

# COMP2303/COMP7306

## C Programming Style Guide

### Version 1.4

Programs written for COMP2303/7306 assignments must follow the style guidelines outlined in this document.

#### Naming Conventions

Variable names and file names will begin with a lowercase letter and names with multiple words will use initial capitals for subsequent words. Names that are chosen should be meaningful and Hungarian notation is NOT to be used. Source files must be named with the suffix `.c` (lower case).

Example variable names: `book`, `newCount`, `setWordLength`

Example filenames: `hello.c`, `stringRoutines.c`

`#define`'d constants are in all uppercase letters and underscores (`_`) are used to separate multiple words.

Examples: `MAX_BIT`, `DEFAULT_SPEED`

Function names should all be lowercase and use underscores to separate multiple words.

Example function names: `main()` `reset_secret_string()`

Type names (i.e. from typedefs), structure and union names should begin with capital letters and use initial capitals for subsequent words.

Example typedef/structure/union names: `SortType`, `FileData`

#### Comments

Comments should be generously added to describe the code. The comments should explain what is happening, how it is being done, what parameters mean, which global variables are used and which are modified, and any restrictions or bugs. Comments must be present and meaningful for at least every global variable declaration and function declaration or definition and other comments are expected in code which is tricky, lengthy or where functionality is not immediately obvious. (If an adequate comment is given for a function declaration, it need not be repeated for the associated function definition.)

No comment is needed for the `main()` function.

Examples:

```
float bookPrice; /* The price of the book, user input */
int sum_function (int a, int b); /* Returns the sum of a and b */
```

You may **not** use C++ style comments (i.e. //).

## Braces {}

Use braces around statements in the body of if, else, for, while, do, etc statements. The C language does not require braces when the body contains only one statement, but you **must** surround it with braces anyway. This helps avoid errors while changing your code. An open-brace appears at the end of the line after function arguments/loop statement/etc (with at least one space before the brace). The closing brace should be lined up underneath the start of the function/loop statement/etc. Braces for structure declarations and array initialisations do not need to follow this layout.

The opening brace for a **function definition** may be either at the end of the line following the arguments or in the left most column of the next line.

Examples:

```
/*
** sum(int n)
**   - returns the integer sum of integers
**     1 through n. Will return 0 if n is 0
**     or less.
**     (Note: won't work if n is large enough to cause the sum to overflow)
*/
int sum (int n) { /* This brace can be at the start of next line if desired */
    int i, s = 0;

    for (i = 1; i <= n; i++) { /* This brace must be here */
        s += i;
    }
    return s;
}
```

while(...) { ... }	for(..., ..., ...) { ... }	do { ... } while(...);	if(...) { ...; }	if(...) { ...; } else { ...; }	switch(...) { case ...: ...; ... }
--------------------------	----------------------------------	------------------------------	------------------------	--	--

## Whitespace

Meaningful parts of code are grouped together by using the whitespace as a separator. Whitespace is composed of horizontal whitespace (i.e. space and tab characters) and vertical whitespace (i.e. blank lines).

### **Vertical whitespace**

Organize your source code into meaningful parts. For example, use blank lines to separate functions from each other. Blank lines are also used to separate groups of statements from each other to make the major steps of an algorithm distinguishable.

## Horizontal whitespace

Use horizontal whitespace to organize each line of code into meaningful parts. It is bad style to not use spaces within a line. Space must be added after each comma and semicolon and either side of assignment operators (=, += etc). Other spaces (e.g. around operators such as < and + or before parentheses) can be added at your discretion.

Examples:

good style	bad style
<code>a = (b + c) * d; a = (b+c)*d;</code>	<code>a=(b+c)*d;</code>
<code>add_up(a, b, c, d);</code>	<code>add_up(a,b,c,d);</code>
<code>for(i = 0; i&lt;10; i++)</code>	<code>for(i=0;i&lt;10;i++)</code>

## Indentation

Use indentations to indicate the level of nesting. Indentation must occur in multiples of four spaces. You should configure your editor so that indents are in multiples of four spaces (but tab stops must remain at 8 characters)<sup>1</sup>. You should indent once each time a statement is nested inside the body of another statement. Always indent whether or not the control structure uses braces.

Examples:

<b>if &amp; for loop</b>	<pre>if (day == 31) {     monthTotal = 0;     for (week = 0; week &lt; 4; ++week) {         monthTotal += receipts[week];     } }</pre>
<b>chained if-elses</b>	<pre>if (month &gt;= 1 &amp;&amp; month &lt;= 3) {     quarter = 1; } else if (month &gt;= 4 &amp;&amp; month &lt;= 6) {     quarter = 2; } else if (month &gt;= 7 &amp;&amp; month &lt;= 9) {     quarter = 3; } else {     quarter = 4; }</pre>
<b>switch form</b>	<pre>switch (month) {     case 2:         daysInMonth = 28;         break;     case 4:     case 6:     case 9:     case 11:         daysInMonth = 30;         break;     default:         daysInMonth = 31;         break; }</pre>

<sup>1</sup> If your editor doesn't support this, it is suggested that you change to one that does (e.g. vi or emacs).

## **Line Length**

Lines must not exceed 79 columns in length including whitespace. The "standard" screen size is 80 columns. Occasionally an expression will not fit in the available space in a line; for example, a function call with many arguments, or a logical expression with many conditions. Such occurrences are especially likely when blocks are nested deeply or long identifiers are used. If a long line needs to be broken up, you need to take care that the continuation is clearly shown. For example, the expression could be broken after a comma in a function call (ideally never in the middle of a parameter expression), or after the last operator that fits on the line. The following continuation must be double indented (8 spaces) so that it is clearly identifiable. If more than one continuation line is required, no further indenting is required.

Examples:

```
someFunction(longExpression1, longExpression2, ...,
             longExpressionN);

if(expressionA || expressionB || expressionC || expressionD ||
    expressionE || expressionF) {
    /* code goes here - indented by 4 */
    ...
}

if(expressionA || expressionB || expressionC || expressionD ||
    expressionE || expressionF || expressionG || expressionH ||
    expressionI) {
    /* code goes here - indented by 4 */
    ...
}
```

## **Overall**

Code must conform to the 1989 ANSI C standard (C89). You should compile your code with the `-Wall` and `-ansi` flags to `gcc`. No errors or warnings should be reported.

In general, functions should not exceed one page of statements (when printed in a reasonable font, i.e. about 50 lines maximum). If a function is longer than this, then it is a good candidate for being broken into meaningful smaller functions. In exceptional circumstances (which should be documented with a comment), functions may be longer than a page.

Principles of modularity should be observed. Related functions and variable definitions should be separated out into their own source files (with appropriate header files for inclusion in other modules as necessary). You should also use functions to prevent excessive duplication of code. If you find yourself writing the same or very similar code in multiple places, you should create a separate function to undertake the task.