**Coding style guide**

The C programming language offers a number of ways to format code. Many programmers abuse this freedom and write unreadable (and thus incomprehensible and unmaintainable) code. While there is more than one way to properly format code, here is a set of guidelines which have been found useful in practice. Note that marks will be taken off for poor formatting (more marks deducted as the term goes on). Some of these guidelines may seem amazingly anal, but they really make a difference when reading code. Remember: you are writing code not just for the compiler, but for other people to read as well. The other person reading your code will most likely be you six months from now, so making sure that your code is readable is extremely important.

In the following, each item has a code associated with it to the left of the description of the item. This code will be used to specify the problem when correcting your code. It is up to you to match the code with the item. Hopefully this will encourage you to read this style guide :-) As a general rule, the earlier items in a section are more important than the later ones and/or represent more common errors.

**The most common style mistakes**

These mistakes occur so often that they're almost universal. Therefore, please pay particular attention to avoiding them. Follow the links to get to the descriptions below. Style mistakes followed by an asterisk (**\***) are caught by the style checker.

* [TABS](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#TABS)**\***   Using tab characters in your code.
* [OPERATOR\_SPACE](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#OPSPACE)**\***   Not putting spaces between operators.
* [COMMA\_SPACE](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#COMMA_SPACE)**\***   Not putting a space after a comma.
* [LINE\_LENGTH](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#LINE_LENGTH)**\***   Writing lines longer than 78 characters.
* [USAGE\_STMT](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#USAGE_STMT)   Missing or inadequate usage statement.
* [EMPTY\_LINES](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#EMPTY_LINES)   Using too many or too few empty (blank) lines in functions.
* [COMMENTS\_FULL\_SENTENCES](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#COMMENTS_FULL_SENTENCES)   Writing comments that are not full sentences.
* [COMMENTS\_GRAMMATICAL](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#COMMENTS_GRAMMATICAL)   Writing comments that are not grammatically correct or are misspelled.
* [COMMENT\_SPACE](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#COMMENT_SPACE)**\***   Not putting a space after the open-comment symbol "/\*" and/or before the close-comment symbol "\*/".
* [COMMENT\_HEADER](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#COMMENT_HEADER)   Not writing a proper comment at the head of a function.
* [FUNCTION\_PROTOTYPES](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#FUNCTION_PROTOTYPES)   Not writing prototypes for all the functions defined in a file.
* [FUNCTION\_BLANK\_LINES](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html#FUNCTION_BLANK_LINES)   Not separating function definitions by blank lines.

**Catalog of style mistakes**

**General**

* **[TABS]**

Never, ever, ever use the tab character (ascii 0x9)! Different people use different tab settings, and code that looks just fine with a tab width of 2 becomes unreadable with a tab width of 8. Unfortunately, many text editing programs will stick in tab characters without making it obvious that they're doing it. If you use emacs for text editing (which I recommend) put the following lines in a file called .emacs, which should be placed in your home directory:

(setq c-mode-hook

'(lambda ( )

(progn

(set-variable 'indent-tabs-mode nil)

;; other customizations, if any, go here

)))

This is actually emacs-lisp code, but don't worry about that. Then exit and restart emacs. Now when you hit the tab key while editing C code, emacs won't actually put any tab characters in your code, but instead will just put in spaces. In addition, emacs is smart enough that when you're editing C code and you hit the tab key, emacs will automatically indent the code to a reasonable point on the line. If you're using emacs in the CS cluster, it should also color your file in a meaningful way (comments will be a different color, for instance). Emacs is very nice to use for editing C code.

If you're using the vim editor instead of emacs, you can see all the tab characters by typing:

:set list

into the editor while in command mode. This will make all tab characters look like "^I" (a circumflex accent followed by a capital I). This makes it easy to go through a file and replace all tabs by *e.g.* four spaces. Better, still, you can put this into your ~/.vimrc file:

set expandtab

and tabs will be printed as spaces.

If you're using an editor other than emacs or vim, your job is harder; you have to go over the line character by character using the forward-character arrow and find out where the tabs are and replace them.

If you don't like removing tabs from your code manually, here's a trick that will help. Let's say you have a file called foo.c and you've run the style checker on it, and every other line has tabs in it. Just do this from the unix prompt (% in this example):

% sed -e 's/\t/ /g' < foo.c > foo.c.notabs

% mv foo.c.notabs foo.c

and your file will no longer have any tabs. On the other hand, this can mess up the indentation, so you should go over it afterwards to make sure it looks presentable and add spaces if necessary. If you don't, and the result is unreadable, I will probably make you redo it.

The "sed" in the command line is a program called sed (which means "stream editor"). It does simple editing on files on a line-by-line basis. So when you type

sed -e 's/\t/ /g' < foo.c > foo.c.notabs

^^^^ 4 space characters here

it executes a command called 's/\t/ /g' on each line of the file foo.c, putting the results into a new file called foo.c.notabs. The line 's/\t/ /g' means "substitute (s) for every tab (\t) character, four space characters (which is what's between the // characters), and do it for every tab in the line (g, which means global)". Note that you have to type this in exactly as I've described it or it won't work.

If you want, you can use more or less than four space characters per tab. Most editors use eight space characters for a tab by default, so that might be a good alternative. That would look like this:

sed -e 's/\t/ /g' < foo.c > foo.c.notabs

^^^^^^^^ 8 space characters here

* **[OPERATOR\_SPACE]**

Use a single space to separate variable names from operators, *i.e.* write

a = b + c \* d;

instead of

a=b+c\*d;

The only exception for this rule is for array subscripts *e.g.*

b = a[i-1] /\* not a[i - 1] \*/

but you can put the spaces in here too if you want. Unfortunately the style checker currently complains if you don't put the spaces in. Don't worry about the warning in this case.

* **[COMMA\_SPACE]**

Always put a space after a comma. There are no exceptions to this rule.

* **[PAREN\_CURLY\_SPACE]**

If you are using a formatting style where the opening curly brace of a block is on the same line as an **if**, **while**, or **for** statement (which I discourage; it's better to put the curly brace on a separate line), make sure that there is a space between the close paren on the line and the open curly brace, *e.g.* do

for (i = 0; i < n; i++) {

/\* code goes here \*/

}

instead of:

for (i = 0; i < n; i++){

/\* code goes here \*/

}

because the latter is hard to read. Similarly, leave a space between an **else** keyword and an opening curly brace if they're on the same line.

* **[LINE\_LENGTH]**

Don't write lines that are longer than 78 characters long. Long lines tend to be wrapped, or worse, to be truncated when printing out the source. Printing out source code is a valuable way to review your code. It is almost never necessary to have long lines, even for long strings; you can always break up a string like this:

printf("this is a really, really, really, really, really, really, "

"really, really, really, really, really, really long string.\n");

and the two strings will be concatenated together. This will work for any number of consecutive strings. Note that this trick only works for literal strings, not for variables which contain (point to) strings.

* **[ANSI\_VIOLATION]**

For portability, you should restrict yourself to pure ANSI-compliant C code exclusively. Note that **gcc** will not do this for you. If you want to be safe you need to use several compiler flags:

gcc -Wall -Wstrict-prototypes -ansi -pedantic

and make sure that your program doesn't generate any warnings. I will not accept code that generates any compiler warnings.

* **[MAGIC\_NUMBER]**

Avoid putting a large number into a file which has no obvious relevance to the surrounding code. This is known as a "magic number" and is often found when setting the size of arrays, *e.g.*:

int my\_array[4096]; /\* 4096 is a magic number \*/

The reason for avoiding this is twofold:

* + It's not usually clear from the context what the significance of the number is.
  + The same number tends to occur several times in the file, which causes problems when you want to change the value.

The right thing to do is this:

#define BUFSIZE 4096 /\* size of buffer \*/

...

int my\_array[BUFSIZE];

Alternatively, it's perfectly valid to declare a constant:

const int BUFSIZE = 4096; /\* size of buffer \*/

...

int my\_array[BUFSIZE];

* **[USELESS\_CODE]**

Don't put in code that has no function or no effect. If it's code that's was only used for debugging purposes, it should be removed before you submit your lab.

* **[USAGE\_STMT]**

If a program is called with incorrect arguments, it should detect that and print a usage statement to the terminal. The usage statement should include the program name. The easiest way to do that is to use **argv[0]** *i.e.*

char usage[] = "usage: %s input\_filename output\_filename\n";

if (argc != 2)

{

fprintf(stderr, usage, argv[0]);

exit(1);

}

Note that the arguments have mnemonic names. Don't write the usage message multiple times. If necessary, you can define a usage function:

void usage(char \*progname)

{

fprintf(stderr, "usage: %s input\_filename output\_filename\n", progname);

}

and then call it like this:

int main(int argc, char \*\*argv)

{

/\* code omitted \*/

if (/\* arguments are incorrect \*/)

{

usage(argv[0]);

return 1;

}

/\* more code omitted \*/

return 0;

}

Alternatively, you could put a call to the exit() function in the usage() function:

void usage(char \*progname)

{

fprintf(stderr, "usage: %s input\_filename output\_filename\n", progname);

exit(1);

}

and then call it like this:

#include /\* declaration of exit() function \*/

int main(int argc, char \*\*argv)

{

/\* code omitted \*/

if (/\* arguments are incorrect \*/)

{

usage(argv[0]); /\* no return needed \*/

}

/\* more code omitted \*/

return 0;

}

In this example, the return 1; line wasn't needed because when exit(1); is called from the usage() function the program will exit with a return value of 1.

Also, make sure that you use fprintf and print to stderr (the error output stream) instead of using printf, which prints to stdout (the normal output stream).

As a general rule, **any** error that involves the user supplying invalid command-line arguments should give rise to a usage statement like the ones described above. You should try to make your usage statements comprehensive enough so that one statement will work for all such errors.

For more on the correct format of usage statements, see [this page](http://courses.cms.caltech.edu/cs11/material/c/mike/misc/cmdline_args.html).

* **[STMTS\_ON\_LINE]**

Never put more than one statement on a line. It makes for unreadable code.

* **[PRECEDENCE]**

Use parentheses to show operator precedence in all cases except that of multiplication/division over addition/subtraction and assignment statements.

* **[EMPTY\_LINES]**

Do not put large numbers of empty lines (> 2) between code sections unless there is a clear need to distinguish different sections of the code. Conversely, **do** put an empty line between logical sections in a single function. An example of this is between the type declarations and the first line of actual code. Another example is at the end of a block in curly braces (though this is a judgment call). Long functions that have no blank lines in them are really hard to read.

* **[BLOCK\_CURLY\_BRACES]**

Use curly braces for the body of all if statements, even if the body is only a single statement. Do the same for else, else if, for, and while statements. The reason for this is twofold: first, it makes the code more readable, and second, it makes it easier to add printf statements for debugging in the body of the expression (which you will frequently have to do).

* **[MATCH\_CURLY\_BRACES]**

If you are using a formatting style where the curly braces of a block are on a separate line (which I encourage), make sure that the column of the curly braces match *e.g.* do this:

for (i = 0; i < n; i++)

{

/\* code goes here \*/

}

instead of:

for (i = 0; i < n; i++)

{

/\* code goes here \*/

} /\* braces don't line up \*/

* **[CODE\_ON\_CURLY\_BRACE\_LINE]**

Don't put code on the same line as an open curly brace. For instance, this is bad:

if (a != 0)

{ a = b + c; /\* ugly \*/

printf("a is now: %d\n");

}

Keep the curly braces on their own lines; this makes the code easier to read. Unfortunately, you often see code written like that in books about programming; the reason is that they have to cram as much code as possible onto a single page. You don't. Instead, write this as:

if (a != 0)

{

a = b + c;

printf("a is now: %d\n");

}

* **[BLOCK\_ON\_SINGLE\_LINE]**

Do not put an entire block on a single line, and most especially do not put it on the same lines as an **if**, **while**, **for** etc. *E.g.* change

if (i < 10) { break; }

to

if (i < 10)

{

break;

}

* **[NO\_INDENTING]**

Lines within a block should be indented relative to lines outside a block.

* **[INCONSISTENT\_INDENTING]**

Lines at the same level of a block should start at the same column.

* **[FOR\_LOOP\_COMPUTATIONS]**

Do not try to do complex calculations in the testing or increment parts of for loops. Don't try to impress everyone with how clever you are; clever code is a maintenance disaster.

* **[VARIABLE\_NAMES]**

Make variable names descriptive as much as possible; avoid one or two character names unless it's for something trivial like a loop index. It's perfectly OK (and usually desirable) to have longer descriptive names for variables. When you do this with names that are actually multiple words, use one of two conventions:

* + the underscore convention: long\_variable\_name
  + the capwords convention: longVariableName

Either convention is OK as long as you're consistent. I (Mike Vanier) prefer the underscore convention, but that's just personal preference.

* **[IMPLICIT\_CONVERSIONS]**

Avoid using implicit int-to-float or int-to-double conversions (or vice-versa) as much as possible. It's hard to keep track of the types of the results otherwise, and C compilers tend not to be very strict about this, which often leads to unexpectedly wrong results. Instead, use explicit type casts when you want to convert an int to a double etc. For instance, this:

int a = 10;

double b;

b = a; /\* implicit conversion \*/

should be written as:

int a = 10;

double b;

b = (double)a; /\* explicit conversion \*/

Yes, it's a bit more verbose, but it's absolutely unambiguous.

**Comments**

* **[COMMENTS\_FULL\_SENTENCES]**

This is the **single most common style mistake**. If a comment is a full sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!), and it should end in a period. I prefer comments that are complete sentences. You should use two spaces after a sentence-ending period.

Bad:

/\* go through the loop and make sure that all the array elements

\* have been set to zero \*/

Good:

/\*

\* Go through the loop and make sure that all the array elements

\* have been set to zero.

\*/

That wasn't so hard, was it?

When you need to refer to identifiers, put them in surrounding single quotes, *e.g.*

/\* The variable 'nitems' represents the number of items in the stack. \*/

If a comment is very short, it doesn't have to be a full sentence or end in a period *e.g.*

i = 1; /\* loop index \*/

This is called an "inline comment". Use these only when describing something *i.e.* in the above code snippet you're saying "The variable 'i' represents a loop index."

* **[COMMENT\_GRAMMATICAL]**

Comments should be grammatically correct. In particular, incorrect spelling is unacceptable. I hate to sound like your high school English teacher, but it's a pain to read code with tons of spelling mistakes. Use a spell checker if you have to.

* **[COMMENT\_SPACE]**

Put a space after the open-comment symbol and before the close-comment symbol *i.e.* do this:

/\* This is a comment that is easy to read. \*/

and not this:

/\*This is a comment that is harder to read.\*/

* **[COMMENT\_C++]**

Do not use C++ style comments *i.e.* comments that start with // and go to the end of the line. It is true that most C compilers (including gcc) accept them, and they will be part of the C standard soon. But for now, it's a non-portable feature.

* **[COMMENT\_MULTI\_LINE]**

Use this style for multi-line comments:

/\*

\* This is a multi-line comment.

\* Spiffy, isn't it?

\*/

Most especially, do **not** use this style:

/\* This is a bad way to write multi-line comments. \*/

/\* You comment out every line individually. \*/

/\* Ugly, isn't it? \*/

People who write comments this way may not be aware of the fact that comments can span more than one line. Well, they can, so take advantage of it.

* **[COMMENT\_BLOCK]**

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a \* and a single space (unless it is indented text inside the comment). Paragraphs inside a block comment are separated by a line containing a single \*. Block comments are best surrounded by a blank line above and below them (or two lines above and a single line below for a block comment at the start of a a new section or function definition). I prefer to start and end a block comment with a line containing a single \*. In other words, a block comment looks like this:

/\*

\* The first line comes after an empty line.

\*

\* Separate paragraphs are also separated by an empty line,

\* and there's an empty line at the end.

\*

\*/

* **[COMMENT\_NON\_OBVIOUS]**

Write comments for anything that isn't completely obvious from the context. In particular, write comments for any tricky algorithm or code you are using. When in doubt, comment more rather than less.

* **[COMMENT\_REDUNDANT]**

Conversely, don't make completely redundant comments, *e.g.*

i = 1; /\* Set i to 1. \*/

What constitutes redundancy is often a judgment call. If in doubt, comment more rather than less.

* **[COMMENT\_MEANINGLESS]**

Don't make meaningless comments *e.g.*

/\* i \*/

i = 1;

Don't laugh; I've actually seen this sort of thing.

* **[COMMENT\_HEADER]**

You should almost always put a comment at the beginning of each function describing what it does. The only exception is when you have a series of very similar functions which are written out one after another, and where the first comment applies (suitably modified) to all of them. This kind of "header comments" (not to be confused with header files) are by far the most important kind of comments, because even if the person reading your code has no idea how a given function works, the header comment will at least tell him what it does and how to use it. You should state what each of the arguments represents and what the function returns. You may also want to describe the algorithm used, its efficiency, and any other relevant factoids. An example:

/\*

\* bubble\_sort:

\* This function takes an array and sorts it in-place using the bubble

\* sort algorithm. This algorithm has a time complexity of O(n^2)

\* where 'n' is the size of the array, which is not very efficient.

\* Therefore, for large arrays use a more efficient algorithm such as

\* quicksort.

\*

\* Arguments:

\* -- arr: the array to be sorted

\* -- size: the length of the array to be sorted

\*

\* Return value: none.

\*

\*/

void bubble\_sort(int arr[], int size)

{

/\* code \*/

}

* **[COMMENTS\_CONSISTENT\_WITH\_CODE]**

Comments that contradict the code are worse than no comments. ALWAYS MAKE A PRIORITY OF KEEPING THE COMMENTS UP-TO-DATE WHEN THE CODE CHANGES!

* **[COMMENT\_INDENT]**

Always indent your comments to the same degree as the surrounding code.

* **[COMMENT\_INLINE]**

Try to line up inline comments where convenient. In other words, don't do this:

x = x + 1 /\* some cool comment about x \*/

y = y + 1 /\* some even cooler comment about y \*/

Instead, do this:

x = x + 1 /\* some cool comment about x \*/

y = y + 1 /\* some even cooler comment about y \*/

Some people like to line up the close-comment token as well. Use your own judgment.

* **[COMMENT\_PRECEDING]**

Do not write comments that apply to the preceding code if you can possibly avoid it. Try to write comments that refer to the current line of code or to the lines of code which immediately follow. For instance, this is bad:

int res;

/\*

\* 'res' contains the result of the program. It will normally be 0,

\* unless an error occurs, in which case it will be 1.

\*/

and this is good:

/\*

\* 'res' contains the result of the program. It will normally be 0,

\* unless an error occurs, in which case it will be 1.

\*/

int res;

This is also bad:

int res; /\* 'res' contains the result of the program.

\* It will normally be 0, unless an error occurs,

\* in which case it will be 1.

\*/

I hope this is obvious, but I see it in students' submissions all the time. It's OK to put a comment on the same line as the code, but only if the entire comment will fit on that line *e.g.*:

int res; /\* 'res' contains the result of the program. \*/

If what you have to say won't fit on the line, put the comment on the lines above the line of code. (Don't use a very long line to try to fit the comment on one line, of course; see LINE\_LENGTH above).

**Functions**

* **[FUNCTION\_PROTOTYPES]**

Always write function prototypes at the top of a file for every function whose definition is in that file. This is not only good documentation, it enables you to use these functions anywhere in the file without having to worry about putting them in strict order of definition. In case you don't know already, this is a function prototype:

int foo(double bar, char \* baz);

It's just a function without a body.

* **[FUNCTION\_BLANK\_LINES]**

Please separate your function definitions by at least two blank lines. Otherwise it's hard to find where a function definition begins.

* **[FUNCTION\_DECOMPOSITION]**

Don't hesitate to decompose your functions into lots of smaller functions. Many people seem to think that calling a function is an incredibly expensive operation. That was true for some languages and some compilers a long time ago, but now almost any C compiler (at least) can "inline" little functions so that there is no calling overhead. Some compilers, such as gcc, even allow you to declare functions as inline, although this isn't in the C standard yet (but it soon will be). I've seen and worked on code that had single functions that were twenty or more pages long, with no comments. Don't do it!

* **[FUNCTION\_STARTING\_COLUMN]**

Start the line that begins a function in column 0 (the leftmost column).

**Finally...**

Don't worry if you can't remember all of these rules; I don't expect you to. At this point it's more important that you develop an intuition for what is good and what is bad style, and if you aren't sure, you can refer back to this page later.