » perform left and right shifts of their left
  
operand by the number of bit positions given by the right operand. Thus
  
x « 2 shifts **x** left by two positions, filling vacated bits with 0; this is
  
equivalent to multiplication by 4. Right shifting an unsigned quantity fills
  
vacated bits with 0. Right shifting a signed quantity will fill with sign bits
  
("arithmetic shift") on some machines such as the **PDP-11,** and with 0-bits
  
("logical shift") on others.

The unary operator - yields the one's complement of an integer; that is,
  
it converts each 1-bit into a 0-bit and vice versa. This operator typically
  
finds use in expressions like

**x & -077**

which masks the last six bits of **x** to zero. Note that x & -077 is indepen­
  
dent of word length, and is thus preferable to, for example, **x & 0177700,** 
  
which assumes that **x** is a 16-bit quantity. The portable form involves no
  
extra cost, since -077 is a constant expression and thus evaluated at com­
  
pile time.

To illustrate the use of some of the bit operators, consider the function

**getbits p,** n) which returns (right adjusted) the n-bit field of **x**that begins at position p. We assume that bit position 0 is at the right end
  
and that **n** and p are sensible positive values. For example,
  
**getbits (x, 4) 3)** returns the three bits in bit positions 4, 3 and 2, right
  
adjusted.

**getbits(x, p, n) /\* get n bits from position p \*/** 
  
**unsigned x, p, n;**

**return((x >> (p+1-n)) & -(-0 << n));**

**x >> (p+1 -n)** moves the desired field to the right end of the word.
  
Declaring the argument **x** to be **unsigned** ensures that when it is right-
  
shifted, vacated bits will be filled with zeros, not sign bits, regardless of the
  
machine the program is run on. -0 is all 1-bits; shifting it left **n** bit posi­
  
tions with -0 « **n** creates a mask with zeros in the rightmost n bits and
  
ones everywhere else; complementing that with - makes a mask with ones
  
in the rightmost **n** bits.

46 THE C PROGRAMMING LANGUAGE CHAPTER 2

**Exercise 2-5.** Modify getbits to number bits from left to right. LI

**Exercise 2-6.** Write a function word length ( ) which computes the word
  
length of the host machine, that is, the number of bits in an **int.** The
  
function should be portable, in the sense that the same source code works
  
on all machines.

**Exercise 2-7.** Write the function **rightrot (n) b)** which rotates the
  
integer n to the right by b bit positions.

**Exercise 2-8.** Write the function **invert (x) p,** n) which inverts (i.e.,
  
changes 1 into 0 and vice versa) the **n** bits of **x** that begin at position p,
  
leaving the others unchanged.

**2.10 Assignment Operators and Expressions**

Expressions such as

= + **2**

in which the left hand side is repeated on the right can be written in the
  
compressed form

**+= 2**

using an *assignment operator* like +=.

Most binary operators (operators like + which have a left and right

operand) have a corresponding assignment operator *op=,* where *op* is one of

+ - \* / % « » &

If *el* and *e2* are expressions, then

*el op= e2*

is equivalent to

*el = (el) op (e2)*

except that *el* is computed only once. Notice the parentheses around *e2:*

\*= y + 1

is actually

x = x \* *(y + 1)*

rather than

**x=x\*y+**

As an example, the function bitcount counts the number of 1-bits in
  
its integer argument.

CHAPTER 2 TYPES, OPERATORS AND EXPRESSIONS 47

bitcount(n) /\* count 1 bits in n \*/
  
unsigned n;

int b;

for (b = 0; n I= 0; n >>= 1)

if (n & 01)

b++;

return(b);

}

Quite apart from conciseness, assignment operators have the advantage
  
that they correspond better to the way people think. We say "add 2 to i"
  
or "increment i by 2," not "take 1, add 2, then put the result back in
  
Thus ± += 2. In addition, for a complicated expression like

yyval[yypv[p3+p4] + yypv[p1+p2]] += 2

the assignment operator makes the code easier to understand, since the
  
reader doesn't have to check painstakingly that two long expressions are
  
indeed the same, or to wonder why they're not. And an assignment opera­
  
tor may even help the compiler to produce more efficient code.

We have already used the fact that the assignment statement has a value
  
and can occur in expressions; the most common example is

while ((c = getchar()) != EOF)

* • •

Assignments using the other assignment operators (+=, —=, etc.) can also

occur in expressions, although it is a less frequent occurrence.

The type of an assignment expression is the type of its left operand.

**Exercise 2-9.** In a 2's complement number system, x & ( x-1 ) deletes the
  
rightmost 1-bit in **x.** (Why?) Use this observation to write a faster version
  
of bitcount.

**2.11 Conditional Expressions**The statements

if (a > b)

z = a;

else

z = b;

of course compute in z the maximum of a and b. The *conditional expres­*
  
*sion,* written with the ternary operator "? : ", provides an alternate way to
  
write this and similar constructions. In the expression

48 THE C PROGRAMMING LANGUAGE CHAPTER 2

*el ? e2 : e3*

the expression *el* is evaluated first. If it is non-zero (true), then the expres­
  
sion *e2* is evaluated, and that is the value of the conditional expression.
  
Otherwise *e3* is evaluated, and that is the value. Only one of *e2* and *e3* is
  
evaluated. Thus to set z to the maximum of **a** and b,

**z = (a > b) ? a : b; /\* z = max(a, b) \*/**

It should be noted that the conditional expression is indeed an expres­
  
sion, and it can be used just as any other expression. If *e2* and *e3* are of
  
different types, the type of the result is determined by the conversion rules
  
discussed earlier in this chapter. For example, if **f** is a **float,** and n is an
  
int, then the expression

**(n > 0) ? f : n**

is of type **double** regardless of whether n is positive or not.

Parentheses are not necessary around the first expression of a condi­
  
tional expression, since the precedence of ? : is very low, just above assign­
  
ment. They are advisable anyway, however, since they make the condition
  
part of the expression easier to see.

The conditional expression often leads to succinct code. For example,
  
this loop prints **N** elements of an array, 10 per line, with each column
  
separated by one blank, and with each line (including the last) terminated by
  
exactly one newline.

**for (i = 0; i < N; i++)**

**printf("%6d%c", a[i], (i%10==9 II i==N-1) ? '\n' :** " ) ;

A newline is printed after every tenth element, and after the N-th. All other
  
elements are followed by one blank. Although this might look tricky, it's
  
instructive to try to write it without the conditional expression.

**Exercise 2-10.** Rewrite the function lower, which converts upper case
  
letters to lower case, with a conditional expression instead of **if—else.** El

**2.12 Precedence and Order of Evaluation**

The table below summarizes the rules for precedence and associativity
  
of all operators, including those which we have not yet discussed. Operators
  
on the same line have the same precedence; rows are in order of decreasing
  
precedence, so, for example, \*, /, and % all have the same precedence,
  
which is higher than that of + and —.

CHAPTER 2 TYPES, OPERATORS AND EXPRESSIONS **49**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Operator | |  |  |  | Associativity |
| ( ) | [ ] -> . |  |  |  | left to right |
| ! | - ++ -- - *( type )* | *\** | *&* | **sizeof** | right to left |
| \* | / % |  |  |  | left to right |
| + | - |  |  |  | left to right |
| « | >> |  |  |  | left to right |
| < | <= > >= |  |  |  | left to right |
| == | ! = |  |  |  | left to right |
| & |  |  |  |  | left to right |
|  |  |  |  |  | left to right |
| I |  |  |  |  | left to right |
| && |  |  |  |  | left to right |
| I **I** |  |  |  |  | left to right |
| ? : |  |  |  |  | right to left |
| = | += -= etc. |  |  |  | right to left |
| , | (Chapter 3) |  |  |  | left to right |

The operators -> and . are used to access members of structures; they will
  
be covered in Chapter 6, along with **sizeof** (size of an object). Chapter 5
  
discusses \* (indirection) and & (address of).

Note that the precedence of the bitwise logical operators &, A and **I** falls
  
below == and ! =. This implies that bit-testing expressions like

**if ( (x & MASK) == 0) ...**

must be fully parenthesized to give proper results.

As mentioned before, expressions involving one of the associative and
  
commutative operators (\*, +, &, ^, I ) can be rearranged even when
  
parenthesized. In most cases this makes no difference whatsoever; in situa­
  
tions where it might, explicit temporary variables can be used to force a par­
  
ticular order of evaluation.

C, like most languages, does not specify in what order the operands of
  
an operator are evaluated. For example, in a statement like

x = **f + g();**

**f** may be evaluated before g or vice versa; thus if either **f or** g alters an
  
external variable that the other depends on, x can depend on the order of
  
evaluation. Again, intermediate results can be stored in temporary variables
  
to ensure a particular sequence.

50 THE C PROGRAMMING LANGUAGE CHAPTER 2

Similarly, the order in which function arguments are evaluated is not
  
specified, so the statement

**printf("%d %d\n", ++n, power(2, n)); /\* WRONG \*/**

can (and does) produce different results on different machines, depending
  
on whether or not n is incremented before power is called. The solution,
  
of course, is to write

**++n;**

**printf("%d %d\n", n, power(2, n));**

Function calls, nested assignment statements, and increment and decre­
  
ment operators cause "side effects" — some variable is changed as a by­
  
product of the evaluation of an expression. In any expression involving side
  
effects, there can be subtle dependencies on the order in which variables
  
taking part in the expression are stored. One unhappy situation is typified
  
by the statement

**a[i] = i++;**

The question is whether the subscript is the old value of i or the new. The
  
compiler can do this in different ways, and generate different answers
  
depending on its interpretation. When side effects (assignment to actual
  
variables) takes place is left to the discretion of the compiler, since the best
  
order strongly depends on machine architecture.

The moral of this discussion is that writing code which depends on order
  
of evaluation is a bad programming practice in any language. Naturally, it is
  
necessary to know what things to avoid, but if you don't know *how* they are
  
done on various machines, that innocence may help to protect you. (The C
  
verifier *lint* will detect most dependencies on order of evaluation.)