CHAPTER 3: Control Flow

The control-flow statements of a language specify the order in which computations are performed. We have already met the most common control-flow constructions in earlier examples; here we will complete the set, and be more precise about the ones discussed before.

3.1 Statements and Blocks

An expression such as \( x = 0 \) or \( i++ \) or \( \text{printf}(...) \) becomes a statement when it is followed by a semicolon, as in

\[
x = 0; \\
i++; \\
\text{printf}(...);
\]

In C, the semicolon is a statement terminator, rather than a separator as it is in languages like Pascal.

Braces \{ and \} are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an if, else, while, or for are another. (Variables can be declared inside any block; we will talk about this in Chapter 4.) There is no semicolon after the right brace that ends a block.

3.2 If-Else

The if-else statement is used to express decisions. Formally, the syntax is

\[
\text{if } (\text{expression}) \\
\quad \text{statement}_1 \\
\text{else} \\
\quad \text{statement}_2
\]
where the else part is optional. The expression is evaluated; if it is true (that is, if expression has a non-zero value), statement₁ is executed. If it is false (expression is zero) and if there is an else part, statement₂ is executed instead.

Since an if simply tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing

```plaintext
if (expression)
```

instead of

```plaintext
if (expression != 0)
```

Sometimes this is natural and clear; at other times it can be cryptic.

Because the else part of an if-else is optional, there is an ambiguity when an else is omitted from a nested if sequence. This is resolved by associating the else with the closest previous else-less if. For example, in

```plaintext
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

the else goes with the inner if, as we have shown by indentation. If that isn’t what you want, braces must be used to force the proper association:

```plaintext
if (n > 0) {
    if (a > b)
        z = a;
} else
    z = b;
```

The ambiguity is especially pernicious in situations like this:

```plaintext
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
    else /* WRONG */
        printf("error -- n is negative\n");
```

The indentation shows unequivocally what you want, but the compiler doesn’t get the message, and associates the else with the inner if. This kind of bug can be hard to find; it’s a good idea to use braces when there are nested ifs.

By the way, notice that there is a semicolon after \( z = a \) in
if (a > b)
  z = a;
else
  z = b;

This is because grammatically, a statement follows the if, and an expression statement like "z = a;" is always terminated by a semicolon.

3.3 Else-If

The construction

    if (expression)
       statement
     else if (expression)
       statement
     else if (expression)
       statement
     else if (expression)
       statement
     else
       statement

occurs so often that it is worth a brief separate discussion. This sequence of if statements is the most general way of writing a multi-way decision. The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and this terminates the whole chain. As always, the code for each statement is either a single statement, or a group in braces.

The last else part handles the "none of the above" or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing

    else
    statement

can be omitted, or it may be used for error checking to catch an "impossible" condition.

To illustrate a three-way decision, here is a binary search function that decides if a particular value x occurs in the sorted array v. The elements of v must be in increasing order. The function returns the position (a number between 0 and n-1) if x occurs in v, and -1 if not.

Binary search first compares the input value x to the middle element of the array v. If x is less than the middle value, searching focuses on the lower half of the table, otherwise on the upper half. In either case, the next step is to compare x to the middle element of the selected half. This process of dividing the range in two continues until the value is found or the range is empty.
/* binsearch: find x in v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;
    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else    /* found match */
            return mid;
    }
    return -1;  /* no match */
}

The fundamental decision is whether x is less than, greater than, or equal to the middle element v[mid] at each step; this is a natural for else-if.

Exercise 3-1. Our binary search makes two tests inside the loop, when one would suffice (at the price of more tests outside). Write a version with only one test inside the loop and measure the difference in run-time. •

3.4 Switch

The switch statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly.

switch (expression) {
    case const-expr: statements
    case const-expr: statements
    default: statements
}

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled default is executed if none of the other cases are satisfied. A default is optional; if it isn’t there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

In Chapter 1 we wrote a program to count the occurrences of each digit, white space, and all other characters, using a sequence of if ... else if ... else. Here is the same program with a switch:
#include <stdio.h>

main() /* count digits, white space, others */
{
    int c, i, nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i] = 0;
    while ((c = getchar()) != EOF)
    {
        switch (c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            ndigit[c-'0']++;
            break;
        case ' ': case '\n': case '\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
        }
    }
    printf("digits =");
    for (i = 0; i < 10; i++)
        printf(" %d", ndigit[i]);
    printf("", nwhite, white space = %d, other = %d\n",
     nwhite, nother);
    return 0;
}

The break statement causes an immediate exit from the switch. Because cases serve just as labels, after the code for one case is done, execution falls through to the next unless you take explicit action to escape. break and return are the most common ways to leave a switch. A break statement can also be used to force an immediate exit from while, for, and do loops, as will be discussed later in this chapter.

Falling through cases is a mixed blessing. On the positive side, it allows several cases to be attached to a single action, as with the digits in this example. But it also implies that normally each case must end with a break to prevent falling through to the next. Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall-throughs should be used sparingly, and commented.

As a matter of good form, put a break after the last case (the default here) even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.
Exercise 3-2. Write a function escape(s, t) that converts characters like newline and tab into visible escape sequences like \n and \t as it copies the string t to s. Use a switch. Write a function for the other direction as well, converting escape sequences into the real characters. ☐

3.5 Loops—While and For

We have already encountered the while and for loops. In

```c
while (expression)
  statement
```

the expression is evaluated. If it is non-zero, statement is executed and expression is re-evaluated. This cycle continues until expression becomes zero, at which point execution resumes after statement.

The for statement

```c
for (expr1; expr2; expr3)
  statement
```

is equivalent to

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

except for the behavior of continue, which is described in Section 3.7.

Grammatically, the three components of a for loop are expressions. Most commonly, expr1 and expr3 are assignments or function calls and expr2 is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If expr1 or expr3 is omitted, it is simply dropped from the expansion. If the test, expr2, is not present, it is taken as permanently true, so

```c
for (;;) {
  ...
}
```

is an "infinite" loop, presumably to be broken by other means, such as a break or return.

Whether to use while or for is largely a matter of personal preference. For example, in

```c
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
  /* skip white space characters */
```

there is no initialization or re-initialization, so the while is most natural.

The for is preferable when there is a simple initialization and increment, since it keeps the loop control statements close together and visible at the top of
the loop. This is most obvious in

```c
for (i = 0; i < n; i++)
...
```

which is the C idiom for processing the first \( n \) elements of an array, the analog of the Fortran DO loop or the Pascal `for`. The analogy is not perfect, however, since the index and limit of a C `for` loop can be altered from within the loop, and the index variable \( i \) retains its value when the loop terminates for any reason. Because the components of the `for` are arbitrary expressions, `for` loops are not restricted to arithmetic progressions. 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.

As a larger example, here is another version of `atoi` for converting a string to its numeric equivalent. This one is slightly more general than the one in Chapter 2; it copes with optional leading white space and an optional + or - sign. (Chapter 4 shows `atof`, which does the same conversion for floating-point numbers.)

The structure of the program reflects the form of the input:

```c
skip white space, if any
get sign, if any
get integer part and convert it
```

Each step does its part, and leaves things in a clean state for the next. The whole process terminates on the first character that could not be part of a number.

```c
#include <ctype.h>

/* atoi: convert s to integer; version 2 */
int atoi(char s[])
{
    int i, n, sign;
    for (i = 0; isspace(s[i]); i++) /* skip white space */
        ;
    sign = (s[i] == '-' ? -1 : 1);
    if (s[i] == '+' || s[i] == '-') /* skip sign */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

The standard library provides a more elaborate function `strtol` for conversion of strings to long integers; see Section 5 of Appendix B.

The advantages of keeping loop control centralized are even more obvious when there are several nested loops. The following function is a Shell sort for sorting an array of integers. The basic idea of this sorting algorithm, which was
invented in 1959 by D. L. Shell, is that in early stages, far-apart elements are compared, rather than adjacent ones as in simpler interchange sorts. This tends to eliminate large amounts of disorder quickly, so later stages have less work to do. The interval between compared elements is gradually decreased to one, at which point the sort effectively becomes an adjacent interchange method.

```c
/* shellsort: sort v[0]...v[n-1] into increasing order */
void shellsort(int v[], int n)
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
    {
```
The commas that separate function arguments, variables in declarations, etc., are not comma operators, and do not guarantee left to right evaluation.

Comma operators should be used sparingly. The most suitable uses are for constructs strongly related to each other, as in the for loop in reverse, and in macros where a multistep computation has to be a single expression. A comma expression might also be appropriate for the exchange of elements in reverse, where the exchange can be thought of as a single operation:

```c
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
  c = s[i], s[i] = s[j], s[j] = c;
```

Exercise 3-3. Write a function `expand(s1,s2)` that expands shorthand notations like `a-z` in the string `s1` into the equivalent complete list `abc...xyz` in `s2`. Allow for letters of either case and digits, and be prepared to handle cases like `a-b-c` and `a-z0-9` and `-a-z`. Arrange that a leading or trailing - is taken literally. □

### 3.6 Loops—Do-while

As we discussed in Chapter 1, the while and for loops test the termination condition at the top. By contrast, the third loop in C, the do-while, tests at the bottom after making each pass through the loop body; the body is always executed at least once.

The syntax of the do is

```c
do
  statement
  while (expression);
```

The statement is executed, then expression is evaluated. If it is true, statement is evaluated again, and so on. When the expression becomes false, the loop terminates. Except for the sense of the test, do-while is equivalent to the Pascal repeat-until statement.

Experience shows that do-while is much less used than while and for. Nonetheless, from time to time it is valuable, as in the following function `itoa`, which converts a number to a character string (the inverse of `atoi`). The job is slightly more complicated than might be thought at first, because the easy methods of generating the digits generate them in the wrong order. We have chosen to generate the string backwards, then reverse it.
/* itoa: convert n to characters in s */
void itoa(int n, char s[]) {  
    int i, sign;
    if ((sign = n) < 0) /* record sign */
        n = -n; /* make n positive */
    i = 0;
    do { /* generate digits in reverse order */
        s[i++] = n % 10 + '0'; /* get next digit */
    } while ((n /= 10) > 0); /* delete it */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}

The do-while is necessary, or at least convenient, since at least one character must be installed in the array s, even if n is zero. We also used braces around the single statement that makes up the body of the do-while, even though they are unnecessary, so the hasty reader will not mistake the while part for the beginning of a while loop.

Exercise 3-4. In a two's complement number representation, our version of itoa does not handle the largest negative number, that is, the value of n equal to -(2^wordsize-1). Explain why not. Modify it to print that value correctly, regardless of the machine on which it runs. □

Exercise 3-5. Write the function itob(n,s,b) that converts the integer n into a base b character representation in the string s. In particular, itob(n,s,16) formats n as a hexadecimal integer in s. □

Exercise 3-6. Write a version of itoa that accepts three arguments instead of two. The third argument is a minimum field width; the converted number must be padded with blanks on the left if necessary to make it wide enough. □

3.7 Break and Continue

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The break statement provides an early exit from for, while, and do, just as from switch. A break causes the innermost enclosing loop or switch to be exited immediately.

The following function, trim, removes trailing blanks, tabs, and newlines from the end of a string, using a break to exit from a loop when the rightmost non-blank, non-tab, non-newline is found.
/* trim: remove trailing blanks, tabs, newlines */
int trim(char s[])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != 't' && s[n] != '\n')
for (...) 
  for (...) {
      ...
      if (disaster)
          goto error;
  }
  ...

error:
    clean up the mess

This organization is handy if the error-handling code is non-trivial, and if errors can occur in several places.

A label has the same form as a variable name, and is followed by a colon. It can be attached to any statement in the same function as the goto. The scope of a label is the entire function.

As another example, consider the problem of determining whether two arrays a and b have an element in common. One possibility is

    for (i = 0; i < n; i++)
      for (j = 0; j < m; j++)
          if (a[i] == b[j])
              goto found;
    /* didn’t find any common element */
    ...

found:
    /* got one:  a[i] == b[j] */
    ...

Code involving a goto can always be written without one, though perhaps at the price of some repeated tests or an extra variable. For example, the array search becomes

    found = 0;
    for (i = 0; i < n && !found; i++)
      for (j = 0; j < m && !found; j++)
          if (a[i] == b[j])
              found = 1;
    if (found)
        /* got one:  a[i-1] == b[j-1] */
        ...
    else
        /* didn’t find any common element */
        ...

With a few exceptions like those cited here, code that relies on goto statements is generally harder to understand and to maintain than code without gotos. Although we are not dogmatic about the matter, it does seem that goto statements should be used rarely, if at all.