CSSE2310/CSSE7231
C Programming Style Guide
Version 2.3.0

The University of Queensland
School of Information Technology and Electrical Engineering

February 2023

Programs written for CSSE2310/7231 assignments must follow the style guidelines outlined in this document.

Where the style guide is open to interpretation, the choices you make regarding your personal style must be consistent throughout your project.

1 Naming Conventions

1.1 Variable names

Variable names and the names of struct/union members will begin with a lowercase letter and names with multiple words will use initial capitals for subsequent words and no underscores – sometimes called camel case[1]. Names that are chosen should be meaningful. Hungarian notation[2] is NOT to be used. It is permissible to use names like `i` or `j` for integer loop counters.

Example variable names: `book`, `newCount`, `setWordLength`, `dictionaryLength`, `wordList`, `cursor`, `fileHandle`

Unacceptable variable names (use of Hungarian notation): `int iCount; int* ptrNum; double dAverage;`

It is permissible to use variable names like `str` to indicate a generic string if, for instance, a function does take a generic string parameter. (This is permissible even though the name indicates the type.) However, if the string variable always has a particular meaning in the context of your program/function then a more appropriate variable name should be used, e.g. `name` or `fileName`.

1.2 File names

C source file names will begin with a lowercase letter and names with multiple words will use initial capitals for subsequent words and no underscores. Names that are chosen should be meaningful. Source files must be named with the suffix `.c` or `.h` (lower case).

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

1.3 Defined constants and macros

Preprocessor constants and macros, i.e. those defined using `#define`, must be named using all uppercase letters, with underscores (_) used to separate multiple words. NOTE: Variables declared with the `const` keyword are to use variable naming, as per 1.1.

Examples: `MAX_BIT`, `DEFAULT_SPEED`

1.4 Function names

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

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

Note that function pointers are variables and should use variable naming, as per 1.1.

[1]This variant of camel case in which the first letter is lower case is sometimes called lower camel case or dromedary case.

[2]Hungarian notation is where characters at the start of the name indicate the type of the variable. Examples include `int iCount; char* pFileName;` and `double dbPi`.

Example: `int (*sortingFunction)(void*, void*) = alpha_sort;`
where `sortingFunction` is the name of a variable and `alpha_sort` is the name of a function.

Uppercase letters and numbers may be used in function names where that usage is conventional (e.g. use of a standard abbreviation) and the function name begins with a lowercase letter.

Example function names: `calculate_GPA()`, `check_W3C_compliance()`

1.5 Type names

Type names (i.e. from typedefs), structure and union names should begin with capital letters and use initial capitals for subsequent words and no underscores. Members within structs and unions should follow the variable naming convention as per section 1.1.

Example typedef/structure/union names: `Contact`, `FileData`, `struct Node`

```c
/* A player within the game */
struct Player {
    char *name;
    int score;
    struct Player *next;
};
```

Example 1: Struct naming and declaration

```c
/* Point (coordinate) */
typedef struct {
    int x;
    int y;
} Point;
```

Example 2: Typedef struct naming and declaration

1.6 Enumerated Types

Enumerated types must be named in the same way as any other type (see 1.5). Members of an enumerated type must follow the same naming as constants (see 1.3).

```c
/* Card suits */
enum CardSuits {
    CLUBS, DIAMONDS, HEARTS, SPADES
};
```

Example 3: Enumerated type naming and horizontal layout

```c
// Program exit codes
enum ExitCodes {
    EXIT_SUCCESS = 0,
    EXIT_ARGS = 1,
    EXIT_FAILURE = 2
};
```

Example 4: Enumerated type naming and vertical layout, with explicit value selection

2 Comments

Comments should be generously added to describe the intent of the code. They are expected in code which is tricky, lengthy or where functionality is not immediately obvious. It is reasonable to assume that the reader has a decent knowledge of the C programming language, so it is not necessary to comment every line within a function.

Comments must be present and meaningful for each of the following:

- Global variables[3]
- Function declarations or definitions
- Enum definitions
- Struct definitions

Comments can use either `/* ... */` or `//` notation.

Comments must not be used to comment out unused code. Use conditional compilation to prevent a block of code being compiled, e.g. `#if 0 ... #endif`, or remove the code completely. Any use of conditional compilation in this way must be accompanied by a comment that explains the reason for its use.

[3]See the note about global variables in section 7

2.1 Function comments

Function comments should describe parameters, return values and any error conditions. ("Error conditions" means the conditions under which the return value indicates an error and/or the conditions under which the function does not return, i.e. the program exits.) Function comments do not need to include the *types* of parameters (these are apparent from the function prototype/definition), nor should they repeat the function prototype. If an adequate comment is given for a function declaration (prototype), it need not be repeated for the associated function definition, even if these are in different files. If a declaration (prototype) and definition are in the same file then either may be commented – but your approach should be consistent, i.e. either comment all prototypes or all definitions – not half-half. If the declaration and definition are in different files (e.g. the declaration is in a `.h` file and the definition is in a `.c` file) then there must be a comment associated with the declaration in the `.h` file. There is no need to have a comment with the definition, but if you do, it must be consistent with the comment associated with the declaration. No comment is needed for the `main()` function.

The presence of a function comment **does not remove the need for comments within a function**. It is likely that only a very short function would have no comments within the body of the function. Any function of reasonable length will almost certainly need comments within the function to describe the intent of the code.

The following is the recommended comment format to ensure that your function comment meets the requirements above.

```c
/* function_name()
 * ---------------
 * Description of what the function does in terms of the parameters goes here...
 *
 * arg1: description of what this argument contains (and, if applicable,
 *           any assumptions about the value, e.g. not null, string length not 0, etc.)
 * argument2: ... (repeat for all arguments)
 *
 * Returns: description of what is returned
 * Errors: (if applicable) description of errors that might occur and what happens if they do, e.g.
 *     the return value will be invalid, or the program exits (i.e. function does not return).
 * REF: (if applicable) if this whole function is copied/inspired then you can reference the source here
 */
int function_name(char* arg1, int argument2) { ...
```

Example 5: Recommended function comment template

See examples 8, 9, 10, 11 and 18 below for examples of function comments.

Comments do not need to follow this format exactly – this is a suggested template. The key thing is that each function comment must describe the behaviour of the function in terms of the parameters, must describe what is returned (may be obvious as part of the behaviour description), and must describe any error conditions and what happens when those errors occur (e.g. certain value returned or program exits or ...).

2.2 Code references

If you are required (by the assignment specification) to include a code reference (for example – but not limited to – you are using code from a man page, or using non-public code you have written previously, or you have been inspired by some third party source), then a reference comment must be provided **adjacent** to the code itself (i.e. not in a header file or in a comment at the top of the file). If the reference applies to a whole function then you may include the reference in the function comment, provided the function comment is with the definition, not the declaration[4]. All reference lines in a comment (i.e. the first and continuation lines) must contain the text "`REF:`" (in uppercase, without the quotes) to indicate that the line contains reference information. This pattern will be searched for in your code, so it is important that match this exactly. See the examples below:

```c
// REF: The following block of code is taken from code submitted by me for the
// REF: CSSE2010 AVR programming assignment in semester two, 2022.
```

Example 6: Example code reference – code you have previously written

```c
// REF: The following code is inspired by the code at
// REF: https://stackoverflow.com/questions/14685406
```

Example 7: Example code reference – code you have been inspired by

[4]See "Function comments" above for the required location of function comments

```c
/* string_comparator()
 * -------------------
 * qsort() comparator function that compares two strings.
 *
 * str1, str2: pointers to strings (char*) that we're comparing.
 *
 * Returns: an integer less then, equal to, or greater than zero if the string pointed to by
 *    str1 is less than, equal to or greater than the string pointed to by str2
 * REF: This function is taken from the qsort(3) man page. The name of the function
 * REF: and spacing have been modified to comply with the style guide.
 */
static int string_comparator(const void *str1, const void *str2)
{
    // Compare the strings. Note that we must cast the void* pointers
    // to pointers to strings, and then dereference them
    return strcmp(*(char* const*)str1, *(char* const*)str2);
}
```

Example 8: Example code reference – in a function comment

```c
/* absolute_value()
 * ----------------
 * Function returns the absolute value of its argument (x).
 *
 * REF: This function was generated by ChatGPT.
 * REF: The function name was then modified to comply with the style guide.
 */
int absolute_value(int x) {
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
}
```

Example 9: Example code reference – in a function comment – for AI generated code

```c
/* abs()
 * -----
 * Function returns the absolute value of a number (n).
 *
 * REF: This function and most of the description above was generated by Github CoPilot.
 * REF: The indentation was then modified to comply with the style guide.
 */
int abs(int n) {
    if (n < 0) {
        return -n;
    }
    return n;
}
```

Example 10: Example code reference – in a function comment – for AI generated code (2)

Note that the examples above show how to reference code from various sources. Just because code is referenced does not necessarily mean that you can use it. See your assignment specification for details of the code that can be used in that particular assignment. For code generated by, or modified with input from, AI tools, you must document your interaction with the tool as described in the *Documentation required for the use of AI tools* document.

3 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 an if statement/loop statement/etc (with exactly one space before the brace). The closing brace should be lined up underneath the start of the function/loop statement/etc. Braces for structure/union/enum 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.

```
/* sum()
 * ---------------
 * Calculates the sum of integers from 1 to n.
 *
 * n: the maximum integer to be summed
 *
 * Returns: the integer sum of integers 1 to n,
 *     or 0 if n is 0 or less
 * Errors: will return an erroneous value if n
 *     is large enough to cause the sum to
 *     overflow
 */
int sum(int n) { // This brace can be at the
                 // start of the next line
    int i, s = 0;

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

    return s;
}
```

Example 11: Correct braces for a function

```
for (...; ...; ...) {
    ...
}
```

Example 12: Correct braces: for loop

```
while (...) {
    ...
}
```

Example 13: Correct braces: while loop

```
do {
    ...
} while (...);
```

Example 14: Correct braces: do while loop

```
if (...) {
    ...
}
```

Example 15: Correct braces: if statement

```
if (...) {
    ...
} else if (...) {
    ...
} else {
    ...
}
```

Example 16: Correct braces: if-else

```
switch (...) {
    case ...:
        ...
        break;
    default:
        ...
}
```

Example 17: Correct braces: switch statement

4 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).

4.1 Vertical whitespace

Organize your source code into meaningful parts. You must use single 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.

```
/* multiply()
 * ---------------
 * Multiplies two integers.
 *
 * x, y: the numbers to be multiplied
 *
 * Returns: the product of x and y
 * Errors: The return value will not be meaningful if the product can't be represented in an
 *     integer, i.e. overflows the bits available.
 */
int multiply(int x, int y) {
    return x * y;
```

```c
}

/* divide()
 * ---------------
 * Divides one integer by another. (The remainder is discarded.)
 *
 * x: the dividend
 * y: the divisor - assumed not to be zero
 *
 * Returns: x divided by y
 * Errors: Behaviour is undefined if y is zero
 */
int divide(int x, int y) {
    return x / y;
}
```

Example 18: Acceptable vertical whitespace between functions

4.2 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. A single space must be added after each comma, as well as each semicolon that is not on the end of a line. A single space should also be added either side of all assignment and binary operators (`= += / - * +` etc). There must be no spaces added around unary operators (`& * + - ~ ! ++ --`) and struct operators (`point->x`, `player.name`). There must be a single space present before an open parenthesis in C control statements, e.g., a space between these keywords and the following parenthesis: `for`, `while`, `if`, `switch`. There must be no spaces before an open parenthesis in function calls, declarations and definitions. There must be no spaces prior to square brackets in array derefencing. There must be no spaces prior to semicolons at the end of a line, but a semicolon can be on a line by itself if required.

When declaring a pointer there must be a single space on one side of the `*`. It can be either side but you must be consistent in your approach.

Acceptable horizontal whitespace

```c
add_up(a, b, c, d);
int *a, b, c[10];
```

Example 19: Acceptable horizontal whitespace: commas

```c
for (i = 0; i < 10; i++)
```

Example 20: Acceptable horizontal whitespace: semi-colons

```c
a = (b + c) * d;
```

Example 21: Acceptable horizontal whitespace: assignment and binary operators

```c
int x = -y;
b = *c;
width = point->y;
```

Example 22: Acceptable horizontal whitespace: unary and struct operators

```c
char* a;
char *b;
//Either style can be used, but you
//must be consistent in your program
```

Example 23: Acceptable pointer declarations

Unacceptable horizontal whitespace

```c
add_up(a,b,c,d);
int * a,b,c[10];
```

Example 24: Unacceptable horizontal whitespace: commas

```c
for (i=0;i<10;i++)
```

Example 25: Unacceptable horizontal whitespace: semi-colons

```c
a=(b+c)*d;
```

Example 26: Unacceptable horizontal whitespace: assignment and binary operators

```c
int x = - y;
b = * c;
width = point -> y;
```

Example 27: Unacceptable horizontal whitespace: unary and struct operators

```c
char*a;
char * b;
```

Example 28: Unacceptable pointer declarations

5 Indentation

Use indentation 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)[5]. 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.

```c
if (day == 31) {
    monthTotal = 0;
    for (week = 0; week < 4; ++week) {
        monthTotal += receipts[week];
    }
}
```

Example 29: Correct indentation: if statement

```c
if (month >= 1 && month <= 3) {
    quarter = 1;
} else if (month >= 4 && month <= 6) {
    quarter = 2;
} else if (month >= 7 && month <= 9) {
    quarter = 3;
} else {
    quarter = 4;
}
```

Example 30: Correct indentation: chained if-else

```c
switch (month) {
    case 2:
        daysInMonth = 28;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        daysInMonth = 30;
        break;
    default:
        daysInMonth = 31;
        break;
}
```

Example 31: Correct indentation: switch statement

If you use the gnu `indent` tool consider the following options as a starting point:
`-linux -i4 -cli4 -l79 --no-tabs`

6 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 continuation line 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.

[5]In other words, do not change your tab stops to 4 characters. If tab characters are present in your code then they indicate an indent to the next multiple of 8. You can replace all tabs by spaces if you wish, but this is not required.

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

if (expressionA || expressionB || expressionC || expressionD ||
        expressionE || expressionF) {
    /* code goes here - indented by 4 spaces */
    ...
    printf("A very long string can be broken like "
            "this");

}

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

Example 32: Correct line length and indentation

7 Overall

Source code should be written to be clear and readable. Anything which works against your code being clear and readable may be penalised. For example (but not restricted to):

- Global and static variables must not be used unless there is no other way to implement the required functionality, for example, they can be used with signal handlers. (Global constants, declared using `const` are permitted.)
- "Magic numbers" are to be avoided. Defined constants should be used instead of number and character literals scattered throughout code without context. 0 and 1 are almost never magic numbers.

```c
for (int i = 1; i <= 26; i++) {
    ...
}
if (name[0] == '\n') {
    ...
}
if (strlen(passwd) < 8) {
    ...
}
```

Example 33: Unacceptable use of magic numbers

```c
#define NUM_LETTERS_IN_ALPHABET 26
#define NEWLINE '\n'
const int minPasswordLength = 8;
for (int i = 1; i <= NUM_LETTERS_IN_ALPHABET; i++) {
    ...
}
if (name[0] == NEWLINE) {
    ...
}
if (strlen(passwd) < minPasswordLength) {
    ....
}
```

Example 34: Use of constants to avoid magic numbers

7.1 Compilation

Your assignments will use the C99 standard, and must be compiled with the following flags as a minimum: `-Wall -pedantic -std=gnu99` . No warnings or errors should be reported. If warnings or errors are reported, they will be treated as style violations.

Every source file in your assignment submission (including .c files and any custom .h files they `#include`) will be checked for style. This applies whether or not they are linked into executables.

Every .h file in your submission must make sense without reference to any other files, other than those which it references by `#include`. Specifically, any declarations must be complete and all types, declarations and any definitions used in the .h file must either come from the .h file itself, or from included headers.

Any components which are obfuscated, not in standard forms or need to be transformed by extra tools in order to be readable will be removed before compilation is attempted.

7.2 File encoding

All source files must use an ASCII-compatible encoding. Files must use Unix line endings (`\n`).

7.3 Function length

Functions should not exceed 50 lines in length, including any comments within the function but excluding any comments prior to the function. If a function is longer than 50 lines, then it is a good candidate for being broken into meaningful smaller functions.

7.4 Modularity

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.

8 Banned Language Features

Use of any of the following C features in your assessments is grounds for a mark of **zero** in that assessment. All of them work against readable and well structured programs. None of these are things you could do by accident.

- Goto: While `goto` is a legal part of the C language, nothing in this course requires its use.
- Digraphs and Trigraphs: This course uses systems which support ASCII or unicode and so have all the required characters.

Other banned features may be listed in your assignment specification.