



Coding Standards

Joel Castellanos

All *projects* must follow these coding standards. Lab assignments, quizzes and exams do not need to follow these standards.

These standards do not represent the best nor the only good way to write C code. If you have experience with C, C++ or a C like language (such as Java) then these standards may not be the standards you are used to using. However, in this class, these are the standards we will use. There are three primary reasons for a standard: First, a standard makes it easier for the instructors to read your code. Second, a class standard makes it easier for a grader to recognize when you have not used a consistent standard. Often when each student is allowed to define his or her own standard, students switch standards multiple times in a single project. It is tedious for a grader to deduce each person's standard and then check for self-consistency. Third, it is good practice for you to learn to follow a standard.

The standard chosen is intentionally different from that used in the textbook. Thus, any code that you copy from the textbook will require reformatting.

As we progress through the semester, none of these standards will change, however additional standards will be appended.

1. The first lines of every source file must be comments of the form:

```
/* **** */
/* **** Author:      YourFirstName YourLastName **** */
/* **** Course:      CS-241 section #          **** */
/* **** Updated:     MM/DD/YYYY                **** */
/* **** */
```

Code Example 1

2. All variable names shall begin with a lower case letter.
3. All functions will be given descriptive names.
4. Variables with a scope of more than one page (25 lines of code, comments and white space) shall be given descriptive names. The temporary variable `c` in two line, code example 6 is a perfect case of when a single letter variable name is sufficiently descriptive.
5. All identifiers declared with `#define` or with the `const` qualifier will be all uppercase.

6. Open brackets will be placed at the beginning of a line (not at the end).

ok	<pre>if (x == 5) { y=y+1; }</pre>
Not CS-241 standard	<pre>if (x == 5) { y=y+1; }</pre>

Code Example 2

7. Each closing bracket shall be on a line by itself. The only exception is that comments may be placed on the line with a closing bracket. The `else if` on the same line as the closing bracket on line 9 of code example 3 is not the CS-241 standard and will be marked down.

1	<pre>if (x == 5)</pre>
2	<pre>{ y=y+1;</pre>
3	<pre>} //ok to have comment here, but nothing else.</pre>
4	<pre>else if (x == 7)</pre>
5	<pre>{ y=y+2;</pre>
6	<pre>}</pre>
7	<pre>if (x == 5)</pre>
8	<pre>{ y=y+1;</pre>
9	<pre>} else if (x == 7)</pre>
10	<pre>{ y=y+2;</pre>
11	<pre>}</pre>

Code Example 3

8. Whenever a structure spans more than one line, brackets must be used. For example:

ok	<pre>if (x == 5) y=y+1;</pre>
ok	<pre>if (x == 5) { y=y+1; }</pre>
Not CS-241 standard	<pre>if (x == 5) y=y+1;</pre>

Code Example 4

9. No function shall contain more than 50 lines of code. This count does not include comments or blank lines. However, the count includes other non-executable lines such as those with a single closing bracket, variable declarations and compiler directives.

10. Code blocks will be indented to show the block structure with *two spaces* per level. Tab characters shall *not* be used for indenting. The block structure indenting is correctly shown in code example 5. For example, the open bracket on line 8 and the corresponding closing bracket on line 14 are both indented two spaces because one starts and the other ends a structure that is one level inside the function `main()`. All statements within this structure must be indented 4 spaces. Lines 9 through 12 make a single statement that contains a structure. The statement is indented 4 spaces. The internal structure of that statement has additional indenting. An open bracket may be on a line all by itself, as in line 5, or may include a statement as in line 8.

```
1 #include <stdio.h>
2 #define DIVIDE_EX_ASCII 247
3
4 int main()
5 {
6     int i;
7     for (i=1; i<100; i++)
8     { int remainder = i % 3;
9       if (remainder)
10      { printf("%d%c3 has a remainder of %d\n",
11            i, DIVIDE_EX_ASCII, remainder);
12      }
13      else printf("%d is divisible by 3\n",i);
14    }
15    return 0;
16 }
```

Code Example 5

11. Variables shall be declared in a scope no larger than they are used. Exceptions to this are identifiers declared with `#define` or with the `const` qualifier. For example, the identifier `remainder` in code example 5 is only used inside the `for` loop. Therefore, `remainder` must be declared *inside* the loop. Declaring `remainder` on line 5 would violate this standard. `DIVIDE_EX_ASCII` is also only used within the loop, however, since it is not a variable, it may be declared at a larger scope than it is used. The variable `i` must be declared outside the loop in order for the code to compile. For example, `gcc` with no options does not allow C++ syntax such as:

```
for (int i=1; i<100; i++)
```

12. No line shall be more than 80 characters. The best way to avoid overly long statements is by not doing too much in a single statement. Lines 2 through 9 in code example 6 are much easier to read than line 1. The `if` statement in line 1 is made less complex by the introduction of temporary variables `Volume1` and `Volume2`. The foolhardy argue it is more efficient not to create such temporary variables. Two counter arguments:

- It is bad programming to sacrifice readability for optimization *that is not needed*. After you notice your code is running too slowly, then discover where most of the time is being spent and optimize those places only.
- Let the compiler do the optimization: any modern C compiler will not create main-memory write and reads to the temporary variables `volume1` and `volume2`. In line 1 and lines 2 through 4, the return value for each call to `getVolume` will be stored in a register.

Thus, this type of “micro optimization” is no optimization and just plain bad programming.

1	<code>if (getVolume(length1, width1, height1) > getVolume(length2, width2, height2)) printf("box 1 is bigger\n"); else printf("box 2 is bigger\n");</code>
2	<code>int volume1 = getVolume(length1, width1, height1);</code>
3	<code>int volume2 = getVolume(length2, width2, height2);</code>
4	<code>if (volume1 > volume2)</code>
5	<code>{ printf("box 1 is bigger\n");</code>
6	<code>}</code>
7	<code>else</code>
8	<code>{ printf("box 2 is bigger\n");</code>
9	<code>}</code>

Code Example 6

Code example 7 shows another case where a temporary variable can shorten a line and improve readability. Creating the temporary variable `c` also improves code maintenance: If the code changes so that the comparison needs to check `stack[topOfStack]` or `stack[topOfStack-2]`, then Line 2 and 3 require only a single change while line 1 requires 4 changes. In code such as line 1, it is common for programmers to make the error of changing only 3 of the 4 cases and creating a hard to spot bug. It should be noted that lines 2 and 3 are no more efficient than line 1 because a good compiler will optimize the repeated subtraction and indirect reference in line 1.

1	<code>if (stack[topOfStack - 1] == '*' stack[topOfStack - 1] == '+' stack[topOfStack - 1] == '-' stack[topOfStack - 1] == '/')</code>
2	<code>char c = stack[topOfStack - 1];</code>
3	<code>if (c == '*' c == '+' c == '-' c == '/')</code>

Code Example 7

<continued> No line shall be more than 80 characters.

There are times when breaking a long statement in to multiple statements is more awkward than keeping the long statement. In such cases, the statement should be broken in a *logical place* and each line over which the long statement is continued must be indented. The indenting must be *at least 2 spaces*, but can be more spaces it that improves readability. Code example 8, indents line 3 so that the comparisons match up.

1	<code>if (commandOption == 'f' commandOption == 'c' commandOption == 'd' commandOption == 'g')</code>
2	<code>if (commandOption == 'f' commandOption == 'c' </code>
3	<code>commandOption == 'd' commandOption == 'g')</code>

Code Example 8

13. No code of more than three lines shall be repeated anywhere within a single project. This is solved by either restructuring logic or by placing the repeated code in a function. In code example 9, restructuring was possible. If there had been an `else if` within the block that did *not* use the repeated code, then a function would have been needed.

1	<code>if (foundOperation)</code>
2	<code>{ if (c == '+')</code>
3	<code>{ listPt[listIdx1] = listPt[listIdx2];</code>
4	<code>listPt[listIdx2] = -1;</code>
5	<code>listIdx2 = -1;</code>
6	<code>listValue[listIdx1] = a+b;</code>
7	<code>foundOperation = 0;</code>
8	<code>}</code>
9	<code>else if (c == '-')</code>
10	<code>{ listPt[listIdx1] = listPt[listIdx2];</code>
11	<code>listPt[listIdx2] = -1;</code>
12	<code>listIdx2 = -1;</code>
13	<code>listValue[listIdx1] = a-b;</code>
14	<code>foundOperation = 0;</code>
15	<code>}</code>
16	<code>}</code>
20	<code>if (foundOperation)</code>
21	<code>{ int tmp;</code>
22	<code>if (c == '+') tmp = a+b;</code>
23	<code>else if (c == '-') tmp = a-b;</code>
24	<code></code>
25	<code>listPt[listIdx1] = listPt[listIdx2];</code>
26	<code>listPt[listIdx2] = -1;</code>
27	<code>listIdx2 = -1;</code>
28	<code>listValue[listIdx1] = tmp;</code>
29	<code>foundOperation = 0;</code>
30	<code>}</code>

Code Example 9

Last Word...

Clean coding standards should be followed *as you code*. Not applied at the end. Of course, only the end product is graded; however, if you take the time to maintain proper indenting and formatting as you write your code, then you will create fewer bugs, and bugs you do create will be easier to spot. Also, when you show your buggy code to an instructor, the instructor will require less time to help you.

