Chapter 4: Statement Formatting

Organization is the key to a well-written program. Good programming style helps present the detail and logic of your program in a clear and easy-to-un stand manner.

Programming style and aesthetics are related. A well-written program pleasing to look at, read, and understand. Your goal in formatting a program is to make it look neat, well-organized, and beautiful.

Formatting the Body of the Program

The sentence is a basic unit of writing. A sentence ends with a terminator question mark, exclamation point, or period. In C, the basic coding unit is statement. C statements do not have terminators, like sentences; rather. they separated from each other by semicolons (`;`).

Well laid-out programs allow the programmer to quickly and easily pick the statement within the program. Running the code together, as shown in following example, hurts readability and clarity:

```
/* Poor programming practice */
biggest=-l;first=0;count=57;init_key_words();
if(debug)open_log_files();table_size=parse_size+lex_size;
```

How many statements are in this program fragment? It's hard to tell. programmer has tried to compact the program by putting as much on each line possible. It's much like the old limerick:

There was a young man from Iran

Whose verses just would not quite scan.

When someone asked why,

He gave this reply:

I like to put as many words on the last line as I possibly can.

It's easier to pick out the statements when there is only one statement per line

```
/* Better programming practice (still needs work) */
biggest=-l;
first=0;
count=57;
init_key_words();
if(debug)
    open_log_files();
table_size=parse_size+lex_size;
```

Rule 4-1:

Write one statement per line.

There are still some problems with this fragment. True, it is much easier find the statements, but they are still hard to read. The problem is that our eyes are trained to treat a set of letters and symbols as one word. Writing sentencewithnospacesmakesitveryhardtoread. Similarly writing a C statement

Rule 4-2:

Put spaces before and after each arithmetic operator, just like you put spaces between words when you write.

```
/* Better programming practice (still needs work) */
biggest = -l;
first = 0;
count = 57;
init_key_words();
if(debug)
    open_log_files();
table_size = parse_size + lex_size;
```

Adding spaces not only improves readability, it also helps to eliminate errors. Consider the statement:

```
*average = *total / *count;      /* Compute the average */
```

Written without spaces this becomes:

```
*average=*total/*count;          /* Compute the average */
```

Looks like the same statement, but it's not. Can you spot the problem?

The operators slash (/) and star (*) take on a new meaning when they are put together with no space. The operator /* is the beginning of a comment. So the result of the compression is:

```
*average=*total    /* count; /* Compute the average */
```

If you have a good C compiler, you will get a warning about nested comments. If you have a typical compiler, you'll get nothing.

Now back to our program fragment. Spaces makes the individual statements easier to read, but the entire fragment is still something of a gray blob. We can do better.

This book is broken up into paragraphs. The paragraph markers separate one set of sentences from another. In C, you can use blank lines to separate code into paragraphs:

```
/* Better programming practice (still needs work) */
biggest = -l;
first = 0;
count = 57;
init_key_words();
if (debug)
    open_log_files();
table_size = parse_size + lex_size;
```

The result is a section of code that uses spaces to separate the coding element into pleasantly arranged groups.

Simplifying complex statements

Sometimes a statement such as an assignment statement grows so long complex that it can't fit on one line. In such cases, consider turning a complex statement into several smaller, simpler statements.

For example, this is syntactically correct, but

```
/* This is a big mess */
gain = (old_value - new_value) /
       (total_old - total_new) * 100.0;
```

It can be rewritten as three smaller statements:

```
/* Good practice */
delta_value = (old_value - new_value);
delta_total = (total_old - total_new);
gain = delta_value / delta_total * 100.0;
```

Rule 4-3:

Change a long, complex statement into several smaller, simpler statements.

Splitting long statements

An alternative to turning one statement into two is to split long statements into multiple lines.

Splitting is an art. The idea is to split the line in a way that does confusion. There is a rule: One statement per line. A two-line statement that rule, so always indent the second line to indicate that it is a continuation.

Rule 4-4:

In a statement that consists of two or more lines, every line except the first must be indented an extra level to indicate that it is a continuation of the first line.

For example:

```
net_profit = gross_profit - overhead -
             cost_of_goods - payroll;
```

This example brings up another question: Do you put operators at the end of the line, as in the previous example, or at the beginning of the next line?

```
net_profit = gross_profit - overhead
           - cost_of_goods - payroll;
```

Actually, either method is acceptable as long as it is used consistently. That means that I don't get to dodge the issue if there is any reasonable basis for choosing one method over the other.

There is a basis: majority rule. Most programmers prefer to put the operators at the end of the line. So let's go with the majority preference and make it the rule:

Rule 4-5:

When writing multi-line statements, put the arithmetic and logical operators at the end of each line.

Splitting and parentheses.

Complex statements in C can contain several levels of parentheses. The following example shows a complex statement that contains many parentheses. The comment below it indicates the nesting level.

```
result = (((x1 + 1) * (x1 + 1)) - ((y1 + 1) * (y1 + 1)));
/* nest  12333333322223333333321112333333322223333333321 */
```

The best place to break the line is where the nesting level is lowest; in this case at the `-` operator in the middle:

```
result =(((x1 + 1) * (x1 + 1)) -
         ((y1 + 1) * (y1 + 1)));
```

Rule 4-6:

When breaking up a line, the preferred split point is where the parenthetic nesting is lowest.

The second line of the example is carefully indented so that the parenthesis line up. Why not align it with the first parenthesis?

```
/* Don't program like this */
result =(((x1 + 1) * (x1 + 1)) -
        ((y1 + 1) * (y1 + 1)));
```

Notice that the lines seem to be a little off. That's because the first line's Level I parenthesis is in the same column as the second line's Level 2 parenthesis.

```
/* Don't Program like this */
result =(1(2(3x1 + 1)3 * (3x1 + 1)3)2 -
        (2(3y1 + 1)3 * (3y1 + 1)3)2)1;
```

Properly aligned (Level 2 to Level 2), the same statement looks like this:

```
result =(1(2(3x1 + 1)3 * (3x1 + 1)3)2 -
          (2(3y1 + 1)3 * (3y1 + 1)3)2)1;
```

Rule 4-7:

Align like level parentheses vertically.

Here's a more complex example:

```
flag = (result == OK) ||
       ((result == WARNING) &&
        ((code == WARN_SHORT) ||
         (code == WARN_EOF))
       );
```

The indentation in this example gives the programmer several clues about the statement's logic. First, there are two major clauses:

```
flag = (result == OK) ||
```

and

```
       ((result == WARNING) &&
        ((code == WARN_SHORT) ||
         (code == WARN_EOF))
       );
```

They are indented at the same level. The second clause goes on for four line This is obvious because its beginning and ending parenthesis have the sam indent. The two `(code ==` lines carry equal weight and are at the same level. Their beginning parenthesis are aligned in the same column.

As you can see, proper splitting and indentation of a multi-line statement c

Splitting a for statement.

A **for** statement is unique, since it is three statement

```
for (<initialization>; <condition>; <increment>)
```

The *<initialization>, <condition>, and <increment>* are three complete C statements. If these statements have any complexity at all, the entire **for** statement is likely to overflow the line. Whenever a **for** grows too long for one line, split it first at the component statement boundaries.

For example, this line:

```
for (index = start; data[index] != 0; ++index)
```

splits like this:

```
for (index = start;
     data[index] != 0;
     ++index)
```

Note that we've aligned the beginnings of the three substatements.

Rule 4-8:

Split long **for** *statements along statement boundaries.*

In the previous example, we turned a one-line **for** statement into three. But if the statement is short enough, can you limit it to two lines?

```
/* Poor practice */
for (index = start; data[index] != 0;
     ++index)
```

The answer is no. The problem is that this split is ambiguous. We could just as easily have written this:

```
/* Poor practice */
for (index = start;
     data[index] != 0; ++index)
```

Consistency is part of good style. If you do things consistently, you set up expectations, which is another way of saying you remove one detail that the reader of the program has to figure out. Don't split a **for** statement one way one time and another the next. You can consistently split a **for** statement into three lines the same way every time, but there is no preferred two-line split. Two-line splits introduce inconsistency. Avoid them.

Rule 4-9:

Always split a **for** *statement into three lines.*

Splitting a switch statement.

The **switch** statement is the most complex statement in the C language. The rule for splitting it is very simple: Don't. If the index expression for a **switch** statement grows too big for one line, split it into two different statements: an assignment and a **switch.**

For example:

```
/* Bad practice */
switch (state_list[cur_state].next_state +
        goto_list[last_last] +
        special_overrides) {
```

should be turned into:

```
/* Good practice */
switch_index = (state_list[cur_state.next_state +
                goto_list[last_last] +
                special_overrides);
switch (switch_index) {
```

Rule 4-10:

Write **switch** *statements on a single line.*

Conditional operators (? :).

When splitting an expression containing a conditional operation (**? :**), try to put the entire conditional clause on a line by itself

```
/* Good practice (preferred) */
result = past-due +
    (total-owed > 0) ? total-owed : 0;
```

Rule 4-11:

Keep conditionals on a single line if possible.

If the conditional clause itself is too long for one line, it can be split into three lines. The format is this:

```
(condition) ?
    (true-value) :
    (false-value)
```

Each line contains one component of the expression. Since the true-value and false-value are sub-sections of the conditional, their lines are indented.

Rule 4-12:

*When splitting a conditional clause (***? :***), write it on three lines: the condition line, the true-value line, and the false-value line. Indent the second and third line an extra level.*

Side effects

When writing a children's story, you must keep the sentence structure simple and avoid compound sentences. Well, a computer is not a child; it doesn't have that much intelligence. But in C coding, you should do anything you can to simply the program. That means avoiding side effects.

A side effect is an operation that is performed in addition the main operation of a statement. For example, the statement:

```
current = count[index++]
```

assigns current a value and increments index. Look out. Any time you start using “and” to describe what a statement does, you're in trouble. The same statement could just as easily have been written this way:

```
current = count[index]
index++;
```

This way, there are no side effects.

C allows very free use of the ++ and -- operators within other statements. Taking advantage of this freedom can create all sorts of problems. Consider the statement:

```
i = 0
out[i++] = in[i++];
```

In fact, consider it a test. Exactly what does this program do?

A) Evaluate `out[i]` as `out[0]`, increment `i` (`i` is now 1), evaluate `in[i]` as `in[1]`, increment `i` (`i` is now 2), do the assignment (`out[0] = in[1]`).

B) Evaluate `in[i]` as `in[0]`, increment `i` (`i` is now 1), evaluate `out[i]` as `out[1]`, increment `i` (`i` is now 2), do the assignment (`out[1] = in[0]`).

C) Evaluate `in[i]` as `in[0]`, evaluate `out[i]` as `out[0]`, increment `i` (`i` is now 1), increment `i` (`i` is now 2), do the assignment (`out[0] = in[0]`).

D) The code is compiler dependent, so the compiler carefully computes the best possible answer and then does something else.

E) If you don't write code like this, you won't have to worry about questions like this.

This code is ambiguous, and the actual code generated can change from compiler to compiler. Sometimes the same compiler will generate different code depending on the state of the optimize switch.

The answer, of course, is E.

Ambiguous code is not the only problem that can occur when ++ and -- are used within other statements. The statements

```
i = 2;
s = square(i++);
```

look innocent enough. But square is a macro designed to square a number:

```
#define square(x)         ((x) * (x))
```

If you expanding the macro, you get this:

```
i = 2;
s = ((i++) * (i++));
```

Suddenly you see that i is not incriminated once as expected, but twice. And s an be assigned the wrong value. Again, this statement is ambiguous.

You can avoid all these problems by writing the ++ on a separate line:

```
i = 2;
s = square(i);
i++;
```

True, putting the ++ inside another statement does make for more compact code, but the value of compactness in C source code is minimal. You're striving for readability and reliability. The one-effect-per-statement rule improves both, especially reliability.

It also simplifies the program. What are the values of i and j after the following code is executed?

```
i = 0;
j = 0;
x = 0;
i = x++;
j = ++x;
```

The increment operator acts differently depending on where it is placed. In front of a variable, the increment is performed before the assignment. Incrementing after causes the assignment to be performed first. So in the example, `i` is 0 (x before increment) and `j` is 2 (x after increment).

This code is a puzzle to some people. But you don't have to remember obscure details like this if you never write code like this. If you simplify the example, it is no longer a puzzle:

```
i = 0;
j = 0;
x = 0;

i = x;
++x;
++x;
j = x;
```

Rule 4-13:

Avoid side effects.

Rule 4-14:

Put the operator ++ and -- on lines by themselves. Do not use ++ and -- inside other statements.

Assignments in other statements

C also allows the programmer to put assignment statements inside other statements. For example:

```
/* Poor practice */
if ((result = do_it()) == 5)
    printf(“It worked\n”);
```

This is another example of a side effect that needs to be avoided. You could just as easily have written this:

```
/* Good practice */
result = do_it();
if (result == 5)
    printf("It worked\n");
```

The second form not only avoids the side effect, but it is simple and clear. The first form is compact, but remember — your goals are readability and reliability.

Unintentional assignments inside other statements can quickly cause trouble. Consider this example.

```
if (result = 5)
     printf("It worked\n");
```

This fragment should print “It worked” only when result is 5. But the code contains a bug. What it actually does is to assign 5 to result, check against (humm... no, 5 is not 0 this time) and print unconditionally.

Experienced programmers recognize this as the old = vs. == bug. They remember it from the cold, dark night when they stayed up till 2 in the morning staring the bug in they eye a dozen times and not recognizing it the first eleven.

Novice programmers, be warned: you will make this mistake, and it will cause you a great deal of pain.

This error is so common that now many compilers issue a warning when they see code like this. For example:

```
Borland C++ Version 3.00
Copyright (c) 1991 Borland International
Warning test.c 5:
Possibly incorrect assignment in function main
```

Rule 4-15:

Never put an assignment statement inside any other statement.

When to use two statements per line

Although there is a rule — one statement per line — don't be fanatical about it. The purpose of the rule is to make the program clear and easy to understand. In some cases, putting two or more statements on one line improves clarity. For example, consider the following code:

```
/* Not as clear as it can be */

token[Ol.word = "if";
token[Ol.value = TOKEN_IF;

token[l].word = "while";
token[l].value = TOKEN_WHILE;

token[2].word = "switch";
token[2].value = TOKEN_SWITCH;

token[3].word = "case";
token[3].value = TOKEN_CASE;
```

This can easily be rewritten as:

```
/* Clearer */
token[O].word = "if";     token[O].value = TOKEN_IF;
token[l].word = "while";  token[l].value = TOKEN_WHILE;
token[2].word = "switch"; token[2].value = TOKEN_SWITCH;
token[3].word = "case";   token[3].value = TOKEN_CASE;
```

There is a pattern to this code. The first example obscures the pattern. You can still see it, but it's not as clear as in the second case, which is coded in two statement per line. To make the pattern clearer, the statements are organized in columns.

Rule 4-16:

If putting two or more statements on a single line improves program clarity, then do so.

Rule 4-17:

When using more than one statement per line, organize the statement into columns.

Logic and Indentation

Over the years many people have tried to develop a way to create a document makes the logic and execution flow of a program easy to understand.

Flowcharts were an early attempt. They presented the program visually using special symbols to denote things like branch statements, input/output, termination. Figure 4-1 on the following page shows a sample flowchart. charts were excellent for small programs, but for programs of nominal size grew too big and bulky. (I remember seeing one that consisted of hundred boxes spread across a 6x5 foot grid of 11x13 inch paper. It took up whole wall of a conference room. Although it was impressive, no one could understand the whole thing.)

Another problem with early flowcharts was that at the time very I computers could do graphics. (Most couldn't even do lowercase text.) result, all flow charts had to be done by hand, and redone if the program changed.

When ALGOL and other structured languages were invented, people discovered that the could use indentation to represent levels of control This is used in C.

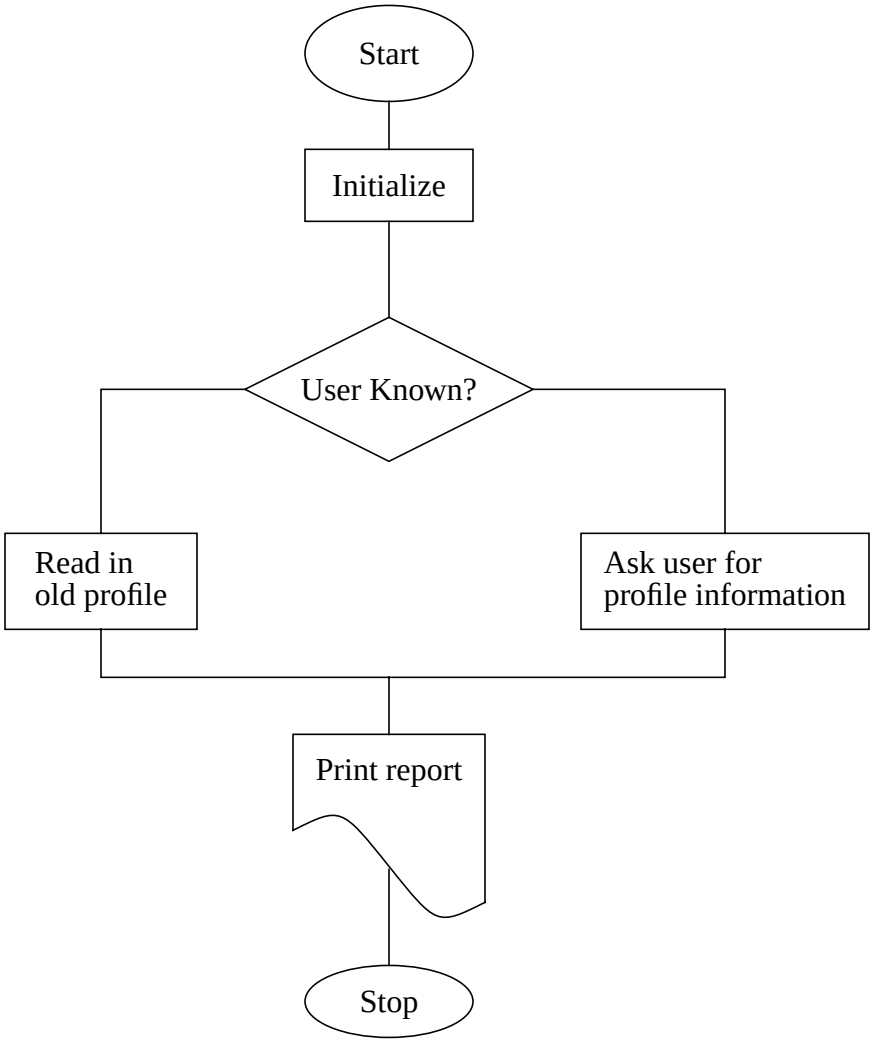

Figure 4-1: A small flow card

For example:

```
while (count > 0) {
     if (balance[index] == EOF_MARKER)
         count =-1;
     else
        total += balance[index];
    ++index;
}
```

In this program fragment, you can easily see that the body of the **while** contains both the **if** and the `++index` statement. The statement `count = -1` is indented an extra level, giving a visual clue that it is part of the **if.**

There are several indentation styles, but they all indent one level for each level of logic.

Rule 4-18:

Indent one level for each new level of logic.

Indentation styles

There are many different styles of indentation, and a vast religious war being waged in the programming community as to which is best. I won't t take sides, but I will present the advantages and disadvantages of each style (Incidentally, the style used throughout this book is the Short Form, chosen only because I'm used to it.)

Short form

In the Short Form, the open brace ({) is put at the end of a line. The text within the braces is indented one level. The close brace (}) is aligned with the beginning of the corresponding statement.

Example:

```
/* Short form indentation
while (! done) {
    printf(“Processing\n”);
    next_entry();
}
if (total <= 0) {
     printf("You owe nothing\n");
     total = 0;
} else {
     printf("You owe %d dollars\n". total);
     all_totals = all_totals + total;
     if (total > 1000)
         printf("You owe a lot\n");
```

The advantage of this style is that it doesn't waste vertical space. The problem is that corresponding braces are not put in the same column. For example, the brace that closes the **while** lines up with the “w” in **while,** not with the brace at the end of the line. This makes it a little more difficult to match braces.

Braces stand alone.

In the Braces Stand Alone method, all braces are placed on separate lines:

```
/* Braces stand alone */
while (! done)
{
    printf("Processing\n");
    next_entry();
}
if (total <= 0)
{
    printf("You owe nothing\n");
    total = 0;
}
else
{
    printf("You owe %d dollars\n", total);
    all_totals = all_totals + total;
}
if (total > 1000)
    printf(“You owe a lot\n”);
```

The advantage of this is that the braces are aligned. The disadvantage is that it takes up more vertical space and tends to spread out the code.

Braces indented too.

This variation on the Braces Stand Alone method indents not only the statement within the braces, but also the braces themselves:

```
/* Braces stand alone */
while (! done)
    {
    printf("Processing\n");
    next_entry();
    }
if (total <= 0)
    {
    printf("You owe nothing\n");
    total = 0;
    }
else
    {
    printf("You owe %d dollars\n", total);
    all_totals = all_totals + total;
    }
if (total > 1000)
    printf(“You owe a lot\n”);
```

This form of indentation is not as common as the other two. It also has the problem of spacing out the code somewhat.

Variations.

One variation on the standard indentation styles concerns if statements that affect a single line. For example:

```
if (total > 1000)
     printf("You owe a lot\n");
```

This style of indentation can create confusion, as illustrated by the following

example:

```
/* Problem code */
if (index < 0)
    fprintf(stderr,"Error: Index out of range\n");
    exit (8);
```

At first glance, it looks like the program will print an error message only if `index` is out of range. (That's what the programmer intended.) But on closer inspection, you'll notice that there are no braces enclosing the two statements under the if. In fact, the code is indented incorrectly.

Indented correctly, the code looks like this:

```
/* Problem code */
if (index < 0)
    fprintf(stderr,"Error: Index out of range\n");
exit (8);
```

The problem is confusion between multi-line if controlled statements and single-line statements. To solve this problem, put single-line statements and their **if**s on the same line:

```
if (total > 1000) printf("You owe a lot\n");
```

This makes very clear that the **if** affects only one line. The problem is that it makes the *printf* line a little more difficult to find and breaks the one-statement-per-line rule.

How much to indent

In this book I indent four spaces for each logic level. Why four? Here are some examples of various indentations.

Two Spaces:

```
/* Short form indentation */
while (! done) {
  printf(“Processing\n”);
  next_entry();
}
if (total <= 0) {
  printf("You owe nothing\n");
  total = 0;
} else {
  printf("You owe %d dollars\n", total);
  all_totals = all_totals + total;
}
  if (total > 1000)
    printf("You owe a lot\n");
```

Four Spaces:

```
/* Short form indentation */
while (! done) {
    printf(“Processing\n”);
    next_entry();
}
if (total <= 0) {
    printf("You owe nothing\n");
    total = 0;
} else {
    printf("You owe %d dollars\n", total);
    all_totals = all_totals + total;
}
    if (total > 1000)
        printf("You owe a lot\n");
```

Eight Spaces:

```
/* Short form indentation */
while (! done) {
        printf(“Processing\n”);
        next_entry();
}
if (total <= 0) {
        printf("You owe nothing\n");
        total = 0;
} else {
```

```
            printf("You owe %d dollars\n", total);
            all_totals = all_totals + total;
    }
            if (total > 1000)
                    printf("You owe a lot\n");
```

The advantage of a smaller indent is that you don't run into the right margin as quickly. The disadvantage is that it's hard to tell the various levels apart.

Larger indents are easier to read, but larger indents mean that you run out of room faster.

Several researchers have studied this problem in detail. They started with the same program and indented it using different indent sizes. They then gave the various flavors of the program to a set of graduate students and told them each to enhance it by adding some additional commands. The students had never seen the program before. The researchers measured time amount of time it took each student to understand and fix the program. As a result of this and other studies like it, they concluded that four spaces is the ideal indentation.

Rule 4-19:

The best indentation size is four spaces.