CHAPTER 3: CONTROL FLOW

The control flow statements of a language specify the order in which   
computations are done. We have already met the most common control   
flow constructions of C 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 printf ( . .. ) becomes a   
*statement* when it is followed by a semicolon, as in

x = 0;

i++;

printf (...);

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

The 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 actually be   
declared inside *any* block; we will talk about this in Chapter 4.) There is   
never a semicolon after the right brace that ends a block.

3.2 If-Else

The if—else statement is used to make decisions. Formally, the syn­   
tax is

if *(expression)*

*statement-1*

else

*statement-2*

where the else part is optional. The *expression* is evaluated; if it is "true"

51

52 THE C PROGRAMMING LANGUAGE CHAPTER 3

(that is, if *expression* has a non-zero value), *statement-1* is done. If it is   
"false" *(expression* is zero) and if there is an else part, *statement-2* is exe­   
cuted instead.

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

if *(expression)*

instead of

if *(expression != 0)*

Sometimes this is natural and clear; at other times it is 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 in   
the usual way — the else is associated with the closest previous else-less   
if. For example, in

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:

if (n > 0) (

if (a > b)

z = a;

else

z = b;

The ambiguity is especially pernicious in situations like:

if (n > 0)

for (i = 0; i < n; i++)   
if (s[i] > 0) (

printf("...");   
return(i);

else /\* WRONG \*/

printf("error - n is zero\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 very hard to find.

By the way, notice that there is a semicolon after z = a in

CHAPTER 3 CONTROL FLOW 53

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

*statement*

occurs so often that it is worth a brief separate discussion. This sequence of   
if's is the most general way of writing a multi-way decision. The   
*expression's* are evaluated in order; if any *expression* is true, the *statement*    
associated with it is executed, and this terminates the whole chain. 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 was 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 "impossi­   
ble" 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.

54 THE C PROGRAMMING LANGUAGE CHAPTER 3

binary(x, v, n) /\* find x in v[0] ... v[n-1] \*/   
int x, v[], 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);

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.

3.4 Switch

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

CHAPTER 3 CONTROL FLOW 55

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 space = %d, other =

nwhite, nother);

The switch evaluates the integer expression in parentheses (in this   
program the character c) and compares its value to all the cases. Each case   
must be labeled by an integer or character constant or constant expression.   
If a case matches the expression value, execution starts at that case. The   
case labeled default is executed if none of the other cases is satisfied. A   
default is optional; if it isn't there and if none of the cases matches, no   
action at all takes place. Cases and default can occur in any order. Cases   
must all be different.

56 THE C PROGRAMMING LANGUAGE CHAPTER 3

The break statement causes an immediate exit from the switch.   
Because cases serve just as labels, after the code for one case is done, exe­   
cution *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 well, as will be discussed later in this chapter.

Falling through cases is a mixed blessing. On the positive side, it allows   
multiple cases for a single action, as with the blank, tab or newline 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 pro­   
gram is modified. With the exception of multiple labels for a single compu­   
tation, fall-throughs should be used sparingly.

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 3-1. Write a function expand (s, t) which converts characters   
like newline and tab into visible escape sequences like \n and \t as it   
copies the string s to t. Use a switch.

3.5 Loops — While and For

We have already encountered the while and for loops. In

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

for *(exprl ; expr2 ; expr3)   
statement*

is equivalent to

*exprl ;*

while *(expr2 ) (*

*statement   
expr3;*

Grammatically, the three components of a for are expressions. Most com­   
monly, *exprl* and *expr3* are assignments or function calls and *expr2* is a rela­   
tional expression. Any of the three parts can be omitted, although the

CHAPTER 3 CONTROL FLOW 57

semicolons must remain. If *exprl* or *expr3* is left out, i is simply dropped   
from the expansion. If the test, *expr2,* is not present, it is taken as per­   
manently true, so

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 taste. For exam­   
ple, in

while ( (c = getchar () ) == " I I c == ' \n' I I c == i\t')   
/\* skip white space characters \*/

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

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

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

which is the C idiom for processing the first N elements of an array, the ana­   
log of the Fortran or PL/I DO loop. The analogy is not perfect, however,   
since the limits of a for loop can be altered from within the loop, and the   
controlling variable i retains its value when the loop terminates for any rea­   
son. 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 a for; it is 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 more general; 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 basic structure of the program reflects the form of the input:

*skip white space, if any*

*get sign, if any*

*get integer part, 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.

58 THE C PROGRAMMING LANGUAGE CHAPTER 3

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

int i, n, sign;

for (i=0; s[i]==" II s[i]==1\n' II s[i]=='\t'; i++)

/\* skip white space \*/

sign = 1;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| if | (s[i] == '+' II s[i] == | |  | /\* sign \*/ |
|  | sign = | (s[i++]=='+') ? 1 : | -1; |  |
| for | (n = 0; | s[i] >= '0' && s[i] | <= | '9'; i++) |
|  | n = 10 | \* n + s[i] - '0'; |  |  |

return(sign \* n);

The advantages of keeping loop control centralized are even more obvi­   
ous when there are several nested loops. The following function is a Shell   
sort for sorting an array of integers. The basic idea of the Shell sort is that   
in early stages, far-apart elements are compared, rather than adjacent ones,   
as in simple 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.

shell(v, n) /\* sort v[0]...v[n-1] into increasing order \*/   
int v[], n;

int gap, i, j, temp;

for (gap = n/2; gap > 0; gap /= 2)

for (i = gap; i < n; i++)

for (j=i-gap; j>=0 && v[j]>v[j+gap]; j—gap)

temp = v[j];

v[j] = v[j+gap];

v[j+gap] = temp;

There are three nested loops. The outermost loop controls the gap between   
compared elements, shrinking it from n/2 by a factor of two each pass until   
it becomes zero. The middle loop compares each pair of elements that is   
separated by gap; the innermost loop reverses any that are out of order.   
Since gap is eventually reduced to one, all elements are eventually ordered   
correctly. Notice that the generality of the for makes the outer loop fit the   
same form as the others, even though it is not an arithmetic progression.

One final C operator is the comma " ", which most often finds use in   
the for statement. A pair of expressions separated by a comma is

CHAPTER 3 CONTROL FLOW 59

evaluated left to right, and the type and value of the result are the type and   
value of the right operand. Thus in a for statement, it is possible to place   
multiple expressions in the various parts, for example to process two indices   
in parallel. This is illustrated in the function reverse (s), which reverses   
the string s in place.

reverse (s) /\* reverse string s in place \*/

char s[];

{

int c, i, j;

for (i = 0, j = strlen(s)-1; i < j; i++, j - -) (

c = s[i];

s[i] = s[j];

s[j] = c;

The commas that separate function arguments, variables in declarations,   
etc., are *not* comma operators, and do *not* guarantee left to right evaluation.

Exercise 3-2. Write a function expand (s1 , s2) which expands short­   
hand 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. (A useful   
convention is that a leading or trailing - is taken literally.) 0

3.6 Loops — Do-while

The while and for loops share the desirable attribute of testing the   
termination condition at the top, rather than at the bottom, as we discussed   
in Chapter 1. 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 is

do

*statement*

while *(expression) ;*

The *statement* is executed, then *expression* is evaluated. If it is true, *state­*   
*ment* is evaluated again, and so on. If the expression becomes false, the   
loop terminates.

As might be expected, do-while is much less used than while and   
for, accounting for perhaps five percent of all loops. Nonetheless, it is   
from time to time valuable, as in the following function itoa, which con­   
verts 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

60 THE C PROGRAMMING LANGUAGE CHAPTER3

methods of generating the digits generate them in the wrong order. We   
have chosen to generate the string backwards, then reverse it.

itoa(n, s) /\* convert n to characters in s \*/

char s[];

int n;

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 char­   
acter must be installed in the array s, regardless of the value of n. 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 3-3. In a 2's complement number representation, our version of   
itoa does not handle the largest negative number, that is, the value of *n*    
equal to -(2wordsize-1). Explain why not. Modify it to print that value   
correctly, regardless of the machine it runs on. 0

Exercise 3-4. Write the analogous function itob (n, s) which converts   
the unsigned integer n into a binary character representation in s. Write   
itoh, which converts an integer to hexadecimal representation. 0

Exercise 3-5. Write a version of itoa which 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. 0

CHAPTER 3 CONTROL FLOW 61   
3.7 Break

It is sometimes convenient to be able to control loop exits 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 statement   
causes the innermost enclosing loop (or switch) to be exited immediately.

The following program removes trailing blanks and tabs from the end of   
each line of input, using a break to exit from a loop when the rightmost   
non-blank, non-tab is found.

#define MAXLINE 1000

main() /\* remove trailing blanks and tabs \*/

int n;

char line[MAXLINE];

while ((n = getline(line, MAXLINE)) > 0) (

while (--n >= 0)

if (line[n] != " St& line[n]

&& line[n] != '\n')

break;

line[n+1] =

printf("%s\n", line);

g e tl ine returns the length of the line. The inner while loop starts at   
the last character of line (recall that --n decrements n before using the   
value), and scans backwards looking for the first character that is not a   
blank, tab or newline. The loop is broken when one is found, or when n   
becomes negative (that is, when the entire line has been scanned). You   
should verify that this is correct behavior even when the line contains only   
white space characters.

An alternative to break is to put the testing in the loop itself:

while ((n = getline(line, MAXLINE)) > 0) (

while (--n >= 0

&& (line[n]==" II line[n]=='\t' II line[n]=='\n'))

* • •

This is inferior to the previous version, because the test is harder to under­   
stand. Tests which require a mixture of &&, I I, ! , or parentheses should   
generally be avoided.

62 THE C PROGRAMMING LANGUAGE CHAPTER 3

3.8 Continue

The continue statement is related to break, but less often used; it   
causes the *next iteration* of the enclosing loop (for, while, do) to begin.   
En the while and do, this means that the test part is executed immediately;   
in the for, control passes to the re-initialization step. (continue applies   
only to loops, not to switch. A continue inside a switch inside a loop   
causes the next loop iteration.)

As an example, this fragment processes only positive elements in the   
array a; negative values are skipped.

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

if (a[i] < 0) /\* skip negative elements \*/

continue;

. . /\* do positive elements \*/   
•

The continue statement is often used when the part of the loop that fol­   
lows is complicated, so that reversing a test and indenting another level   
would nest the program too deeply.

Exercise 3-6. Write a program which copies its input to its output, except   
that it prints only one instance from each group of adjacent identical lines.   
(This is a simple version of the UNIX utility *uniq.)* CI

3.9 Goto's and Labels

C provides the infinitely-abusable goto statement, and labels to branch   
to. Formally, the goto is never necessary, and in practice it is almost   
always easy to write code without it. We have not used goto in this book.

Nonetheless, we will suggest a few situations where goto's may find a   
place. The most common use is to abandon processing in some deeply   
nested structure, such as breaking out of two loops at once. The break   
statement cannot be used directly since it leaves only the innermost loop.   
Thus:

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

CHAPTER 3 CONTROL FLOW 63

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.

As another example, consider the problem of finding the first negative   
element in a two-dimensional array. (Multi-dimensional arrays are discussed   
in Chapter 5.) One possibility is

for (i = 0; i < N; i++)   
for (j = 0; j < M; j++)   
if (v[il [i] < 0)

goto found;

/\* didn't find \*/

found:

/\* found one at position i, 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 exam­   
ple, the array search becomes

found = 0;

for (i = 0; i < N && !found; i++)

for (j = 0; j < M && !found; j++)

found = v[i][j] < 0;

if (found)

/\* it was at i-1, j-1 \*/

* • •

else

/\* not found \*/

* • •

Although we are not dogmatic about the matter, it does seem that goto   
statements should be used sparingly, if at all.