Chapter 3: Statements and Control Flow

Statements are the ``steps'' of a program. Most statements compute and assign values or call functions, but we will eventually meet several other kinds of statements as well. By default, statements are executed in sequence, one after another. We can, however, modify that sequence by using *control flow constructs* which arrange that a statement or group of statements is executed only if some condition is true or false, or executed over and over again to form a *loop*. (A somewhat different kind of control flow happens when we call a function: execution of the caller is suspended while the called function proceeds. We'll discuss functions in chapter 5.)

My definitions of the terms *statement* and *control flow* are somewhat circular. A statement is an element within a program which you can apply control flow to; control flow is how you specify the order in which the statements in your program are executed. (A weaker definition of a statement might be ``a part of your program that does something,'' but this definition could as easily be applied to expressions or functions.)

3.1 Expression Statements

[This section corresponds to K&R Sec. 3.1]

Most of the statements in a C program are *expression statements*. An expression statement is simply an expression followed by a semicolon. The lines

```
        i = 0;
        i = i + 1;
and
        printf("Hello, world!\n");
```

are all expression statements. (In some languages, such as Pascal, the semicolon separates statements, such that the last statement is not followed by a semicolon. In C, however, the semicolon is a statement terminator; all simple statements are followed by semicolons. The semicolon is also used for a few other things in C; we've already seen that it terminates declarations, too.)

Expression statements do all of the real work in a C program. Whenever you need to compute new values for variables, you'll typically use expression statements (and they'll typically contain assignment operators). Whenever you want your program to do something visible, in the real world, you'll typically call a function (as part of an expression statement). We've already seen the most basic example: calling the function `printf` to print text to the screen. But anything else you might do--read or write a disk file, talk to a modem or printer, draw pictures on the screen--will also involve function calls. (Furthermore, the functions you call to do these things are usually different depending on which operating system you're using. The C language does not define them, so we won't be talking about or using them much.)

Expressions and expression statements can be arbitrarily complicated. They don't have to consist of exactly one simple function call, or of one simple assignment to a variable. For one thing, many functions return values, and the values they return can then be used by other parts of the expression. For example, C provides a `sqrt` (square root) function, which we might use to compute the hypotenuse of a right triangle like this:

```
c = sqrt(a*a + b*b);
```

To be useful, an expression statement must do something; it must have some lasting effect on the state of the program. (Formally, a useful statement must have at least one *side effect*.) The first two sample expression statements in this section (above) assign new values to the variable `i`, and the third one calls `printf` to print something out, and these are good examples of statements that do something useful.

(To make the distinction clear, we may note that degenerate constructions such as

```
0;
i;
```

or

```
i + 1;
```

are syntactically valid statements--they consist of an expression followed by a semicolon--but in each case, they compute a value without doing anything with it, so the computed value is discarded, and the statement is useless. But if the ``degenerate'' statements in this paragraph don't make much sense to you, don't worry; it's because they, frankly, don't make much sense.)

It's also possible for a single expression to have multiple side effects, but it's easy for such an expression to be (a) confusing or (b) undefined. For now, we'll only be looking at expressions (and, therefore, statements) which do one well-defined thing at a time.

3.2 `if` Statements

[This section corresponds to K&R Sec. 3.2]

The simplest way to modify the control flow of a program is with an `if` statement, which in its simplest form looks like this:

```
if(x > max)
        max = x;
```

Even if you didn't know any C, it would probably be pretty obvious that what happens here is that if `x` is greater than `max`, `x` gets assigned to `max`. (We'd use code like this to keep track of the maximum value of `x` we'd seen--for each new `x`, we'd compare it to the old maximum value `max`, and if the new value was greater, we'd update `max`.)

More generally, we can say that the syntax of an `if` statement is:

```
if( expression )
        statement
```

where *expression* is any expression and *statement* is any statement.

What if you have a series of statements, all of which should be executed together or not at all depending on whether some condition is true? The answer is that you enclose them in braces:

```
if( expression )
        {
        statement<sub>1</sub>
        statement<sub>2</sub>
        statement<sub>3</sub>
        }
```

As a general rule, anywhere the syntax of C calls for a statement, you may write a series of statements enclosed by braces. (You do not need to, and should not, put a semicolon after the closing brace, because the series of statements enclosed by braces is not itself a simple expression statement.)

An `if` statement may also optionally contain a second statement, the ``else` clause," which is to be executed if the condition is not met. Here is an example:

```
if(n > 0)
        average = sum / n;
else    {
        printf("can't compute average\n");
        average = 0;
        }
```

The first statement or block of statements is executed if the condition *is* true, and the second statement or block of statements (following the keyword `else`) is executed if the condition is *not* true. In this example, we can compute a meaningful average only if `n` is greater than 0; otherwise, we print a message saying that we cannot compute the average. The general syntax of an `if` statement is therefore

```
if( expression )
        statement<sub>1</sub>
else
        statement<sub>2</sub>
```

(where both *statement*`<sub>`*1*`</sub>` and *statement*`<sub>`*2*`</sub>` may be lists of statements enclosed in braces).

It's also possible to nest one `if` statement inside another. (For that matter, it's in general possible to nest any kind of statement or control flow construct within another.) For example, here is a little piece of code which decides roughly which quadrant of the compass you're walking into, based on an `x` value which is positive if you're walking east, and a `y` value which is positive if you're walking north:

```
if(x > 0)
        {
        if(y > 0)
                printf("Northeast.\n");
        else    printf("Southeast.\n");
        }
```

```
else    {
        if(y > 0)
                printf("Northwest.\n");
        else    printf("Southwest.\n");
        }
```

When you have one `if` statement (or loop) nested inside another, it's a very good idea to use explicit braces `{}`, as shown, to make it clear (both to you and to the compiler) how they're nested and which `else` goes with which `if`. It's also a good idea to indent the various levels, also as shown, to make the code more readable to humans. Why do both? You use indentation to make the code visually more readable to yourself and other humans, but the compiler doesn't pay attention to the indentation (since all whitespace is essentially equivalent and is essentially ignored). Therefore, you also have to make sure that the punctuation is right.

Here is an example of another common arrangement of `if` and `else`. Suppose we have a variable `grade` containing a student's numeric grade, and we want to print out the corresponding letter grade. Here is code that would do the job:

```
if(grade >= 90)
        printf("A");
else if(grade >= 80)
        printf("B");
else if(grade >= 70)
        printf("C");
else if(grade >= 60)
        printf("D");
else    printf("F");
```

What happens here is that exactly one of the five `printf` calls is executed, depending on which of the conditions is true. Each condition is tested in turn, and if one is true, the corresponding statement is executed, and the rest are skipped. If none of the conditions is true, we fall through to the last one, printing ``F''.

In the cascaded `if`/`else`/`if`/`else`/... chain, each `else` clause is another `if` statement. This may be more obvious at first if we reformat the example, including every set of braces and indenting each `if` statement relative to the previous one:

```
if(grade >= 90)
        {
        printf("A");
        }
else    {
        if(grade >= 80)
                {
                printf("B");
                }
        else    {
                if(grade >= 70)
                        {
                        printf("C");
                        }
                else    {
                        if(grade >= 60)
```

```
                        {
                        printf("D");
                        }
                else    {
                        printf("F");
                        }
                }
        }
```

By examining the code this way, it should be obvious that exactly one of the `printf` calls is executed, and that whenever one of the conditions is found true, the remaining conditions do not need to be checked and none of the later statements within the chain will be executed. But once you've convinced yourself of this and learned to recognize the idiom, it's generally preferable to arrange the statements as in the first example, without trying to indent each successive `if` statement one tabstop further out. (Obviously, you'd run into the right margin very quickly if the chain had just a few more cases!)

3.3 Boolean Expressions

An `if` statement like

```
if(x > max)
        max = x;
```

is perhaps deceptively simple. Conceptually, we say that it checks whether the condition `x > max` is ``true'' or ``false''. The mechanics underlying C's conception of ``true'' and ``false,'' however, deserve some explanation. We need to understand how true and false values are represented, and how they are interpreted by statements like `if`.

As far as C is concerned, a true/false condition can be represented as an integer. (An integer can represent many values; here we care about only two values: ``true'' and ``false.'' The study of mathematics involving only two values is called Boolean algebra, after George Boole, a mathematician who refined this study.) In C, ``false'' is represented by a value of 0 (zero), and ``true'' is represented by any value that is nonzero. Since there are many nonzero values (at least 65,534, for values of type `int`), when we have to pick a specific value for ``true,'' we'll pick 1.

The *relational operators* such as `<`, `<=`, `>`, and `>=` are in fact operators, just like `+`, `-`, `*`, and `/`. The relational operators take two values, look at them, and ``return'' a value of 1 or 0 depending on whether the tested relation was true or false. The complete set of relational operators in C is:

```
<       less than
<=      less than or equal
>       greater than
>=      greater than or equal
==      equal
!=      not equal
```

For example, `1 < 2` is 1, `3 > 4` is 0, `5 == 5` is 1, and `6 != 6` is 0.

We've now encountered perhaps the most easy-to-stumble-on ``gotcha!'' in C: the equality-testing operator is ==, not a single =, which is assignment. If you accidentally write

```
if(a = 0)
```

(and you probably will at some point; everybody makes this mistake), it will *not* test whether `a` is zero, as you probably intended. Instead, it will *assign* 0 to `a`, and then perform the ``true'' branch of the `if` statement if `a` is *non*zero. But `a` will have just been assigned the value 0, so the ``true'' branch will never be taken! (This could drive you crazy while debugging--you wanted to do something if `a` was 0, and after the test, `a` *is* 0, whether it was supposed to be or not, but the ``true'' branch is nevertheless not taken.)

The relational operators work with arbitrary numbers and generate true/false values. You can also combine true/false values by using the *Boolean operators*, which take true/false values as operands and compute new true/false values. The three Boolean operators are:

```
&&      and
||      or
!       not (takes one operand; ``unary'')
```

The `&&` (``and'') operator takes two true/false values and produces a true (1) result if both operands are true (that is, if the left-hand side is true **and** the right-hand side is true). The `||` (``or'') operator takes two true/false values and produces a true (1) result if either operand is true. The `!` (``not'') operator takes a single true/false value and negates it, turning false to true and true to false (0 to 1 and nonzero to 0).

For example, to test whether the variable `i` lies between 1 and 10, you might use

```
if(1 < i && i < 10)
        ...
```

Here we're expressing the relation ```i` is between 1 and 10'' as ``1 is less than `i` **and** `i` is less than 10.''

It's important to understand why the more obvious expression

```
if(1 < i < 10)                  /* WRONG */
```

would not work. The expression `1 < i < 10` is parsed by the compiler analogously to `1 + i + 10`. The expression `1 + i + 10` is parsed as `(1 + i) + 10` and means ``add 1 to `i`, and then add the result to 10.'' Similarly, the expression `1 < i < 10` is parsed as `(1 < i) < 10` and means ``see if 1 is less than `i`, and then see if the result is less than 10.'' But in this case, ``the result'' is 1 or 0, depending on whether `i` is greater than 1. Since both 0

and 1 are less than 10, the expression `1 < i < 10` would *always* be true in C, regardless of the value of `i`!

Relational and Boolean expressions are usually used in contexts such as an `if` statement, where something is to be done or not done depending on some condition. In these cases what's actually checked is whether the expression representing the condition has a zero or nonzero value. As long as the expression is a relational or Boolean expression, the interpretation is just what we want. For example, when we wrote

```
if(x > max)
```

the `>` operator produced a 1 if `x` was greater than `max`, and a 0 otherwise. The `if` statement interprets 0 as false and 1 (or any nonzero value) as true.

But what if the expression is not a relational or Boolean expression? As far as C is concerned, the controlling expression (of conditional statements like `if`) can in fact be *any* expression: it doesn't have to ``look like'' a Boolean expression; it doesn't have to contain relational or logical operators. All C looks at (when it's evaluating an `if` statement, or anywhere else where it needs a true/false value) is whether the expression evaluates to 0 or nonzero. For example, if you have a variable `x`, and you want to do something if `x` is nonzero, it's possible to write

```
if(x)
        statement
```

and the statement will be executed if `x` is nonzero (since nonzero means ``true'').

This possibility (that the controlling expression of an `if` statement doesn't have to ``look like'' a Boolean expression) is both useful and potentially confusing. It's useful when you have a variable or a function that is ``conceptually Boolean,'' that is, one that you consider to hold a true or false (actually nonzero or zero) value. For example, if you have a variable `verbose` which contains a nonzero value when your program should run in verbose mode and zero when it should be quiet, you can write things like

```
if(verbose)
        printf("Starting first pass\n");
```

and this code is both legal and readable, besides which it does what you want. The standard library contains a function `isupper()` which tests whether a character is an upper-case letter, so if `c` is a character, you might write

```
if(isupper(c))
        ...
```

Both of these examples (`verbose` and `isupper()`) are useful and readable.

However, you will eventually come across code like

```
if(n)
        average = sum / n;
```

where `n` is just a number. Here, the programmer wants to compute the average only if `n` is nonzero (otherwise, of course, the code would divide by 0), and the code works, because,

in the context of the `if` statement, the trivial expression `n` is (as always) interpreted as ``true'' if it is nonzero, and ``false'' if it is zero.

``Coding shortcuts'' like these can seem 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, you can read them as ``if `x` is nonzero'' or ``if `f()` returns nonzero.''

3.4 `while` Loops

[This section corresponds to half of K&R Sec. 3.5]

Loops generally consist of two parts: one or more *control expressions* which (not surprisingly) control the execution of the loop, and the *body*, which is the statement or set of statements which is executed over and over.

The most basic *loop* in C is the `while` loop. A `while` loop has one control expression, and executes as long as that expression is true. This example repeatedly doubles the number 2 (2, 4, 8, 16, ...) and prints the resulting numbers as long as they are less than 1000:

```
int x = 2;

while(x < 1000)
        {
        printf("%d\n", x);
        x = x * 2;
        }
```

(Once again, we've used braces `{}` to enclose the group of statements which are to be executed together as the body of the loop.)

The general syntax of a `while` loop is

```
while( expression )
        statement
```

A `while` loop starts out like an `if` statement: if the condition expressed by the *expression* is true, the *statement* is executed. However, after executing the statement, the condition is tested again, and if it's still true, the statement is executed again. (Presumably, the condition depends on some value which is changed in the body of the loop.) As long as the condition remains true, the body of the loop is executed over and over again. (If the condition is false right at the start, the body of the loop is not executed at all.)

As another example, if you wanted to print a number of blank lines, with the variable `n` holding the number of blank lines to be printed, you might use code like this:

```
while(n > 0)
        {
        printf("\n");
        n = n - 1;
        }
```

After the loop finishes (when control ``falls out'' of it, due to the condition being false), `n` will have the value 0.

You use a `while` loop when you have a statement or group of statements which may have to be executed a number of times to complete their task. The controlling expression represents the condition ``the loop is not done'' or ``there's more work to do.'' As long as the expression is true, the body of the loop is executed; presumably, it makes at least some progress at its task. When the expression becomes false, the task is done, and the rest of the program (beyond the loop) can proceed. When we think about a loop in this way, we can seen an additional important property: if the expression evaluates to ``false'' before the very first trip through the loop, we make *zero* trips through the loop. In other words, if the task is already done (if there's no work to do) the body of the loop is not executed at all. (It's always a good idea to think about the ``boundary conditions'' in a piece of code, and to make sure that the code will work correctly when there is no work to do, or when there is a trivial task to do, such as sorting an array of one number. Experience has shown that bugs at boundary conditions are quite common.)

3.5 `for` Loops

[This section corresponds to the other half of K&R Sec. 3.5]

Our second loop, which we've seen at least one example of already, is the `for` loop. The first one we saw was:

```
for (i = 0; i < 10; i = i + 1)
        printf("i is %d\n", i);
```

More generally, the syntax of a `for` loop is

```
for( expr<sub>1</sub> ; expr<sub>2</sub> ; expr<sub>3</sub> )
        statement
```

(Here we see that the `for` loop has three control expressions. As always, the *statement* can be a brace-enclosed block.)

Many loops are set up to cause some variable to step through a range of values, or, more generally, to set up an initial condition and then modify some value to perform each succeeding loop as long as some condition is true. The three expressions in a `for` loop encapsulate these conditions: *expr*`<sub>`*1*`</sub>` sets up the initial condition, *expr*`<sub>`*2*`</sub>` tests whether another trip through the loop should be taken, and *expr*`<sub>`*3*`</sub>` increments or updates things after each trip through the loop and prior to the next one. In our first example, we had `i = 0` as *expr*`<sub>`*1*`</sub>`, `i < 10`

as *expr₂*, `i = i + 1` as *expr₃*, and the call to `printf` as *statement*, the body of the loop. So the loop began by setting `i` to 0, proceeded as long as `i` was less than 10, printed out `i`'s value during each trip through the loop, and added 1 to `i` between each trip through the loop.

When the compiler sees a `for` loop, first, *expr₁* is evaluated. Then, *expr₂* is evaluated, and if it is true, the body of the loop (*statement*) is executed. Then, *expr₃* is evaluated to go to the next step, and *expr₂* is evaluated again, to see if there *is* a next step. During the execution of a `for` loop, the sequence is:

```
expr<sub>1</sub>
expr<sub>2</sub>
statement
expr<sub>3</sub>
expr<sub>2</sub>
statement
expr<sub>3</sub>
...
expr<sub>2</sub>
statement
expr<sub>3</sub>
expr<sub>2</sub>
```

The first thing executed is *expr₁*. *expr₃* is evaluated after *every* trip through the loop. The last thing executed is always *expr₂*, because when *expr₂* evaluates false, the loop exits.

All three expressions of a `for` loop are optional. If you leave out *expr₁*, there simply is no initialization step, and the variable(s) used with the loop had better have been initialized already. If you leave out *expr₂*, there is no test, and the default for the `for` loop is that another trip through the loop should be taken (such that unless you break out of it some other way, the loop runs forever). If you leave out *expr₃*, there is no increment step.

The semicolons separate the three controlling expressions of a `for` loop. (These semicolons, by the way, have nothing to do with statement terminators.) If you leave out one or more of the expressions, the semicolons remain. Therefore, one way of writing a deliberately infinite loop in C is

```
for(;;)
        ...
```

It's useful to compare C's `for` loop to the equivalent loops in other computer languages you might know. The C loop

```
for(i = x; i <= y; i = i + z)
```

is roughly equivalent to:

```
for I = X to Y step Z   (BASIC)
```

```
do 10 i=x,y,z          (FORTRAN)

for i := x to y        (Pascal)
```

In C (unlike FORTRAN), if the test condition is false before the first trip through the loop, the loop won't be traversed at all. In C (unlike Pascal), a loop control variable (in this case, `i`) is guaranteed to retain its final value after the loop completes, and it is also legal to modify the control variable within the loop, if you really want to. (When the loop terminates due to the test condition turning false, the value of the control variable after the loop will be the first value for which the condition failed, not the last value for which it succeeded.)

It's also worth noting that a `for` loop can be used in more general ways than the simple, iterative examples we've seen so far. The ``control variable'' of a `for` loop does not have to be an integer, and it does not have to be incremented by an additive increment. It could be ``incremented'' by a multiplicative factor (1, 2, 4, 8, ...) if that was what you needed, or it could be a floating-point variable, or it could be another type of variable which we haven't met yet which would step, not over numeric values, but over the elements of an array or other data structure. Strictly speaking, a `for` loop doesn't have to have a ``control variable'' at all; the three expressions can be anything, although the loop will make the most sense if they are related and together form the expected initialize, test, increment sequence.

The powers-of-two example of the previous section does fit this pattern, so we could rewrite it like this:

```
int x;

for(x = 2; x < 1000; x = x * 2)
        printf("%d\n", x);
```

There is no earth-shaking or fundamental difference between the `while` and `for` loops. In fact, given the general `for` loop

```
for(expr<sub>1</sub>; expr<sub>2</sub>; expr<sub>3</sub>)
        statement
```

you could usually rewrite it as a `while` loop, moving the initialize and increment expressions to statements before and within the loop:

```
expr<sub>1</sub> ;
while(expr<sub>2</sub>)
        {
        statement
        expr<sub>3</sub> ;
        }
```

Similarly, given the general `while` loop

```
while(expr)
        statement
```

you could rewrite it as a `for` loop:

```
for(; expr; )
        statement
```

Another contrast between the `for` and `while` loops is that although the test expression (*expr₂*) is optional in a `for` loop, it is required in a `while` loop. If you leave out the controlling expression of a `while` loop, the compiler will complain about a syntax error. (To write a deliberately infinite `while` loop, you have to supply an expression which is always nonzero. The most obvious one would simply be `while(1)` .)

If it's possible to rewrite a `for` loop as a `while` loop and vice versa, why do they both exist? Which one should you choose? In general, when you choose a `for` loop, its three expressions should all manipulate the same variable or data structure, using the initialize, test, increment pattern. If they don't manipulate the same variable or don't follow that pattern, wedging them into a `for` loop buys nothing 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, and if the `for` loop you're looking at doesn't end up matching that pattern, you've been momentarily misled.)

3.6 `break` and `continue`

[This section corresponds to K&R Sec. 3.7]

Sometimes, due to an exceptional condition, you need to jump out of a loop early, that is, before the main controlling expression of the loop causes it to terminate normally. Other times, in an elaborate loop, you may want to jump back to the top of the loop (to test the controlling expression again, and perhaps begin a new trip through the loop) without playing out all the steps of the current loop. The `break` and `continue` statements allow you to do these two things. (They are, in fact, essentially restricted forms of `goto`.)

To put everything we've seen in this chapter together, as well as demonstrate the use of the `break` statement, here is a program for printing prime numbers between 1 and 100:

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

main()
{
int i, j;

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

for(i = 3; i <= 100; i = i + 1)
        {
        for(j = 2; j < i; j = j + 1)
                {
                if(i % j == 0)
                        break;
                if(j > sqrt(i))
                        {
                        printf("%d\n", i);
                        break;
```

```
                        }
                }
        }

return 0;
}
```

The outer loop steps the variable `i` through the numbers from 3 to 100; the code tests to see if each number has any divisors other than 1 and itself. The trial divisor `j` loops from 2 up to `i`. `j` is a divisor of `i` if the remainder of `i` divided by `j` is 0, so the code uses C's ``remainder'' or ``modulus'' operator `%` to make this test. (Remember that `i % j` gives the remainder when `i` is divided by `j`.)

If the program finds a divisor, it uses `break` to break out of the inner loop, without printing anything. But if it notices that `j` has risen higher than the square root of `i`, without its having found any divisors, then `i` must not have any divisors, so `i` is prime, and its value is printed. (Once we've determined that `i` is prime by noticing that `j > sqrt(i)`, there's no need to try the other trial divisors, so we use a second `break` statement to break out of the loop in that case, too.)

The simple algorithm and implementation we used here (like many simple prime number algorithms) does not work for 2, the only even prime number, so the program ``cheats'' and prints out 2 no matter what, before going on to test the numbers from 3 to 100.

Many improvements to this simple program are of course possible; you might experiment with it. (Did you notice that the ``test'' expression of the inner loop `for(j = 2; j < i; j = j + 1)` is in a sense unnecessary, because the loop always terminates early due to one of the two `break` statements?)