Chapter 5: Statement Details

Statements are basic building blocks of a C program, very much as sentences are basic building blocks of English writing. C provides programmers with a rich set of operations, allowing them to easily construct complex and powerful statements. This power must be used judiciously, however. It is far too easy to create complex, unreadable, sometimes indecipherable and unreliable C code. The rules discussed this chapter will help you create simple, readable, reliable code.

Doing Nothing

One of the most overlooked statements is the “do nothing”, or null, statement. The syntax for this statement is extremely simple:

Because it's so tiny, the null statement can easily be missed. For example, the code:

```
for (i = 0; string[i] != 'x'; ++i);
```

actually contains two statements: a for statement and a null statement. Most people must look closely at this code to find the null statement.

That's bad style. The structure of a well-constructed program is obvious; it does not require close inspection. We need to do something to the null statement to make it obvious, and comment lines easily provide the answer:

```
/* Do nothing */;
```

Now the code fragment looks like this:

```
for (i = 0; string[i] != 'x'; ++i)
    /* Do nothing */;
```

With this construction, it is obvious that there are two statements.

Rule 5-1:

Always put a comment in the null statement, even if it is only

```
/* Do Nothing */;
```

Arithmetic Statements

C provides the programmer with a rich set of operators. There are 15 precedence rules in C (&& comes before `||`, etc.). For example, in this statement:

```
result = 1 << 5 + 1;
```

does the compiler perform the << or the + first? In other words, is the statement equivalent to this:

```
result = (1 << 5) + 1;
```

or to this:

```
result = 1 << (5 + 1);
```

it turns out that + comes before <<, so the second version is correct.

The problem is that all these rules are difficult to remember. I've been programming in C for over 10 years and I can't remember all the rules. I even had to look up the answer to this problem.

Even if you remember all the rules, have pity on the programmer who will be reading your code someday and who may not have your memory. I've devised a practical subset that's simple to memorize:

Rule 5-2:

In C expressions, you can assume that `*`*,* `/`*, and % come before* `+` *and* `-`*. Put parentheses around everything else.*

Following this rule, the problem expression becomes:

```
result = 1 << (5 + 1);
```

and in this statement, the order of operations is obvious.

Function Headings

All C code is contained in functions. The function heading defines its return type and the parameters.

Example:

```
float average(float total, int n_items)
```

There are actually two styles of function declarations. Throughout this book I've been using the newer ANSI-C style. Older compilers allow only the traditional K&R style:

```
float average(total, n_items)
float total;
int n_items;
```

The ANSI-C style is preferred because it is more modem, less error prone, and compatible with C++. Reserve the use of the K&R style for old compilers that don't allow ANSI style declarations.

Rule 5-3:

Use ANSI style function declarations whenever possible.

K&R style parameters

Some of the older C compilers force you to use K&R style parameters. This format does not allow types in the function declaration. The types immediately follow the function declaration.

```
int total(values, n_values)
int values[];
int n_values;
```

Strictly speaking, the declaration `int n_values` is redundant. The type of all parameters defaults to **int**. So you could have written this function as:

```
/* Poor style */
int total(values, n_values)
int values[];
```

The problem with this is the problem that occurs with all defaults: you can't tell the programmer's intent. Did the programmer intend to make `n_value` integer or leave out a declaration? When you explicitly declare all parameters, you eliminate an doubt.

Rule 5-4:

When using K&R parameters, declare a type for every parameter.

The type declarations for the parameters may be specified in any order. For example:

```
/* Poor style */
int total(values, n_values)
int n_values;
int values[];
```

The problem here is that you are fighting with the natural one-to-one correspondence between parameters and their type declarations. It's a lot easier find things if they are put in order.

Rule 5-5:

When using K&R parameters, put the type declarations for the parameters in the same order as the occur in the function header.

Return type

In C, defining the function type is optional. If the type is not specified defaults to **int.** For example:

```
int do_it(void);
```

and

```
do_it(void);
```

are equivalent in C. However, they are not the same to the programmer because I second form is ambiguous. There can be two reasons for not specifying a function type: the return type really should be **int,** or the programmer forgot to define correct return type.

The explicit declaration of an **int** function type tells the reader of the program, "Yes, this function really does return an **int.**"

Rule 5-6:

Always declare a function type

It is possible to have an integer function that doesn't return anything. For example

```
do_more(void)
{
    /*......    */
    return;
}
```

Code like this can be found in older programs that pre-date the invention **void** type. Good style means that you tell the reader as much as possible

Rule 5-7:

Always declare functions that do not return a value as **void**.

Number of parameters

In theory, functions can have any number of parameters. In practice, not quite true. That's because while the compilers may be able to han function with 100 parameters, programmers cannot.

Long parameter lists remind me of a passage from the UNIX mag tape manual page: “Devices /dev/rmt0, /dev/rmt4, /dev/rmt8, /dev/nrmt0, /dev /dev/nrmt8 are the rewinding low density, rewinding medium density, rewinding high density, non-rewinding low density, non-rewinding medium density, non-rewinding high density devices, respectively.” The problem with long list that you tend to lose track of things.

What's the device name of the “non-rewinding medium density” tape Try to figure it out without counting on your fingers**.**

Suppose you want to define a function to draw a line. You could write it like this:

```
/*******************************************************
 * DrawLine - Draw a line                              *
 *      draws a line from current point (set by        *
 *      a previous call to GotoPoint to x,y)           *
 *                                                     *
 * Parameters                                          *
 *      x - Point we draw to (x co-ordinate)           *
 *      y - Point we draw to (y co-ordinate)           *
 *      style - line style (DASHED, SOLID)             *
 *      brush - the type of brush to draw with         *
 *      pattern - pattern for filling in the line      *
 *              (STRIPPED, CROSS_HATCH, SOLID)         *
 *      end_style - how to draw the ends               *
 *              (CAPPED, FLUSH, ...)                   *
 *      front -- true if the line is to be drawn over  *
 *              everything (false, draw in back)       *
 *******************************************************/
void DrawLine(int x, int y,
    style_type style, color_type color,
    brush_type brush, pattern_type pattern,
    end_style_type end_style, boolean front);
```

This is a disaster waiting to happen. All sorts of problems can easily occur. You forget a parameter, get the parameters out of order, or generally confuse things. Allowing no more than five parameters to a function can help alleviate this.

Rule 5-8:

Allow no more that five parameters to a function.

But now what do you do with the function *DrawLine?* The solution is to take this herd of parameters and stuff them into a structure:

```
struct draw_style {
     /* style for drawing (DASHED, SOLID) */
    style_type style,

    /* color (BLACK, WHITE, BLUE,...) */
    color_type color,

    /* the type of brush to draw with */
    brush_type brush,

    pattern_type pattern, /* pattern (STRIPPED, SOLID) */

    /* line ends (CAPPED, FLUSH, ...) */
    end_style_type end_style,

    boolean front               /* Front or back */
);

/*****************************************************
 * DrawLine - Draw a line                            *
 *      draws a line from current point (set by      *
 *      a previous call to GotoPoint to x,y)         *
 *                                                   *
 * Parameters                                        *
 *      x - Point we draw to (x co-ordinate)         *
 *      y - Point we draw to (y co-ordinate)         *
 *      how - structure describing how to draw the line *
 *****************************************************/
void DrawLine(int x, int y, struct draw_style *how);
```

There are tremendous advantages to using structures for complex parameter, passing. First, it's easier to remember a structure's field name than it is a parameter's position number. (Without looking, can you tell if `pattern` is the fifth or sixth parameter?)

Another advantage is that you need set only the fields that apply. You can ignore the others. If you used a long parameter list, you must have something for, each parameter.

Structures also make it easy to specify default value. For example, if you define the following defaults:

Field	Default Value	Integer value of default
`style`	SOLID	0
`color`	BLACK	0

Field	Default Value	Integer value of default
brush	SMALL_ROUND	0
pattern	SOLID	0
end_style	CAPPED	0
front	FALSE	0

then the statement:

```
memset(&current_style, '\O', sizeof(struct style));
```

initializes the entire structure with default values. (Note: For this to work, the default must all be zero.)

Passing parameters in globals

Another way of passing parameters to and from a function is to not use parameters at all. Instead, values are passed through global variables. For example, consider the following function:

```
/*******************************************************
 * GetTocken - read the next token                     *
 *                                                     *
 * Globals used                                        *
 *      in_file -- file to get token from              *
 *      token -- the token we just got                 *
 *      error -- 0 = no error                          *
 *               non-zero = error code                 *
 *******************************************************/
```

There are many problems with this type of parameter passing. First, obscures the interface between the function and the outside world. What's type of token? You can't tell. Also, suppose you want to handle multiple files Then your main code must keep reassigning in-file so that it points to what file you are using. For example:

```
in_file = main_file
GetToken();
main_token = token;

in_file = include_file;
GetToken();
include_token = token;
```

It's much easier to write:

```
main_token = GetToken(in_file, &error);
include_token = GetToken(include_file, &error);
```

A good function interface provides the user with a single, small interface to the function. When parameters are passed as globals, the function interface in divided into two (or more) parts. The global declarations are in on part of header file and the function declaration in another.

Also, using these types of functions is difficult, requiring multiple statements. It's extremely easy to get things wrong. When you pass parameters as parameters, C checks both the number and type of the parameters. These checks are a big help in improving reliability.

Rule 5-9:

Avoid using global variables where function parameters will do.

XView style parameter passing

XView programming uses a nearly unique parameter passing style. (It shares this style with the Suntools system from Sun.) For example, the function *XvSet* is defined as:

```
XvSet(handle,
    item, [value], [value], [value], ....
    item, [value], [value], [value], ....
    NULL);
```

The unique feature of this calling sequence is the use of variable parameter lists. “Item” is a XView attribute. The number of parameters that follow arc defined by the attribute.

For example, a typical XvSet function looks like this:

```
XvSet(popup,
    PANEL_CHOICE_NROWS, 5,
    PANEL_CHOICE_STRINGS,
        "Start",
        "Run",
        "Abort",
        "Pause",
        "Continue",
        NULL,
    NULL);
```

The parameter `PANEL_CHOICE_NROWS` is followed by a single number, and `PANEL_CHOICE_STRINGS` is followed by a list of names. This list is terminated by a `NULL`. The entire parameter list is terminated by a `NULL`.

Programmers at Sun went to a lot of work devising this parameter-passing mechanism. It's too bad they came up with something so poor. There are many problems with this style.

First, since the parameter list is variable length and the types of the variables are not fixed, it is impossible to check the type and number of parameters. This defeats any type-checking built into C or *lint*.

The second problem occurs when a terminator is accidentally omitted. In the example, the list of names is terminated by a `NULL`. What would happen if you forgot it?

```
XvSet(popup,
    PANEL_CHOICE_NROWS, 5,
    PANEL_CHOICE_STRINGS,
        "Start",
        "Run",
        "Abort",
        "Pause",
        "Continue",
    NULL);
```

The *XvSet* function reads the parameters, finds the NULL after the strings, and assumes that it ends the strings. *XvSet* thinks more parameters follow, but there are none, so it reads random memory and goes crazy.

It would be much easier to turn the very general *XvSet* into a series of simpler, specialized functions:

```
XvSetPanelNrows(popup, 5);
XvSetPanelChoiceStrings(popup, string_list);
```

This style of parameter passing lets C's type checking do its job and avoids many potential problems.

Rule 5-10:

Avoid variable length parameter lists. They are difficult to program and can easily cause trouble.

The if Statement

The **if/else** statement presents the programmer with some special problems. The first is ambiguity. There is a small “hole” in the syntax, as illustrated in the following example:

```
if (a)
    if (b)
        printf("First\n");
  else /* Indentation is off */
      printf("Second\n");
```

The question is, which if does the else go with?

A) It goes with `if (a)`

B) It goes with `if (b)`

C) The answer doesn't matter if I don't write code like this.

Give yourself ten points if you answered C. If you don't write silly code, you won't have to answer silly questions. (For the purist, the **else** goes with the nearest **if.** Answer B.)

By using braces, you can avoid the problem of ambiguity as well as make your code clearer.

```
if (a)  {
    if (b)
        printf("First\n");
    else
        printf("Second\n");
}
```

Rule 5-11:

*When an **if** affects more than one line, enclose the target in braces.*

if/else chains

Frequently programmers need to implement a decision tree. This usually results in a chain of **if/else** statements. Using our current indentation rules, this results in code that looks like this:

```
if (code == ALPHA) {
    do_alpha();
} else {
    if (code == BETA) {
        do_beta();
    }else {
        if (code ==GAMMA) {
            do_gamma();
        } else {
            do_error();
}
```

This format adds needless complexity to your program, but how do you simplify it? The solution is to treat the word pair **else if** as a single keyword.

Rewriting the code using this rule results in this:

```
if (code == ALPHA) {
    do_alpha();
} else if (code == BETA) {
    do_beta();
}else if (code ==GAMMA) {
   do_gamma();
} else
    do_error();
```

This is at once simpler and easier to understand.

Rule 5-12:

In an **if** *chain, treat the words* **else** *if as one keyword.*

if and the comma operator

The comma operator is used to combine statements. For example, the statements:

```
x = 1;
y = 2;
```

are treated as a single statement when written as:

```
x = 1, y = 1;
```

With simple statements, the comma operator is not very useful. However it can be used in conjunction with **if** to provide the programmer with a unique shorthand.

```
if (flag)
    x =1, y = 1;
```

This example is syntactically equivalent to:

```
if (flag) {
     x = 1;
     y = 1;
}
```

The problem with the comma operator is that when you use it you break the rule of one statement per line, which obscures the structure of the program.

Rule 5-13:

Never use the comma operator when you can use braces instead.

The while Statement

Sometimes you want to have a loop go on forever (or until you hit a **break).** There are two common ways of specifying an infinite loop.

```
while (1)
```

and

```
for (;;)
```

The first (`while`) is preferred because it is more obvious and causes I confusion than `for(;;)`. The `while` statement gives the programmer a simple looping mechanism, and because it is so simple there are not a lot of style rule go with it.

Rule 5-14:

When looping forever, use `while (1)` *instead of* `for(;;)`.

Some programmers are tempted to put assignment statements inside a **while** conditional, like this:

```
/* Poor practice */
while ((ch = getc()) != EOF) {
    / *....  * /
```

This is breaks the no side effects rule. It is compact, but it obscures some of the logic in the code. It is more effectively written like this:

```
/* Good practice */
while (1) {
    ch = getc();
    if (ch == EOF)
        break;
    /* ... */
```

This way, you can see the statements explicitly instead of having to extract the from some cryptic logic.

The do/while Statement

The **do/while** statement is rarely seen in practical C programs. That s because it's redundant- there's nothing that you can do with a **do/while** that can't be done with **while** and **break**.

Because it is so rare, many programmers are surprised when they see it. Some don't even know what to do with it. For these reasons, it is better to simply not use it.

Rule 5-15:

Avoid using **do/while.** *Use* **while** *and* **break** *instead.*

The for Statement

There are two common problems with use of the **for** statement. They can have too little content, or too much.

Missing parts of for loops

The **for** statement is actually three statements in one. Sometimes all three parts are not needed, so one or more is left blank:

```
/* Poor practice */
set_start(&start);

for (;start < end; start++) {
    / *... * /
```

There is a slight problem with this code. We've broken one of rules and did “nothing” silently. The initialization section of the **for** is the empty statement “;”. But with just a “;” to guide you, how can you tell that the programmer didn't accidentally omit the initialization statement? In fact you can't. But including the comment: `/* Start already set */` tells you the omission was intentional.

```
/* Better practice */
set_start(&start);

for (/* Start already set */;start < end; start++) {
    / *... * /
```

You also need a comment when there is no conditional clause. For example:

```
for (start = 0; /* break below */; start++) {
```

Overstuffed for loops

So far we've discussed what happens when you put too little information for loop. It's also possible to put in too much. As mentioned before, the c operator can be used to combine statements in an **if.** This also works for statement. For example, the statement:

```
for (two = 2, three = 3, two < 50; two +=2, three += 3)
```

is perfectly legal. This statement causes the variable two to increment by 2 and the variable *three* to increment by 3, all in one loop.

The notation is complex and confusing. However spreading out the loop clarifies the logic:

```
two = 2;
three = 3;
while (two < 50) {

    /*.... */
    two += 2;
    three += 3;
}
```

You'll note that we have also changed the **for** loop to a **while.** It could be I as a **for,** but here the **while** shows the structure of the code more clearly.

Stringing together two statements using the comma operator is sometimes useful in a **for** loop, but such cases are rare.

Rule 5-16:

Use the comma operator inside a for statement only to put together two statements. Never use it to combine three statements.

The printf Statement

The *printf* function and its cousins are used for outputting the data. The function can be used to print one or more lines. For example:

```
printf("Beginning = %d\ncurrent = %d\n End=%d\n",
    beginning, current, end);
```

Although compact, this obscures what is being output. You are writing three lines, so why not use three *printf* statements?

```
printf("Beginning = %d\n", beginning);
printf(“Current = %d\n", current);
printf("End = %d\n", end);
```

Using this style, You can easily see the structure of the output.

Rule 5-17:

Use one printf per line of output.

Some people might argue that it take more time to do things this way since there are three function calls instead of one. The *printf* function is relatively slow. The amount of overhead in a function call takes 1/1000 of the time it takes to execute even a simple *printf*, so the overhead of the two extra calls in negligible.

Another problem occurs with the use of the *printf*, *puts*, and *putc* function, If you always use *printf*, you have consistency. If you use a mixture of *printf*, and *putc*, then you increase efficiency at the expense of consistency.

For example:

```
/* Consistent */
printf(“Starting phase II\n");
printf("Size = %d\n”, phase_size);
printf(“Phase type %c\n", phase_type);

/* Efficient */
puts("Starting phase II\n");
printf("Size =%d\n”, phase_size);
puts(“Phase type ");
putc(phase_type);
putc('\n');
```

In most cases, the increase in efficiency is very small. You probably won't notice any speedup in your program unless the code is executed thousands of rich in an inner loop. The difference in consistency is extremely noticeable, however. In most cases, readability and consistency considerations outweigh any efficiency considerations.

Rule 5-18:

Unless extreme efficiency is warranted, use printf instead of puts and putc.

goto and Labels

Good programmers avoid the **goto** statement because it breaks the structure of the program. But every once in a while, even the best programmer needs to use a **goto**.

The **goto** label doesn't fit anywhere in the indentation rules. It's not part of the regular structure, so in order to give it a home, make it stand out, and generally get out of the way, put it up against the left margin.

```
        for (x = 0; x < 10; ++x) {
            for (y = 0; y < 10; ++y) {
                if (data[x][y] == look_for) {
                    goto found_it;
                }
            }
        }
found_it:
```

Rule 5-19:

Start **goto** *labels in the first column.*

The switch Statement

The **switch** statement is the most complex statement in C. It allows the programmer to perform a complex branching operation with a single statement, but sometimes it can be confusing.

Good programming style can make the **switch** statement clearer and more reliable. Consider the following statement:

```
/* Poor practice */
switch (state) {
    case BEGIN_STATE:
         printf("Beginning\n");
    case PROC_STATE:
         printf("Processing\n");
         break;
     case FINISH_STATE:
         printf("Finishing\n");
}
```

At the end of the `BEGIN_STATE` case, there is no **break,** so the program falls through. Thus, when `state = BEGIN_STATE`, you'll get the messages:

```
Beginning
Processing
```

Is this intentional or accidental? From this code, there is no way to know. If the programmer intends a fall through, he or she needs to tell people about it. The comment "`/* Fall Through */`" would help immensely, yielding:

```
/* Not so poor */
switch (state) {
    case BEGIN_STATE:
        printf("Beginning\n");
        /* Fall through */
    case PROC_STATE:
        printf("Processing\n");
        break;
     case FINISH_STATE:
         printf("Finishing\n");
}
```

Rule 5-20:

End every case in a **switch** *with a* **break** *or the comment* `/* Fall Through */`

Now consider the last **case,** `FINISH_STATE`. It doesn't need a **break** because it's at the end of the **switch.** However, you may want to consider putting in a **break** to avoid future problems. For example, you may want to add another **case,** perhaps one for `ABORT_STATE`. This would give you this:

```
/* Surprise! */
switch (state) {
    case BEGIN_STATE:
        printf("Beginning\n");
        /* Fall through */
    case PROC_STATE:
        printf("Processing\n");
        break;
    case FINISH_STATE:
        printf("Finishing\n");
    case FINISH_STATE:
        printf(“Aborting\n”);
}
```

You may have noticed the error: You need a **break** after the `FINISH_STATE` case. If you get in the habit of always putting in a **break** at the end of a **switch** statement, then you don't have to worry about having to put it in during cod modifications.

Good habits are better than a good memory any day.

Rule 5-21:

Always put a break at the end of the last case in a switch statement.

The **switch** statement now looks like this:

```
/* Almost there */
switch (state) {
    case BEGIN_STATE:
        printf("Beginning\n");
        /* Fall through */
    case PROC_STATE:
        printf("Processing\n");
        break;
    case FINISH_STATE:
        printf("Finishing\n");
        break;
    case FINISH_STATE:
        printf(“Aborting\n”);
        break;
}
```

But what happens when state is `STATE_IDLE`? There are several possible answers:

1. It is ignored
2. This is a ran-time error

3. When you execute this code, state will never contain `STATE_IDLE`, so you don't have to worry about what will happen.

As far as C is conceded, when state is `STATE_IDLE`, then the **switch** is ignored. But that's not good enough. Did the programmer intentionally ignore out-of-range cases, or was it accidental? Again, you don't know. If the programmer intended bad states to be ignored, he or she could have written:

```
default:
    /* Do nothing */
    break;
```

This makes explicit what was implied.

Rule 5-22:

> *Always include a* **default** *case in every* **switch,** *even if it consists of nothing but a null statement.*

But suppose the programmer says that state can never be anything other than the three cases. Do you need a default for something that will never happen?

The answer is a resounding yes! Any experienced programmer will tell you that things that can never happen do happen. A good defensive programming technique is to include code for the impossible:

```
default:
    fprintf(stderr,
       "Internal error. Impossible state %d\n", state);
    exit(1);
```

So the full-blown switch statement has evolved into this:

```
/* Good style */
switch (state) {
    case BEGIN_STATE:
        printf("Beginning\n");
        /* Fall through */
    case PROC_STATE:
        printf("Processing\n");
        break;
    case FINISH_STATE:
        printf("Finishing\n");
        break;
    case FINISH_STATE:
        printf(“Aborting\n”);
        break;
    default:
        fprintf(stderr,
           “Internal error. Impossible state %d\n”, state);
        exit(1);
}
```

Your work on this statement can be summarized with this rule: Always put everything in the **switch** and make it all obvious.

Debug Printing

In spite of all the interactive debuggers, there are still times a programmer needs to use a debugging *printf*. The problem is how to separate the debugging output from the real stuff. One trick is to begin all debug printouts with “##”:

```
printf("## state = %d\n", state);
```

This not only makes it easy to identify the debug statements in the log, it also makes it easy to remove them after the program is debugged. All you have to do is search for each line containing “##” and delete it.

Shut up Statements

Always compile your programs will all possible warning enabled. If you are running under UNIX, run your program through the program lint. You want the compiler to find as many potential problems in your code as possible, so you don't have to do it in the debugging stage.

Sometimes you get warning about things you know about. For example, you might define a variable copyright and never use it. Sometimes the compiler or *lint* will allow you to turn off a warning for a single statement or variable. But sometimes it won't.

For example, there is no way to turn off the “Variable defined but not used” message in the Borland C compiler for a single variable. It's either the whole program or nothing.

The solution to this problem is a set of statements designed solely to turn off warning messages. For example:

```
static char *copyright = "Copyright 1992 SDO";
/ *.... * /
main()


(void)copyright;      /* Avoid warning */
```

In this case the statement *(void)copyright* "uses" the variable *copyright. The* statement itself does nothing. In fact, the compiler knows it does nothing and generates no code for it. The only reason to put this statement in the code is to trick the compiler into thinking that the variable is being used so it won't issue a warning. Note that a comment is supplied to explain what was done. Otherwise someone looking at this code later might think we're crazy.

The program lint gets upset when you don't use the value returned by a function. The cast (void) can be used to tell *lint* "I know that this function returns value, but I don't care."

```
i = get_into;     /* No warning */
get_into;         /* Warning */
(void)get_into;   /* No warning */
```