Chapter 3: Control Flow

Section 3.1: Statements and Blocks

Page 55: Deep sentence: There is no semicolon after the right brace that ends a block.

Nothing more to say here, but it’s a frequent point of confusion.

Section 3.2: If-Else

The syntax description here may seem to suggest that statement_1 and statement_2 must be single, simple statements, but, as mentioned in section 3.1, a block of statements enclosed in braces {} is equivalent to a single statement.

Page 56: “Coding shortcuts” like

    if(expression)

    can indeed be cryptic, but they’re also quite common, so you’ll need to be able to recognize them even if you don’t choose to write them in your own code. Whenever you see code like

    if (x)

    or

    if (f())

    where x or f() do not have obvious “Boolean” names, just mentally add in != 0.

    Don’t worry too much if the multiple if/else ambiguity described on page 56 doesn’t make perfect sense; just note the deep sentence:

    …it’s a good idea to use braces when there are nested ifs.

Section 3.3: Else-If

Pages 57-58: Binary search is an extremely important algorithm, but it turns out that it is subtle to get the implementation just right. (It has been observed that although the first binary search was published
in 1946, the first published binary search without bugs did not appear until 1962.) The basic idea is the same as the algorithm we all tend to use when we’re asked to guess a number between 1 and 100: “Is it between 1 and 50? Yes? Okay, is it between 25 and 50? No? Okay, is it between 1 and 12? ... “ (Don’t worry if you can’t follow all of the details of the algorithm or the code on page 58, but do remember to be extremely careful if you’re ever asked to write a binary search routine.)

Section 3.4: Switch

Pages 58-59: We won’t be concentrating on switch statements much (they’re a bit of a luxury; there’s nothing you can do with a switch that you can’t do with an if/else chain, as in section 3.3 on page 57). But they’re quite handy, and good to know about.

The example on page 59 is about as contrived as the example in section 1.6 (page 22) which it replaces, but studying both examples will give you an excellent feel for how a switch statement works, what the if/then statements are that a switch is equivalent to and how to map between the two, and why a switch statement can be convenient.

In the example in the text, note especially the way that ten case labels are attached to one set of statements (ndigit[c-'0']++). As the authors point out, this works because of the way switch cases “fall through,” which is a mixed blessing.

The danger of fall-through is illustrated by:

```c
switch(food) {
    case APPLE:
        printf("apple\n");
    case ORANGE:
        printf("orange\n");
        break;
    default:
        printf("other\n");
}
```

When food is APPLE, this code erroneously prints

apple
orange

because the break statement after the APPLE case was omitted.

Section 3.5: Loops -- While and For

Page 60: Remember that, as always, the statement can be a brace-enclosed block.

Make sure you understand how the for loop

```c
for (expr1; expr2; expr3)
    statement
```
is equivalent to the while loop

```c
expr1;
while (expr2) {
    statement
    expr3;
}
```

There is nothing magical about the three expressions at the top of a for loop; they can be arbitrary expressions, and they’re evaluated just as the expansion into the equivalent while loop would suggest. (Actually, there are two tiny differences: the behavior of continue, which we’ll get to in a bit, and the fact that the test expression, `expr2`, is optional and defaults to “true” for a for loop, but is required for a while loop.)

`for(;;)` is one way of writing an infinite loop in C; the other common one is `while(1)`. Don’t worry about what a break would mean in a loop, we’ll be seeing it in a few more pages.

Pages 60-61: Deep sentences: Whether to use while or for is largely a matter of personal preference ... Nonetheless, it is bad style to force unrelated computations into the initialization and increment of a for, which are better reserved for loop control operations.

In general, the three expressions in a for loop should all manipulate (initialize, test, and increment) the same variable or data structure. If they don’t, they are “unrelated computations,” and a while loop would probably be clearer. (The reason that one loop or the other can be clearer is simply that, when you see a for loop, you expect to see an idiomatic initialize/test/increment of a single variable or data structure, and if the for loop you’re looking at doesn’t end up matching that pattern, you’ve been momentarily misled.)

Page 61: When the authors say that “the index and limit of a C for loop can be altered from within the loop,” they mean that a loop like

```c
int i, n = 10;
for(i = 0; i < n; i++) {
    if(i == 5)
        i++;
    printf("%d\n", i);
    if(i == 8)
        n++;}
```

where `i` and `n` are modified within the loop, is legal. (Obviously, such a loop can be very confusing, so you’ll probably be better off not making use of this freedom too much.)

When they say that “the index variable... retains its value when the loop terminates for any reason,” you may not find this too surprising, unless you’ve used other languages where it’s not the case. The fact that loop control variables retain their values after a loop can make some code much easier to write; for example, the atoi function at the bottom of this page depends on having its `i` counter manipulated by several loops as it steps over three different parts of the string (whitespace, sign, digits) with `i`’s value preserved between each step.

Deep sentence: Each step does its part, and leaves things in a clean state for the next.
This is an extremely important observation on how to write clean code. As you study the atoi code, notice that it falls into three parts, each implementing one step of the pseudocode description: *skip white space, get sign, get integer part and convert it*. At each step, i points at the next character which that step is to inspect. (If a step is skipped, because there is no leading whitespace or no sign, the later steps don’t care.)

You may hear the term invariant used: this refers to some condition which exists at all stages of a program or function. In this case, the invariant is that i always points to the next character to be inspected. Having some well-chosen invariants can make code *much* easier to write and maintain. If there aren’t enough invariants—if i is sometimes the next character to look at and sometimes the character that was just looked at—debugging and maintaining the code can be a nightmare.

In the atoi example, the lines

```c
for (i = 0; isspace(s[i]); i++) /* skip white space */
;
```

are about at the brink of “forcing unrelated computations into the initialization and increment,” especially since so much has been forced into the loop header that there’s nothing left in the body. It would be equally clear to write this part as

```c
i = 0;
while (isspace(s[i]))
  i++;   /* skip white space */
```

The line

```c
sign = (s[i] == '-') ? -1 : 1;
```

may seem a bit cryptic at first, though it’s a textbook example of the use of ?. The line is equivalent to

```c
sign = 1;
if(s[i] == '-')
  sign = -1;
```

*Pages 61-62*: It’s instructive to study this Shell or “gap” sort, but don’t worry if you find it a bit bewildering.

*Deep sentence: Notice how the generality of for makes the outer loop fit the same form as the others, even though it is not an arithmetic progression.*

The point is that loops don’t have to count 0, 1, 2... or 1, 2, 3... . (This one counts n/2, n/4, n/8... . Later we’ll see loops which don’t step over numbers at all.)

*Page 63: Deep sentence: The commas that separate function arguments, variables in declarations, etc. are not comma operators...*

This looks strange, but it’s true. If you say
for (i = 0, j = strlen(s)-1; i < j; i++, j--)

the first comma says to do i = 0 then do j = strlen(s)-1, and the second comma says to do i++ then do j-. However, when you say

gline(line, MAXLINE);

the comma just separates the two arguments line and MAXLINE; they both have to be evaluated, but it doesn’t matter in which order, and they’re both passed to getline. (If the comma in a function call were interpreted as a comma operator, the function would only receive one argument, since the value of the first operand of the comma operator is discarded.) Since the comma operator discards the value of its first operand, its first operand had better have a side effect. The expression

++a,++b

increments a and increments b and (if anyone cares) returns b’s value, but the expression

a+1,b+1

adds 1 to a, discards it, and returns b+1.

If the comma operator isn’t making perfect sense, don’t worry about it for now. You’re most likely to see it in the first or third expression of a for statement, where it has the obvious meaning of separating two (or more) things to do during the initialization or increment step. Just be careful that you don’t accidentally write things like

for(i = 0; j = 0; i < n && j < j; i++; j++) /* WRONG */

or

for(i = 0, j = 0, i < n && j < j, i++, j++) /* WRONG */

The correct form of a multi-index loop is something like

for(i = 0, j = 0; i < n && j < j; i++, j++)

Semicolons always separate the initialization, test, and increment parts; commas may appear within the initialization and increment parts.

Section 3.6: Loops -- Do-while

Page 63: Note the semicolon following the parenthesized expression in the do-while loop; it’s a required part of the syntax.

Make sure you understand the difference between a while loop and a do-while loop. A while loop executes strictly according to its conditional expression: if the expression is never true, the loop executes zero times. The do-while loop, on the other hand, makes an initial “no peek” foray through the loop body no matter what.
To see the difference, let's imagine three different ways of writing the loop in the itoa function on page 64. Suppose we somehow forgot to use a termination condition at all, and wrote something like

```c
for(;;) {
    s[i++] = n % 10 + '0';
    n /= 10;
}
```

Eventually, $n$ becomes zero, but we keep going around the loop, and we convert a number like 123 into a string like "0000000000123", except with an infinite number of leading zeroes. (Mathematically, this is correct, but it's not what we want here, especially if we want our program to use a finite amount of time and space.)

Our next attempt might be

```c
while(n > 0) {
    s[i++] = n % 10 + '0';
    n /= 10;
}
```

so that we stop creating digits when $n$ reaches 0. This works fine for positive numbers, but for 0, it stops too soon: it would convert the number 0 to the empty string "". That's why the do-while loop is appropriate here; the fact that it always makes at least one pass through the loop makes sure that we always generate at least one digit, even if it's 0.

(It's also useful to look at the invariants in this loop: during each trip through the loop, $n$ contains the rest of the number we have to convert, $s[]$ contains the digits we've already converted, and $i$ points at the next cell in $s[]$ which is to receive a digit. Each trip through the loop converts one digit, increments $i$, and divides $n$ by 10.)

**Section 3.7: Break and Continue**

*Pages 64-65:* Note that a break inside a switch inside a loop causes a break out of the switch, while a break inside a loop inside a switch causes a break out of the loop.

Neither break nor continue has any effect on a brace-enclosed block of statements following an if. break causes a break out of the innermost switch or loop, and continue forces the next iteration of the innermost loop.

There is no way of forcing a break or continue to act on an outer loop.

Another example of where continue is useful is when processing data files. It's often useful to allow comments in data files; one convention is that a line beginning with a # character is a comment, and should be ignored by any program reading the file. This can be coded with something like

```c
while(getline(line, MAXLINE) > 0) {
    if(line[0] == '#')
        continue;
```
The alternative, without a continue, would be

```c
while(getline(line, MAXLINE) > 0) {
    if(line[0] != '#') {
        /* process data file line */
    }
}
```

but now the processing of normal data file lines has been made subordinate to comment lines. (Also, as the authors note, it pushes most of the body of the loop over by another tab stop.) Since comments are exceptional, it’s nice to test for them, get them out of the way, and go on about our business, which the code using continue nicely expresses.

Section 3.8: Goto and Labels

Pages 65-66: A tremendous amount of impassioned debate surrounds the lowly goto statement, which exists in many languages. Some people say that gotos are fine; others say that they must never be used. You should definitely shy away from gotos, but don’t be dogmatic about it; some day, you’ll find yourself writing some screwy piece of code where trying to avoid a goto (by introducing extra tests or Boolean control variables) would only make things worse.

Page 66: When you find yourself writing several nested loops in order to search for something, such that you would need to use a goto to break out of all of them when you do find what you’re looking for, it’s often an excellent idea to move the search code out into a separate function. Doing so can make both the “found” and “not found” cases easier to handle. Here’s a slight variation on the example in the middle of page 66, written as a function:

```c
/* return i such that a[i] == b[j] for some j, or -1 if none */
int findequal(int a[], int na, int b[], int nb)
{
    int i, j;
    for(i = 0; i < na; i++) {
        for(j = 0; j < nb; j++) {
            if(a[i] == b[j])
                return i;
        }
    }
    return -1;
}
```

This function can then be called as

```c
i = findequal(a, na, b, nb);
if(i == -1)
    /* didn’t find any common element */
else /* got one */
```

(The only disadvantage here is that it’s trickier to return i and j if we need them both.)