Chapter 2: Types, Operators, and Expressions

Page 35: Deep Sentence: The type of an object determines the set of values it can have and what operations can be performed on it.

This is a fairly formal, mathematical definition of what a type is, but it is traditional (and meaningful). There are several implications to remember:

1. The “set of values” is finite. C’s int type cannot represent all of the integers; its float type cannot represent all floating-point numbers.

2. When you’re using an object (that is, a variable) of some type, you may have to remember what values it can take on and what operations you can perform on it. For example, there are several operators which play with the binary (bit-level) representation of integers, but these operators are not meaningful for and may not be applied to floating-point operands.

3. When declaring a new variable and picking a type for it, you have to keep in mind the values and operations you’ll be needing.

In other words, picking a type for a variable is not some abstract academic exercise; it’s closely connected to the way(s) you’ll be using that variable.

You don’t need to worry about the list of “small changes and additions” made by the ANSI standard, unless you started learning C long ago or have a keen interest in its history. We’ll be using these new features indiscriminately, usually without comment.

Section 2.1: Variable Names

Page 35: Deep sentence: Don’t begin variable names with underscore, however, since library routines often use such names.

If you happen to pick a name which “collides” with (is the same as) a name already chosen by a library routine, either your code or the library routine (or both) won’t work. Naming issues become very significant in large projects, and problems can be avoided by setting guidelines for who may use which names. One of these guidelines is simply that user code should not use names beginning with an underscore, because these names are (for the most part) “reserved to the implementation” (that is, reserved for use by the compiler and the standard library).
Note that case is significant; assuming that case is ignored (as it is with some other programming languages and operating systems) can lead to real frustration.

The convention that all-upper-case names are used for symbolic constants (i.e. as created with the #define directive, which we learned about in section 1.4) is arbitrary, but useful. Like the various conventions for code layout (page 10), this convention is a good one to accept (i.e. not get too creative about), until you have some very good reason for altering it.

Page 35-6: Deep sentence: Keywords like if, else, int, float, etc., are reserved; you can’t use them as variable names.

You can find the complete list of keywords in appendix A2.4 on page 192.

Section 2.2: Data Types and Sizes

Page 36: If you can look at this list of “a few basic types in C” and say to yourself, “Oh, how simple, there are only a few types, I won’t have to worry much about choosing among them,” you’ll have an easy time with declarations. (Some masochists wish that the type system were more complicated so that you could specify more things about each variable, but those of us who would rather not have to specify these extra things each time are glad that we don’t have to.)

Note that the basic types are defined as having at least a certain size. There is no specification that a short int will be exactly 16 bits, or that a long int will be exactly 32 bits. Some programmers become obsessed with knowing exactly what sizes things will be in various situations, and write programs which depend on things having certain sizes. Exact sizes are occasionally important, but most of the time we can sidestep size issues and let the compiler do most of the worrying.

Most of the simple variables in most programs are of types int, long int, or double. Typically, we’ll use int and double for most purposes, and long int any time we need to hold values greater than 32,767. We’ll rarely use individual variables of type char; although we’ll use plenty of arrays of char. Types short int and float are important primarily when efficiency (speed or memory usage) is a concern, and for us it usually won’t be.

Note that even when we’re manipulating individual characters, we’ll usually use an int variable, for the reason discussed in section 1.5.1 on page 16.

Section 2.3: Constants

Page 37: We write constants in decimal, octal, or hexadecimal for our convenience, not the compiler’s. The compiler doesn’t care; it always converts everything into binary internally, anyway. (There is, however, no good way to specify constants in source code in binary.)

Pages 37-38: Read the descriptions of character and string constants carefully; most C programs work with these data types a lot, and their proper use must be kept in mind. Note particularly these facts:

1. The character constant ‘x’ is quite different from the string constant "x".
2. The value of a character is simply “the numeric value of the character in the machine’s character set.”

3. Strings are terminated by the null character, \0. (This applies to both string constants and to all other strings we’ll build and manipulate.) This means that the size of a string (the number of char’s worth of memory it occupies) is always one more than its length (i.e. as reported by strlen) appears to be.

As we saw in Section 1.6 on page 23, it’s possible to switch rather freely between thinking of a character as a character and thinking of it as its value. For example, the character ‘0’ (that is, the character that can print on your screen and looks like the number zero) has in the ASCII character set the internal value 48. Another way of saying this is to notice that the following expressions are all true:

\`
\'0' == 48  
\'0' == '\060' 
\'0' == '\x30'
\`

We’ll have a bit more to say about characters and their small integer representations in section 2.7.

Note also that the string "48" consists of the three characters ‘4’, ‘8’, and ‘\0’. Also in section 2.7 we’ll meet the atoi function which computes a numeric value from a string of digits like this.

Page 39: We won’t be using enumerations, so you don’t have to worry too much about the description of enumeration constants.

Section 2.4: Declarations

Page 40: You may wonder why variables must be declared before use. There are two reasons:

1. It makes things somewhat easier on the compiler; it knows right away what kind of storage to allocate and what code to emit to store and manipulate each variable; it doesn’t have to try to intuit the programmer’s intentions.

2. It forces a bit of useful discipline on the programmer: you cannot introduce variables willy-nilly; you must think about them enough to pick appropriate types for them. (The compiler’s error messages to you, telling you that you apparently forgot to declare a variable, are as often helpful as they are a nuisance: they’re helpful when they tell you that you misspelled a variable, or forgot to think about exactly how you were going to use it.)

Although there are a few places where “certain declarations can be made implicitly by context”, making use of these removes the advantages of reason 2 above, so I recommend always declaring everything explicitly.

Most of the time, I recommend writing one declaration per line (as in the “latter form” on page 40). For the most part, the compiler doesn’t care what order declarations are in. You can order the declarations alphabetically, or in the order that they’re used, or to put related declarations next to each other. Collecting all variables of the same type together on one line essentially orders declarations by type, which isn’t a very useful order (it’s only slightly more useful than random order).
If you’d rather not remember the rules for default initialization (namely that “external or static variables are initialized to zero by default” and “automatic variables for which there is no initializer have... garbage values”), you can get in the habit of initializing everything. It never hurts to explicitly initialize something when it would have been implicitly initialized anyway, but forgetting to initialize something that needs it can be the source of frustrating bugs.

Don’t worry about the distinction between “external or static variables”; we haven’t seen it yet.

One mild surprise is that const variables are not “constant expressions” as defined on page 38. You can’t say something like

```c
const int maxline = 1000;
char line[maxline+1]; /* WRONG */
```

Section 2.5: Arithmetic Operators

Page 41: Keep in the back of your mind somewhere the fact that the behavior of the / and % operators is not precisely defined for negative operands. This means that -7 / 4 might be -1 or -2, and -7 % 4 might be -3 or +1. The difference won’t matter for the simple programs we’ll be writing at first, but eventually you’ll get bit by it if you don’t remember it.

An additional arithmetic operation you might be wondering about is exponentiation. Some languages have an exponentiation operator (typically ^ or **), but C doesn’t.

The term “precedence” refers to how “tightly” operators bind to their operands (that is, to the things they operate on). In mathematics, multiplication has higher precedence than addition, so 1 + 2 * 3 is 7, not 9. In other words, 1 + 2 * 3 is equivalent to 1 + (2 * 3). C is the same way.

The term “associativity” refers to the grouping when two or more operators of the same precedence participate next to each other in an expression. When an operator (like subtraction) associates “left to right,” it means that 1 - 2 - 3 is equivalent to (1 - 2) - 3 and gives -4, not +2.

By the way, the word “arithmetic” as used in the title of this section is an adjective, not a noun, and it’s pronounced differently than the noun: the accent is on the third syllable.

Section 2.6: Relational and Logical Operators

If it isn’t obvious, >= is greater-than-or-equal-to, <= is less-than-or-equal-to, == is equal-to, and != is not-equal-to. We use >=, <=, and != because the symbols >=, <=, and != are not common on computer keyboards, and we use == because equality testing and assignment are two completely different operations, but = is already taken for assignment. (Obviously, typing = when you mean == is a very easy mistake to make, so watch for it. Some compilers will warn you when you use one but seem to want the other.)

The fact that evaluation of the logical operators && and || “stops as soon as the truth or falsehood of the result is known” refers to the fact that
“false” AND anything is false

or, in C,

\(0 \&\& \text{anything} = 0\)

while, on the other hand,

“true” OR anything is true

or, in C,

\(1 \|\| \text{anything} = 1\)

Looking at these another way, if you want to do something if thing1 is true and thing2 is true, and
you’ve just noticed that thing1 is false, you don’t even need to check thing2. Similarly, if you’re
supposed to do something if thing3 is true or thing4 is true, and you notice that thing3 is true, you can
go ahead and do whatever it is you’re supposed to do without checking thing4.

C works the same way, and if it’s not true that “most C programs rely on these properties,” it’s
certainly true that many do.

For another example of the usefulness of this “short-circuiting” behavior, suppose we’re taking the
average of n numbers. If n is zero, that is, if we don’t have any numbers to take the average of, we don’t
want to divide by zero. Code like

\[
\text{if}(n \neq 0 \&\& \text{sum} / n > 1)
\]

is common: it tests whether n is nonzero and the average is greater than 1, but it does not have to
worry about dividing by zero. (If, on the other hand, the compiler always evaluated both sides of the &&
before checking to see whether they were both true, the code above could divide by zero.)

\textbf{Page 42:} Note the extra parentheses in

\[
(c = \text{getchar()}) \neq ‘\n’
\]

Since this is a common idiom, you’ll need to remember the parentheses. What would

\[
c = \text{getchar() \neq ‘\n’}
\]

do?

C’s treatment of Boolean values (that is, those where we only care whether they’re true or false) is
straightforward. We’ll have more to say about it later, but for now, note that a value of zero is “false,”
and any nonzero value is “true.” You might also note that there is no necessary connection between
statements like if() which expect a true/false value and operators like >= and && which generate
true/false values. You can use operators like >= and && in any expression, and you can use any
expression in an if() statement.
The authors make a good point about style: if valid is conceptually a Boolean variable (that is, it’s an integer, but we only care about whether it’s zero or nonzero, in other words, “false” or “true”), then

```c
if(valid)
```

is a perfectly reasonable and readable condition. However, when values are not conceptually Boolean, I encourage you to make explicit comparisons against 0. For example, we could have expressed our average-taking code as

```c
if(n == 0 && sum / n > 1)
```

but I think it’s clearer to be explicit and say

```c
if(n != 0 && sum / n > 1)
```

(However, many C programmers feel that expressions like

```c
if(n && sum / n > 1)
```

are “more concise,” so you will see them all the time and you should be able to read them.)

### Section 2.7: Type Conversions

The conversion rules described here and on page 44 are straightforward, but they’re quite important, so you’ll need to learn them well. Usually, conversions happen automatically and when you want them to, but not always, so it’s important to keep the rules in mind. (Recall the discussion of 5/9 on page 12.)

Page 43: Deep sentence: A char is just a small integer, so chars may be freely used in arithmetic expressions.

Whether you treat a “small integer” as a character or an integer is pretty much up to you. As we saw earlier, in the ASCII character set, the character ‘0’ has the value 48. Therefore, saying:

```c
int i = '0';
```

is the same as saying

```c
int i = 48;
```

If you print i out as a character, using

```c
putchar(i);
```

or

```c
printf("%c", i);
```

(the %c format prints characters; see page 13), you’ll see the character ‘0’. If you print it out as a number:
printf("%d", i);

you’ll see the value 48.

Most of the time, you’ll use whatever notation matches what you’re trying to do. If you want the character ‘0’, you’ll use ‘0’. If you want the value 48 (as the number of months in four years, or something), you’ll use 48. If you want to print characters, you’ll use putchar or printf %c, and if you want to print integers, you’ll use printf %d. Occasionally, you’ll cross over between thinking of characters as characters and as values, such as in the character-counting program in section 1.6 on page 22, or in the atoi function we’ll look at next. (You should never have to know that ‘0’ has the value 48, and you should never have to write code which depends on it.)

Page 43: To illustrate the “schizophrenic” nature of characters (are they characters, or are they small integer values?), it’s useful to look at an implementation of the standard library function atoi. (If you’re getting overwhelmed, though, you may skip this example for now, and come back to it later.) The atoi routine converts a string like “123” into an integer having the corresponding value.

As you study the atoi code at the top of page 43, figure out why it does not seem to explicitly check for the terminating ‘\0’ character.

The expression

s[i] – ‘0’

is an example of the “crossing over” between thinking about a character and its value. Since the value of the character ‘0’ is not zero (and, similarly, the other numeric characters don’t have their “obvious” values, either), we have to do a little conversion to get the value 0 from the character ‘0’, the value 1 from the character ‘1’, etc. Since the character set values for the digit characters ‘0’ to ‘9’ are contiguous (48-57, if you must know), the conversion involves simply subtracting an offset, and the offset (if you think about it) is simply the value of the character ‘0’. We could write:

s[i] – 48

if we really wanted to, but that would require knowing what the value actually is. We shouldn’t have to know (and it might be different in some other character set), so we can let the compiler do the dirty work by using ‘0’ as the offset (since subtracting ‘0’ is, by definition, the same as subtracting the value of the character ‘0’).

The functions from <ctype.h> are being introduced here without a lot of fanfare. Here is the main loop of the atoi routine, rewritten to use isdigit:

for (i = 0; isdigit(s[i]); ++i)
    n = 10 * n + (s[i] – ‘0’);

Don’t worry too much about the discussion of signed vs. unsigned characters for now. (Don’t forget about it completely, though; eventually, you’ll find yourself working with a program where the issue is significant.) For now, just remember:
1. Use int as the type of any variable which receives the return value from getchar, as discussed in section 1.5.1 on page 16.

2. If you’re ever dealing with arbitrary “bytes” of binary data, you’ll usually want to use unsigned char.

Page 44: As we saw in section 2.6 on page 44, relational and logical operators always “return” 1 for “true” and 0 for “false.” However, when C wants to know whether something is true or false, it just looks at whether it’s nonzero or zero, so any nonzero value is considered “true.” Finally, some functions which return true/false values (the text mentions isdigit) may return “true” values of other than 1.

You don’t have to worry about these distinctions too much, and you also don’t have to worry about the fragment

\[ d = c >= '0' \&\& c <= '9' \]

as long as you write conditionals in a sensible way. If you wanted to see whether two variables a and b were equal, you’d never write

\[ \text{if}((a == b) == 1) \]

(although it would work: the == operator “returns” 1 if they’re equal). Similarly, you don’t want to write

\[ \text{if}(\text{isdigit}(c) == 1) \]

because it’s equally silly-looking, and in this case it might not work. Just write things like

\[ \text{if}(a == b) \]

and

\[ \text{if}(\text{isdigit}(c)) \]

and you’ll steer clear of most problems. (Make sure, though, that you never try something like if('0' <= c <= '9'), since this wouldn’t do at all what it looks like it’s supposed to.)

The set of implicit conversions on page 44, though informally stated, is exactly the set to remember for now. They’re easy to remember if you notice that, as the authors say, “the ‘lower’ type is promoted to the ‘higher’ type,” where the “order” of the types is

\[ \text{char} < \text{short int} < \text{int} < \text{long int} < \text{float} < \text{double} < \text{long double} \]

(We won’t be using long double, so you don’t need to worry about it.) We’ll have more to say about these rules on the next page.

Don’t worry too much for now about the additional rules for unsigned values, because we won’t be using them at first.
Do notice that implicit (automatic) conversions do happen across assignments. It’s perfectly acceptable to assign a char to an int or vice versa, or assign an int to a float or vice versa (or any other combination). Obviously, when you assign a value from a larger type to a smaller one, there’s a chance that it might not fit. Therefore, compilers will often warn you about such assignments.

Page 45: Casts can be a bit confusing at first. A cast is the syntax used to request an explicit type conversion; coercion is just a more formal word for “conversion.” A cast consists of a type name in parentheses and is used as a unary operator. You may have used languages which had conversion operators which looked more like function calls:

```
integer i = 2;
floating f = floating(i); /* not C */
integer i2 = integer(f); /* not C */
```

In C, you accomplish the same thing with casts:

```
int i = 2;
float f = (float)i;
int i2 = (int)f;
```

(Actually, in C, we wouldn’t need casts in those initializations at all, because conversions between int and float are some of the ones that C performs automatically.)

To further understand both how implicit conversions and explicit casts work, let’s study how the implicit conversions would look if we wrote them out explicitly. First we’ll declare a few variables of various types:

```
char c1, c2;
int i1, i2;
long int L1, L2;
double d1, d2;
```

Next we’ll look at the kinds of conversions which C automatically performs when performing arithmetic on two dissimilar types, or when assigning a value to a dissimilar type. The rules are straightforward: when performing arithmetic on two dissimilar types, C converts one or both sides to a common type; and when assigning a value, C converts it to the type of the variable being assigned to.

If we add a char to an int:

```
i2 = c1 + i1;
```

the fourth rule on page 44 tells us to convert the char to an int, as if we’d written

```
i2 = (int)c1 + i1;
```

If we multiply a long int and a double:

```
d2 = L1 * d1;
```

the second rule tells us to convert the long int to a double, as if we’d written
d2 = (double)L1 * d1;

An assignment of a char to an int

i1 = c1;

is as if we’d written

i1 = (int)c1;

and an assignment of a float to an int

i1 = f1;

is as if we’d written

i1 = (int)f1;

Some programmers worry that implicit conversions are somehow unreliable and prefer to insert lots of explicit conversions. I recommend that you get comfortable with implicit conversions—they’re quite useful—and don’t clutter your code with extra casts.

There are a few places where you do need casts, however. Consider the code

i1 = 200;
i2 = 400;
L1 = i1 * i2;

The product 200 x 400 is 80000, which is not guaranteed to fit into an int. (Remember that an int is only guaranteed to hold values up to 32767.) Since 80000 will fit into a long int, you might think that you’re okay, but you’re not: the two sides of the multiplication are of the same type, so the compiler doesn’t see the need to perform any automatic conversions (none of the rules on page 44 apply). The multiplication is carried out as an int, which overflows with unpredictable results, and only after the damage has been done is the unpredictable value converted to a long int for assignment to L1. To get a multiplication like this to work, you have to explicitly convert at least one of the int’s to long int:

L1 = (long int)i1 * i2;

Now, the two sides of the * are of different types, so they’re both converted to long int (by the fifth rule on page 44), and the multiplication is carried out as a long int. If it makes you feel safer, you can use two casts:

L1 = (long int)i1 * (long int)i2;

but only one is strictly required.

A similar problem arises when two integers are being divided. The code

i1 = 1;
f1 = i1 / 2;
does not set f1 to 0.5, it sets it to 0. Again, the two operands of the / operand are already of the same type (the rules on page 44 still don’t apply), so an integer division is performed, which discards any fractional part. (We saw a similar problem in section 1.2 on page 12.) Again, an explicit conversion saves the day:

\[
f1 = (\text{float})i1 / 2;
\]

Alternately, in a case like this, you can use a floating-point constant:

\[
f1 = i1 / 2.0;
\]

In either case, as soon as one of the operands is floating point, the division is carried out in floating point, and you get the result you expect.

Implicit conversions always happen during arithmetic and assignment to variables. The situation is a bit more complicated when functions are being called, however.

The authors use the example of the sqrt function, which is as good an example as any. sqrt accepts an argument of type double and returns a value of type double. If the compiler didn’t know that sqrt took a double, and if you called

\[
sqrt(4);
\]

or

\[
\text{int} \ n = 4;
\text{sqrt}(n);
\]

the compiler would pass an int to sqrt. Since sqrt expects a double, it will not work correctly if it receives an int. Therefore, it was once always necessary to use explicit conversions in cases like this, by calling

\[
sqrt((\text{double})4)
\]

or

\[
sqrt((\text{double})n)
\]

or

\[
sqrt(4.0)
\]

However, it is now possible, with a function prototype, to tell the compiler what types of arguments a function expects. The prototype for sqrt is

\[
\text{double sqrt(double)};
\]

and as long as a prototype is in effect (“in scope,” as the cognoscenti would say), you can call sqrt without worrying about conversions. When a prototype is in effect, the compiler performs implicit conversions during function calls (specifically, while passing the arguments) exactly as it does during simple assignments.
Obviously, using prototypes makes for much safer programming, and it is recommended that you use them whenever possible. For the standard library functions (the ones already written for you), you get prototypes automatically when you include the header files which describe sets of library functions. For example, you get prototypes for all of C’s built-in math functions by putting the line

```c
#include <math.h>
```

at the top of your program. For functions that you write, you can supply your own prototypes, which we’ll be learning more about later.

However, there are a few situations (we’ll talk about them later) where prototypes do not apply, so it’s important to remember that function calls are a bit different and that explicit conversions (i.e. casts) may occasionally be required. Don’t imagine that prototypes are a panacea.

Page 46: Don’t worry about the rand example.

Section 2.8: Increment and Decrement Operators

The distinction between the prefix and postfix forms of ++ and -- will probably seem strained at first, but it will make more sense once we begin using these operators in more realistic situations.

The authors point out that an expression like (i+j)++ is illegal, and it’s worth thinking for a moment about why. The ++ operator doesn’t just mean “add one”; it means “add one to a variable” or “make a variable’s value one more than it was before.” But (i+j) is not a variable, it’s an expression; so there’s no place for ++ to store the incremented result. If you were bound and determined to use ++ here, you’d have to introduce another variable:

```c
int k = i + j;
k++;
```

But really, when you want to add one to an expression, just use

```c
i + j + 1
```

Another unfortunate (and utterly meaningless) example is

```c
i = i++;
```

If you want to increment i (that is, add one to it, and store the result back in i), either use

```c
i = i + 1;
```

or

```c
i++;
```

Don’t try to combine the two.

Page 47: Deep sentence: In a context where no value is wanted, just the incrementing effect, as in
if (c == '\n')
    nl++;  
  
_prefix and postfix are the same._

In other words, when you’re just incrementing some variable, you can use either the nl++ or ++nl form. But when you’re immediately using the result, as in the examples we’ll look at later, using one or the other makes a big difference.

In that light, study one of the examples on this page—squeeze, the modified getline, or strcat—and convince yourself that it would not work if the wrong form of increment (++i or ++j) were used.

You may note that all three examples on pages 47-48 use the postfix form. Postfix increment is probably more common, though prefix definitely has its uses, too.

You may notice the keyword void popping up in a few code examples. void is a type we haven’t met yet; it’s a type with no values and no operations. When a function is declared as “returning” void, as in the squeeze and strcat examples on pages 47 and 48, it means that the function does not return a value. (This was briefly mentioned on page 30 in chapter 1.)

Section 2.9: Bitwise Operators

_Page 48:_ The bitwise operators are definitely a bit (pardon the pun) more esoteric than the parts of C we’ve covered so far (and, indeed, than probably most of C). We won’t concentrate on them, but they do come up all the time, so you should eventually learn enough about them to recognize what they do, even if you don’t use them in any of your own programs for a while. You may skip this section for now, though.

To see what the bitwise operators are doing, it may help to convert to binary for a moment and look at what’s happening to the individual bits. In the example on page 48, suppose that n is 052525, which is 21845 decimal, or 101010101010101 binary. Then n & 0177, in base 2 and base 8 (binary and octal) looks like

101010101010101
& 000000001111111
---------------
1010101

In the second example, if SET_ON is 012 and x is 0, then x | SET_ON looks like

00000000
| 00000101
-----
1010
12

and if x starts out as 402, it looks like

100000010
| 000001010
-----
10001010
412
Note that with &, anywhere we have a 0 we turn bits off, and anywhere we have a 1 we copy bits through from the other side. With |, anywhere we have a 1 we turn bits on, and anywhere we have a 0 we leave bits alone.

You’ll frequently see the word mask used, both as a noun and a verb. You can imagine that we’ve cut a mask or stencil out of cardboard, and are using spray paint to spray through the mask onto some other piece of paper. For |, the holes in the mask are like 1’s, and the spray paint is like 1’s, and we paint more 1’s onto the underlying paper. (If there was already paint under a hole, nothing really changes if we get more paint on it; it’s still a “1”.)

The & operator is a bit harder to fit into this analogy: you can either imagine that the holes in the mask are 1’s and you’re spraying some preservative which will fix some of the underlying bits after which the others will get washed off, or you can imagine that the holes in the mask are 0’s, and you’re spraying some erasing paint or some background color which obliterates anything (i.e. any 1’s, any foreground color) it reaches.

For a bit more information on “bitwise” operations, see the handout, “A Brief Refresher on Some Math Often Used in Computing.”

Page 49: Work through the example at the top of the page, and convince yourself that 1 & 2 is 0 and that 1 && 2 is 1.

The precedence of the bitwise operators is not what you might expect, and explicit parentheses are often needed, as noted in this deep sentence from page 52:

Page52: Deep Sentence: Note that the precedence of the bitwise operators &, ^, and | falls below == and !=. This implies that bit-testing expressions like

\[ \text{if } ((x \& \text{ MASK}) == 0) \ldots \]

\textit{must be fully parenthesized to give proper results.}

Section 2.10: Assignment Operators and Expressions

Page 50: You may wonder what it means to say that “expr is computed only once” since in an assignment like

\[ i = i + 2 \]

we don’t “evaluate” the i on the left hand side of the = at all, we assign to it. The distinction becomes important, however, when the left hand side (expr1) is more complicated than a simple variable. For example, we could add 2 to each of the n cells of an array a with code like

\begin{verbatim}
int i = 0;
while(i < n)
  a[i++] += 2;
\end{verbatim}

If we tried to use the expanded form, we’d get
int i = 0;
while(i < n)
    a[i++] = a[i++] + 2;

and by trying to increment i twice within the same expression we’d get (as we’ll see) undesired, unpredictable, and in fact undefined results. (Of course, a more natural form of this loop would be

for(i = 0; i < n; i++)
    a[i] += 2;

and with the increment of i moved out of the array subscript, it wouldn’t matter so much whether we used a[i] += 2 or a[i] = a[i] + 2.)

Page 51: To make the point more clear, the “complicated expression” without using += would look like


(What’s going on here is that the subexpression yypv[p3+p4] + yypv[p1+p2] is being used as a subscript to determine which cell of the yyval array to increment by 2.)

The sentence on p. 51 that includes the words “the assignment statement has a value” is a bit misleading: an assignment is really an expression in C. Like any expression, it has a value, and it can therefore participate as a subexpression in a larger expression. (If the distinction between the terms “statement” and “expression” seems vague, don’t worry; we’ll start talking about statements in the next chapter.)

Section 2.11: Conditional Expressions

“Ternary” is a ten-dollar word meaning “having three operands.” (It’s analogous to the terms unary and binary, which refer to operators having one and two operands, respectively.) The conditional operator is a bit of a frill, and it’s a bit obscure, so you may skip section 2.11 in the book on first reading, but please read the comments in these notes just below (under the mention of “annoying compulsion”).

Page 52: To see what the ?: operator has bought us, here is what the array-printing loop might look like without it:

for(i = 0; i < n; i++) {
    printf("%6d", a[i]);
    if(i%10==9 || i==n-1)
        printf("\n");
    else
        printf(" ");
}

You may be finding this compulsion to write “compact” or “concise” code using operators like ++ and += and ?: a bit annoying. There are three things to know:

1. In complicated code, these operators allow an economy of expression which is beneficial. Mathematicians are constantly inventing new notations, in which one letter or symbol stands for a complicated expression or operation, in order to solve complicated problems without drowning in so much verbiage that it would be impossible to follow an argument or
check for errors. Computer programs are large and complex, so well-chosen abbreviations can make them easier to work with, too.

2. Some C programmers, it’s true, do take the urge to write succinct or concise code to excess, and end up with cryptic, bewildering, obfuscated, impenetrable messes. (I’m not apologizing for them: I hate overly abbreviated, impossible-to-read code, too!)

3. Since there is overly concise C code out there, it’s occasionally necessary to dissect a piece of it and figure out what it does, so you need to have enough familiarity with these operators, and with some standard, idiomatic ways in which they’re commonly combined, so that you won’t be utterly stymied.

However, there is nothing that says that you have to write concise code yourself. Don’t be lured into thinking that you’re not a “real C programmer” until you routinely and easily write code which no one else can read. Write in a style that’s comfortable to you; don’t be embarrassed if your code seems “simple.” (Actually, the very best code seems simple, too.) With time, you’ll probably come to appreciate at least some of the idioms, and to be comfortable enough with them that you may want to use a few of them yourself, after all.

Section 2.12: Precedence and Order of Evaluation

Note that precedence is not the same thing as order of evaluation. Precedence determines how an expression is parsed, and it has an influence on the order in which parts of it are evaluated, but the influence isn’t as strong as you’d think. Precedence says that in the expression

\[1 + 2 \times 3\]

the multiplication happens before the addition. But if we have several function calls, such as

\[f() + g() \times h()\]

we have no idea which function will be called first; the compiler might arrange to call f() first even though its value won’t be needed until last. If we were to write an abomination like

\[i = 1;\]
\[a[i++] + a[i++] \times a[i++]\]

we would have no way of knowing which order the three increments would happen in, and in fact the compiler wouldn’t have any idea either. We could not argue that since multiplication has higher precedence than addition, and since multiplication associates from left to right, the second i++ would have to happen first, then the third, then the first. (Actually, associativity never says anything about which side of a single binary operator gets evaluated first; associativity says which of several adjacent same-precedence operators happens first.)

In general, you should be wary of ever trying to second-guess the relative order in which the various parts of an expression will be evaluated, with two exceptions:

1. You can obviously assume that precedence will dictate the order in which binary operators are applied. This typically says more than just what order things happens in, but also what
the expression actually means. (In other words, the precedence of * over + says more than that the multiplication “happens first” in 1 + 2 * 3; it says that the answer is 7, not 9.)

2. You can assume that the && and || operators are evaluated left-to-right, and that the right-hand side is not evaluated at all if the left-hand side determines the outcome.

To look at one more example, it might seem that the code

```c
int i = 7;
printf("%d\n", i++ * i++);
```

would have to print 56, because no matter which order the increments happen in, 7x8 is 8x7 is 56. But ++ just says that the increment happens later, not that it happens immediately, so this code could print 49 (if it chose to perform the multiplication first, and both increments later). And, it turns out that ambiguous expressions like this are such a bad idea that the ANSI C Standard does not require compilers to do anything reasonable with them at all, such that the above code might end up printing 42, or 8923409342, or 0, or crashing your computer.

Finally, note that parentheses don’t dictate overall evaluation order any more than precedence does. Parentheses override precedence and say which operands go with which operators, and they therefore affect the overall meaning of an expression, but they don’t say anything about the order of subexpressions or side effects. We could not “fix” the evaluation order of any of the expressions we’ve been discussing by adding parentheses. If we wrote

```c
f() + (g() * h())
```

we still wouldn’t know whether f(), g(), or h() would be called first. (The parentheses would force the multiplication to happen before the addition, but precedence already would have forced that, anyway.) If we wrote

```c
(i++) * (i++)
```

the parentheses wouldn’t force the increments to happen before the multiplication or in any well-defined order; this parenthesized version would be just as undefined as i++ * i++ was.

---

**Page 53: Deep sentence:** Function calls, nested assignment statements, and increment and decrement operators cause “side effects”—some variable is changed as a by-product of the evaluation of an expression.

(There’s a slight inaccuracy in this sentence: *any* assignment expression counts as a side effect.)

It’s these “side effects” that you want to keep in mind when you’re making sure that your programs are well-defined and don’t suffer any of the undefined behavior we’ve been discussing. (When we informally said that complex expressions had several things going on “at once,” we were actually referring to expressions with multiple side effects.) As a general rule, you should make sure that each expression only has one side effect, or if it has several, that different variables are changed by the several side effects.
The moral is that writing code that 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, you won’t be tempted to take advantage of a particular implementation.

The first edition of K&R said:

...if you don’t know how they are done on various machines, that innocence may help to protect you.

I actually prefer the first edition wording. Many textbooks encourage you to write small programs to find out how your compiler implements some of these ambiguous expressions, but it’s just one step from writing a small program to find out, to writing a real program which makes use of what you’ve just learned. And you don’t want to write programs that work only under one particular compiler, that take advantage of the way that compiler (but perhaps no other) happens to implement the undefined expressions. It’s fine to be curious about what goes on “under the hood,” and many of you will be curious enough about what’s going on with these “forbidden” expressions that you’ll want to investigate them, but please keep very firmly in mind that, for real programs, the very easiest way of dealing with ambiguous, undefined expressions (which one compiler interprets one way and another interprets another way and a third crashes on) is not to write them in the first place.