CSSE2310/CSSE7231
C Programming Style Guide
Version 2.0.2

The University of Queensland
School of Information Technology and Electrical Engineering

September 2014

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 and File names

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` or `.h` (lower case).

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

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

Unacceptable variable names: `FILE *fp; char *aPtr; char *thingStr;`

1.2 Defined Constants

#defined constants are in all uppercase letters, with underscores (_) being used to seperate multiple words. (The exceptions are defines `true` and `false` to comply with c99).

NOTE: Variables declared with the `const` keyword are to use variable naming, as per 1.1.

Examples: `MAX_BIT`, `DEFAULT_SPEED`

1.3 Function names

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

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

1.4 Type names

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: `Contact`, `FileData`, `struct Node`

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

Example 1: Struct naming and declaration

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

Example 2: Typedef struct naming and declaration

1.5 Enumerated Types

Enumerated types must be named in the same way as any other type (see 1.4). Members of an enumerated type must follow the same naming as constants (see 1.2).

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

Example 3: Enumerated type naming and horizontal layout

```
/* 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. Comments must be present and meaningful for at least every global variable declaration and function declaration or definition. Other comments 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.

Function comments should describe parameters, return values and error conditions. They should not include parameter types, nor should they include the function prototype. 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.

You may use `//` comments in c99 assignments.

```
/* Global Variables */
/* The map to use for the game */
char **map;
/* Width of the map */
int width;

/* Prototypes (NOTE: function comments are not required here.) */
/* Returns the sum of a and b */
int sum_function(int a, int b);
```

Example 5: Comments for global variables and function prototypes

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 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.

```
/**
 * Calculates the integer sum of integers 1 through n.
 * Returns this sum, or 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 the next line */
    int i, s = 0;

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

    return s;
}
```

Example 6: Correct braces for a function

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

Example 7: Correct braces: for loop

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

Example 8: Correct braces: while loop

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

Example 9: Correct braces: do while loop

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

Example 10: Correct braces: if statement

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

Example 11: Correct braces: if-else

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

Example 12: 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. 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.

```
/* Multiply two numbers */
int multiply(int x, int y) {
    return x * y;
}

/* Divide two numbers */
int divide(int x, int y) {
    return x / y;
}
```

Example 13: 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. Space must be added after each comma, as well as each semicolon that is not on the end of a line. Space should also be added either side of all assignment and binary operators (`= += / - * +` etc). There should be no spaces added around unary operators (`& * + - ~ ! ++ --`) and struct operators (`point->x`, `player.name`).

Acceptable horizontal whitespace

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

Example 14: Acceptable horizontal whitespace: commas

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

Example 15: Acceptable horizontal whitespace: semi-colons

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

Example 16: Acceptable horizontal whitespace: assignment and binary operators

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

Example 17: Acceptable horizontal whitespace: unary and struct operators

Unacceptable horizontal whitespace

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

Example 18: Unacceptable horizontal whitespace: commas

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

Example 19: Unacceptable horizontal whitespace: semi-colons

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

Example 20: Unacceptable horizontal whitespace: assignment and binary operators

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

Example 21: Unacceptable horizontal whitespace: unary and struct operators

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). 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.

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

Example 22: Correct indentation: if statement

```
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 23: Correct indentation: chained if-else

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

Example 24: 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 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.

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

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

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

Example 25: Correct line length and indentation

7 Overall

7.1 Compilation

Your assignments use either "ANSI C" (C90) or C99 (Your lecturer will tell you which one applies).

For ANSI C you must compile with the flags: `-Wall -pedantic -ansi`

For C99 use: `-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 .c file in your assignment submission (and any custom .h files they #include) will be checked for style. This applies whether or not they are linked into executables. To facilitate this process, any .c files in your submission which do not compile individually (ie `gcc -c file.c` +relevant flags) will attract an additional penalty equivalent to 5 compiler warnings per file.

Every .h file in your submission must make sense without reference to any other files other than those which it #includes. 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.

7.2 Function length

Functions should not exceed 50 lines in length, including any comments. If a function is longer than 50 lines, 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 50 lines.

7.3 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.