Indenting C Programs

Last Revision: December, 1998

(McCann)

Having trouble deciding how to arrange statements relative to their neighboring statements? Losing points on assignments because your instructor says you aren't aligning your program statements correctly? You've come to the right web page. The sections of this page cover the guidelines of acceptable code indentation. If you are a beginning programmer, you may not know how to use all of the statements covered in this document. Just start reading from the beginning, and skip over the sections that cover statements you don't know yet. Or, if you want to learn about indenting a particular statement, use the following outline to jump to the appropriate section.

Topic Outline

---

1. Why Indentation is Important

1. Clarity

When people learn programming, they learn best when they attempt simple programs of minor utility. An unfortunate consequence of this is that the programs are rarely revisited by their authors. Try it sometime; dig out one of your old programs and see if you can easily figure out the details of its operation. If you used a poor indentation style, you will probably have trouble following the logic of the code. And it's your own code! Embarrassing, isn't it?

Now think about larger, more complex programs. They tend to be written to do useful chores, and because they are useful they tend to "hang around" for a long time. Then, eventually, someone decides that the program would be even more useful if it did a few more chores in addition to its current duties. Before long, an unfortunate programmer will be assigned to make the necessary additions to the program, and they will struggle with the task if the code is hard to comprehend.

What could be worse? This: *YOU* could be that unfortunate programmer!

The moral: When you write code, no matter how insignificant or trivial it may seem to you now, write it to be easy to read. In particular, adopt a standard indentation style for your code, and stick with it throughout the program. Other programmers will thank you for it.

2. Existing Standards

People have been writing programs for several decades now, so it should be no surprise that some indentation conventions have been adopted over the years. What may be a surprise is that authors of textbooks for beginning programmers rarely devote much time to a discussion of acceptable indentation styles. Instead, they usually adopt a style for their example code and encourage their readers to adopt it as their own.

The problem with that approach, as far as I'm concerned, is that it doesn't encourage the beginner to adopt a style that the beginner finds appealing. If you like a style, you are more likely to use it. I'd much rather see you use a consistent, acceptable style that you like and I don't than see you pretend to use a consistent, acceptable style that you hate but I adore. Programming is an art, and its beauty is in the eye of its beholder. Try to find a workable intersection between utility and attractiveness.

Having said that, you're much better off finding and adopting an existing style than creating one from scratch. Where can you go to find examples of existing styles?

- **Textbooks**. Each author has their own style. Try visiting the library and look though some books that cover your programming language. This is particularly helpful when you are using a relatively uncommon construct of the language and you have no idea how to

indent it.

- **The Internet**. Lots of instructors have created lots of web pages for their programming courses, and some of them have taken the time to create documents on programming style, including indentation.

  Also, there do exist some general programming style documents. Visit my Programming Style Documents page for a few links. Even if there isn't one for your particular language, the ideas covered in these documents are usually applicable to other languages as well.

- **Language Standards Documents**. Your language is almost certainly maintained by an ANSI and/or ISO committee. If you can find a published copy of the language standard document, you can find out what indentation style they like to use. Often, the style from the standard is the one textbook authors adopt.

What about programs that will automatically indent source code for you? Such "crutches" do exist, but I discourage their use on code that is being written from scratch because you are likely to develop bad habits if you begin to rely on the program to fix your mistakes for you. Someday you will need to write a program on a different computer system that doesn't have that crutch, and you'll be in trouble. If you're too lazy to indent your own code, you're probably too lazy to be a good programmer.

---

2. Basic Tenets of Indentation

There are a few indentation habits that you should learn to do right away, and they should be done every time you write code.

1. The "Golden Rule" of Indentation

Always indent the body (bodies) of a statement a uniform amount from the first character of the statement. The "body" of a statement is the action (or set of actions) that the statement controls. Statements that have bodies include:

- Loops
- If statements
- Subprograms

For example, here is an IF-ELSE statement. There is a set of actions to be performed when the condition is true, and another set to be carried out when the condition is false. These two sets are controlled by the IF-ELSE statement, and thus should be indented from the "i" in the word "if":

```
if (victor(human)) {
        human_wins++;
        printf("I am your humble servant.\n");
} else {
        computer_wins++;
        printf("Your destiny is under my control!\n");
}
```

Why do we do this? Because this makes it easy for the reader to understand which actions are performed as part of the IF-ELSE statement. The alternative (no indentation) is much harder to understand:

```
if (victor(human)) {
human_wins++;
printf("I am your humble servant.\n");
} else {
computer_wins++;
printf("Your destiny is under my control!\n");
}
```

Sure, you can figure it out, but you have to work at it. Do you like to struggle to comprehend the logic of other people's programs? Of course not. Apply this observation to your own code.

By how many spaces should you indent the bodies of the statements? If you follow these guidelines, you'll do fine. Indent by:

- at least 2 spaces
- no more than 8 spaces (1 tab)
- the same amount for all statement bodies

If you indent by only one space, the reader will find it difficult to notice the change. If you indent by too much, you'll soon run out of room on the line for your program statement. And, if you aren't consistent, the reader may think that two statement bodies are at the same level of nesting when they aren't. I like to use tabs for indentation; it's hard to miscount them, and they're easy to use.

2. The Main Function

To indent the main function, just apply the "Golden Rule". I like to apply the rule to the variable declaration types, too, but then I prefer to take it step further and place each variable on a separate line. This leaves space after the variable name for a comment (but that's the subject of another style document!).

Example:

```
int main (int argc, char *argv[])
{
        int     a,      /* an integer variable */
                b;      /* another lousy comment */
        float   c;      /* always write informative comments! */

        printf("Give me an 'a' : ");
        scanf("%d",&a);
        printf(" Give me a 'b' : ");
        scanf("%d",&b);
        . . .
}
```

3. Auxiliary (User-Defined) Functions

In general, handle these the same way that you handle the main program; again, be consistent.

It is common practice to align the definitions of all of the subprograms of the file with the main program, like this:

```
#include <stdio.h>

int function_a (int z)
{
        . . .
}

int function_b (char y)
{
        . . .
}

int function_c (float x)
{
        . . .
}

int main (void)
{
        . . .
}
```

C doesn't permit functions to be nested within other functions, so we don't have to worry about how to handle that situation.

One difference between the main function and the others is the length of the parameter list. Should you write a function with a parameter list longer than the width of the line, feel free to divide the list across lines, but try to indent the second line (and any subsequent lines) by a large amount so that the reader can see that it is part of the parameter list, and not a local variable declaration.

Do this:

```
int binary_search (struct collection list[], int first_index,
                        int last_index, key_type target);
```

Or maybe this:

```
int binary_search (struct collection    list[],
                   int                  first_index,
                   int                  last_index,
                   key_type             target          );
```

But don't do this:

```
int binary_search (struct collection list[], int first_index,
int last_index, key_type target);
```

---

3. Two Common Compound Statement Indentation Styles

There is uniform agreement that indenting the bodies of statements is a good idea. There is much less agreement on where to place the compound statement braces ('{' and '}'). While there are many variations, there are only two major approaches. Both have the goal of making it easy to match the closing brace with the corresponding statement.

As with indentation, the key is consistency. If you decide to align the braces around the body of a function, then align the braces around the bodies of ALL functions. When a reader sees one function in your program, s/he will expect to see the same styles used in all of your functions.

1. Align "{" with "}"

You can't go too far wrong with this approach. By aligning the braces and indenting the body of the compound statement, the reader has no doubt about which statements form the compound statement's body.

For example:

```
if (victor(human))
  {
        human_wins++;
        printf("I am your humble servant.\n");
  }
else
  {
        computer_wins++;
        printf("Your destiny is under my control!\n");
  }
```

A minor disadvantage of this method is that it consumes a lot of lines. Now, lines are not expensive, but programmers like to be able to place a lot of code on the screen at one time, and this method "wastes" lines.

2. Align Control Statement with "}"

In programming, compound statements are not inserted into the code just for the heck of it; they are added because a set of statements needs to be grouped. And why do we have to form groups? To create the body of

a statement, of course. So, we can reason that the closing brace is also marking the close of the statement, not just the close of the body.

With that logic in mind, we might choose to align the closing brace with the statement whose body it closes. The opening brace is placed at the end of the first line of the statement.

Example:

```
if ( (!victor(human)) && (!victor(computer)) ) {
        number_of_draws++;
        printf("We are well-matched.\n");
}
```

---

4. Selection Statements: IF, IF-ELSE, and SWITCH

In this section, I present a variety of code examples to demonstrate how to indent various forms of the selection statements.

1. IF with a Simple Body

If the body of an IF statement is just a single statement, then there isn't a need for a set of braces surrounding it.

Should the statement be short enough, it is acceptable to place it on the same line as the IF. Many people (myself included) don't care for this style because it tends to hide the body of the IF. Remember, the idea is to make the program's logic clear. Having said that, here's an example of this style:

```
if (count < 0) count = 0;
```

By placing the (indented) body on the next line, you make the whole statement (and also its purpose) much easier to identify:

```
if (count < 0)
        count = 0;
```

I maintain that it is not a bad idea to go ahead and place braces around these one-statement bodies, too. The reason: Should you later decide to add a statement to the body, you don't have to remember to add the braces to complete the compound statement:

```
if (count < 0) {
        count = 0;
}
```

2. IF with a Compound Body

Nothing new to say here; our previous examples have covered this case. Just indent the statements of the body uniformly and stick with your adopted brace-placement style.

```
if ( (!victor(human)) && (!victor(computer)) ) {
        number_of_draws++;
```

```
        printf("We are well-matched.\n");
}
```

3. IF-ELSE with Simple Bodies

The same rules apply here as they apply to the ordinary IF. Braces are not necessary, but I like to use them anyway.

A "no-braces" example:

```
if (temperature < 55)
        printf("It could be warmer...\n");
else
        printf("It could be colder...\n");
```

If we add braces to this, how should we align them? You can align the braces, of course, as we did in one of the previous examples. Otherwise, you might choose to align both closing braces with the "i" in "if". Here's the same example with that brace alignment:

```
if (temperature < 55) {
        printf("It could be warmer...\n");
} else {
        printf("It could be colder...\n");
}
```

I like this approach, because I think it helps to tie the entire IF-ELSE statement together.

4. IF-ELSE with Compound Bodies

Of course, in this case the braces are required. Just be consistent with your style. Here's a repetition of one of the earlier examples that assumed the use of the "align the braces" approach:

```
if (victor(human))
  {
        human_wins++;
        printf("I am your humble servant.\n");
  }
else
  {
        computer_wins++;
        printf("Your destiny is under my control!\n");
  }
```

5. IF-ELSE with Mixed Simple and Compound Bodies

This case can be a little awkward, particularly if you like to leave off the braces from simple bodies, like this:

```
if (square_footage > 2500) {
        tax = square_footage * 0.025;
        printf("Do you need a loan?\n");
} else
        tax = 0;
```

That just looks funny to me, but it's perfectly acceptable. As you already know, I'm partial to the idea of "bracing" even the simple bodies:

```
if (square_footage > 2500) {
        tax = square_footage * 0.025;
```

```
        printf("Do you need a loan?\n");
} else {
        tax = 0;
}
```

6. SWITCH

The SWITCH statement is very handy, but it does present some novel indentation challenges. In a way, the SWITCH has multiple bodies, if you think of each CASE as the start of another body of code.

The standard way to indent the SWITCH is to align the CASEs with the SWITCH, and then indent the statements from the level of the CASEs. It is really just an extension of the IF-ELSE indentation scheme. For example:

```
switch(letter) {
case ' ':
        printf("This is a space,\n");
        break;
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
        printf("This is a vowel.\n");
        break;
default:
        printf("This is is something else.\n");
}
```

I've never liked that approach; to me, aligning all of those CASEs with the SWITCH breaks up the flow of the statement, and makes it too hard to recognize the end of the SWITCH. I prefer to indent the CASEs from the SWITCH, and then indent the statements from the CASES in a slightly different way:

```
switch(letter) {
        case ' ': printf("This is a space,\n");
                  break;
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u': printf("This is a vowel.\n");
                  break;
        default : printf("This is is something else.\n");
}
```

What makes this acceptable? The indentation is consistent, and the closing brace aligns with the corresponding statement. So long as I stick with this style for all SWITCH statements in the program, there's no problem.

---

5. Iteration Statements: WHILE, FOR, and DO-WHILE

If you can handle indenting IFs and IF-ELSEs, you can handle loops. There's really nothing new to know.

1. WHILE with a Simple Body

If the body of the WHILE loop is short enough, you could place it on the same line as the loop statement, but, as with the IF, I don't recommend this practice:

```
i = 0;
while ((i < LIST_SIZE) && (list[i] != MAX_SCORE)) i++;
```

Instead, make it a habit to place the indented body on the next line:

```
i = 0;
while ((i < LIST_SIZE) && (list[i] != MAX_SCORE))
        i++;
```

And if you're like me, you will add the braces just to be safe:

```
i = 0;
while ((i < LIST_SIZE) && (list[i] != MAX_SCORE)) {
        i++;
}
```

It is possible to have a WHILE loop that doesn't have *any* statements in its body. See the discussion of the FOR loop with a Simple Body later in this page for ideas on how to handle this situation.

2. WHILE with a Compound Body

There are no choices to make here, other than which closing brace alignment to use. (And because you want to try to use the same brace style for your IFs and your loops, you may have made this choice already!)

```
i = 0;
while (i < LIST_SIZE) {
        printf("Element %d is %d.\n",i,list[i]);
        i++;
}
```

3. FOR with a Simple Body

FORs are interesting because it is not unusual to be able to write a FOR loop without a body. (This is possible for WHILEs, too; it's just not as commonly seen.) Consider this example of finding the first occurrence of a value within an array:

```
for (i=0; (i < LIST_SIZE) && (list[i] != MAX_SCORE); i++);
if (i >= LIST_SIZE)
        printf("Sorry; %d was not found.\n",MAX_SCORE);
```

There's no body! Looks strange, doesn't it? Worse, it looks confusing; the semicolon at the end is easy to overlook, which might lead the reader to believe that the "if" statement is the loop's body. To help the reader, it is best to place the semicolon on the next line:

```
for (i=0; (i < LIST_SIZE) && (list[i] != MAX_SCORE); i++)
        ;
if (i >= LIST_SIZE)
        printf("Sorry; %d was not found.\n",MAX_SCORE);
```

Even better, put a comment with it:

```
for (i=0; (i < LIST_SIZE) && (list[i] != MAX_SCORE); i++)
        /* null loop body */ ;
if (i >= LIST_SIZE)
        printf("Sorry; %d was not found.\n",MAX_SCORE);
```

Apart from that situation, just indent the FORs like you would the WHILEs:

```
for (i=0; i < LIST_SIZE; i++)
        printf("Element %d is %d.\n",i,list[i]);
```

4. FOR with a Compound Body

Nothing new here!

```
for (sum=0, i=0; i < LIST_SIZE; i++) {
        printf("Element %d is %d.\n",i,list[i]);
        sum += list[i];
}
```

5. DO-WHILE with a Simple Body

If the DO-WHILE has a simple body, you could just put the whole loop on one line:

```
i = -1;
do  i++;  while ((i < LIST_SIZE) && (list[i] != MAX_SCORE));
```

If you don't care for that style (and I don't blame you), you can place the parts on separate lines and indent the body from the level of the DO and WHILE keywords:

```
i = -1;
do
        i++;
while ((i < LIST_SIZE) && (list[i] != MAX_SCORE));
```

The problem with this approach is that the reader might get confused by the word "while" on the left edge; it would be easy to mistake the "while" of the DO-WHILE loop for the "while" that starts the WHILE loop. If you are of the habit of always adding braces around loop bodies, you can use the closing brace to avoid confusion:

```
i = -1;
do {
        i++;
} while ((i < LIST_SIZE) && (list[i] != MAX_SCORE));
```

6. DO-WHILE with a Compound Body

In this case, you'll need braces around the body (of course!). If you like the "align the braces" style, the DO-WHILE loop will look a little odd:

```
i = 0;
do
{
        printf("Element %d is %d.\n",i,list[i]);
        i++;
} while (i < LIST_SIZE);
```

The "do" seems to be a little lost on a line by itself, but there's nothing wrong with this. I prefer the look of the loop with the closing brace

aligned with the "do" only:

```
i = 0;
do {
        printf("Element %d is %d.\n",i,list[i]);
        i++;
} while (i < LIST_SIZE);
```

---

6. Indentation of Nested Statements

When you have code with complex statements as part of (nested inside of) other complex statements, indentation is especially important. If you get sloppy, you may end up fooling the reader (and perhaps even yourself) into believing that a statement is outside of the body of a statement, when in fact it is really inside (or vice-versa).

1. "Levels" of Indentation

I think it helps to consider each tab at the start of a line as a "level" of indentation. For example, here's a fairly simple program with the indentation levels listed ahead of each line:

```
[0] int main (void)
[0] {
[1]        int     i;
[1]        float   sum = 0, thousandth = 0.001;

[1]        for (i=1; i<=1000; i++) {
[2]                sum += thousandth;
[2]                if (i % 100 == 0)
[3]                        printf("After %4d iterations, sum = %.10f\n",i,sum);
[1]        }

[1]        printf("\nUsing %%.2f in printf rounds sum to be %.2f,\n",sum);

[1]        if (sum != 1.00)
[2]                printf("but C says that sum does not equal 1.00.\n");
[1]        else
[2]                printf("and C says that sum does equal 1.00.\n");

[1]        return 0;
[0] }
```

The observation to make is that the body of a statement is one level deeper than the statement's own level. This is how nested statements are supposed to be indented; always indent the bodies from the level of the corresponding statement.

Consider the FOR loop in the above program. The FOR statement is at level 1, and so its closing brace is also at level 1. The body of the FOR loop consists of two statements, the increment of sum and the IF, both of which are therefore indented a level deeper than the FOR. The IF has a body of its own, which in turn is a level deeper than the IF. Because of this consistency in indentation, the structure of the loop is clear, and we will not assume that the printf in the IF's body is a statement in the body of the FOR.

2. Nested IF-ELSE Statements

Here's an exception to the "always indent bodies a level deeper" rule. It is frequently necessary to create a deeply-nested set of IF-ELSE statements.

For instance, consider the classic example of trying to assign a letter grade to an exam score. If we stick to the normal indentation guidelines, here's what we'll create:

```
if (score >= 89.5)
        grade = 'A';
else
        if (score >= 79.5)
                grade = 'B';
        else
                if (score >= 69.5)
                        grade = 'C';
                else
                        if (score >= 59.5)
                                grade = 'D';
                        else
                                grade = 'F';
```

This is a good example of the "stair-step effect" that results from this sort of nesting. If there are only a few levels of nesting, there is no problem. But if there are a lot of levels, or if the code with each level is of even moderate complexity, we'll find that we are running out of room to write statements on each line. Instead of creating clear, easy-to-read code, we'll be creating a mess along the right side of the screen.

To combat this problem, programmers relax the normal indentation rules in favor of a "compressed" approach that doesn't look as good, but that is still reasonably easy to understand, especially after you've seen it used a few times. The compromise is to place each IF on the same line as the previous ELSE, and to align the ELSEs. Here's the result of this style on the previous example:

```
if (score >= 89.5)
        grade = 'A';
else if (score >= 79.5)
        grade = 'B';
else if (score >= 69.5)
        grade = 'C';
else if (score >= 59.5)
        grade = 'D';
else
        grade = 'F';
```

As I said, not as nice, but more functional.

A similar problem can occur with nested loops. Should that happen to you, one solution is to place the body of the inner-most loop or loops in a separate subprogram.

Should you find yourself with a complex piece of code that has loops inside of IFs inside of two other loops inside of..., I have two suggestions. First, strongly consider the use of a subprogram or two to hold the inner-most code. Second, if for some reason you don't want to use a subprogram, you could indent each line by a reduced amount -- 4 spaces instead of 8 per level, for example. If you do this, you should apply the change to the ENTIRE program, just to be consistent.

---

7. "But How Do I Indent *THIS* Correctly?"

Obviously, I can't provide examples to help you indent *every* possible situation. Someday you will undoubtedly find yourself writing a section of code that you can't indent in a pleasing way. Just do the best you can. Remember to indent whatever the code is in a consistent manner that is as easy to read as you can make it.

---

8. Self-Test

OK, let's see how well you learned these lessons. Here's a really poorly indented program that I lifted from my Developing Good Programming Style web page. Cut-n-paste it from the web browser into an editor or a word processor and try to fix it. When you are done, click here to check your answer against mine.

```
#include <stdio.h>
int main(void) {
int seg[10] = {6,2,5,5,4,5,6,3,7,6};
int d1, d2, d3, d4, m=0, td, ts;
for (d1=0; d1<2; d1++)
  for (d2=0; d2<10; d2++)
         for (d3=0; d3<6; d3++)
 for (d4=0; d4<10; d4++)
    if ((!((d1==0)&&(d2==0))) && (!((d1==1)&&(d2>2)))) {
          if (d1==0) {
ts = seg[d2] + seg[d3] + seg[d4];
       td = d2 + d3 + d4;
  if (ts == td) { m++;
printf(" %1d:%1d%1d\n",d2,d3,d4); }
       } else {
ts = seg[d1] + seg[d2] + seg[d3] + seg[d4];
 td = d1 + d2 + d3 + d4;
        if (ts == td) { m++;
          printf("%1d%1d:%1d%1d\n",d1,d2,d3,d4); }
} }
return 0; }
```

Want to learn more about good programming style? Please visit my Programming Style Documents page for pointers to additional documents about programming style.

Do you have a comment on this page? I'd like to hear it; you can email me at mccannl@acm.org.