CSSE2310/CSSE7231
C Programming Style Guide
Version 3.06

The University of Queensland
School of Information Technology and Electrical Engineering

1 August 2024

Programs written for CSSE2310/CSSE7231 assignments must follow the style guidelines outlined in this document. The formatting requirements are loosely based on the `WebKit` style as implemented by `clang-format`. There are also additional requirements beyond this, e.g. naming conventions and function and line length constraints. The tool `clang-tidy` can be used to check many of these requirements. Scripts are available to do this (see below).

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

This style guide applies only to C source files, i.e. `.c` and `.h` files. It does not apply to other files that must be created for CSSE2310/CSSE7231, e.g. `Makefiles`.

0 Icons and Tools

0.1 Icons used in this document

The following icons are used to indicate which tools (or not) can be used to check for compliance with the style guide.
CLANG-FORMAT – indicates that `clang-format` (with the appropriate arguments) can be used to check for and correct compliance with this style requirement (e.g. fixing indentation and spacing).
CLANG-TIDY – indicates that `clang-tidy` (with the appropriate arguments) can be used to check for compliance with this style requirement. Any corrections must be applied manually (e.g. renaming variables to be consistent with the naming requirements).
MANUAL CHECK – indicates that this style requirement must be checked and fixed manually (e.g. making sure a function comment contains the required information).

0.2 Tools

The tool `2310reformat.sh` is available on `moss.labs.eait.uq.edu.au` to correct those elements of style able to be checked and corrected by `clang-format`, i.e. those items indicated by the CLANG-FORMAT icon below. This script will run `clang-format` with the appropriate arguments and reformat the files in-place (if necessary) – i.e. your source files may be modified. Don't run this tool if you don't want this to happen. You must specify the source files that you want reformatted on the command line. You can run `2310reformat.sh *.h *.c` to reformat all of your source files in the current directory.

The tool `2310stylecheck.sh` is available on `moss` to check those elements of style able to be checked with `clang-format` and `clang-tidy` (with the appropriate arguments), i.e. those items indicated by the CLANG-FORMAT and CLANG-TIDY icons below. Details of the lines on which issues are found are reported to standard output.

Unfortunately, the messages output by `clang-format` are not particularly useful – they'll say "`warning: code should be clang-formatted`" for each line that needs to be fixed – so it is probably easiest to run `2310reformat.sh` to fix these issues prior to running `2310stylecheck.sh`.

You can specify particular source files on the command line to `2310stylecheck.sh`, or, if none are specified, then the tool tries to check all of the .c and .h files in the current directory.

This script will be used to check compliance with the style guide during assignment marking. Any issues flagged will cost you marks so it is in your interest to fix them.

1 Indentation

1.1 Tabs vs spaces

CLANG-FORMAT

Use spaces not tab characters to implement indentation.

It is recommended (but not enforced) that tab characters are not used inside string constants either – use the escape sequence `\t` instead if a tab character is required.

1.2 Indent size

CLANG-FORMAT

Indentation must occur in multiples of four spaces. You must indent once each time a statement is nested inside the body of another statement.

```c
if (day == 31) { // the following statements are indented by 4 spaces
    monthTotal = 0;
    for (week = 0; week < 4; ++week) { // following statement indented by 4
        monthTotal += receipts[week];
    }
}
```

Example 1: Correct indentation for if statement and loop

1.3 Comments

CLANG-FORMAT

Comments on a line by themselves (i.e. not following code) must be indented the same as code.

For `//` comments that start after a statement, continuation lines can either be aligned with the `//` or the code.

```c
if (month >= 1 && month <= 3) {
    // comments on a line by themselves must have the same indenting level
    // as code would have
    quarter = 1;
} else if (month >= 4 && month <= 6) {
    quarter = 2; // multi-line comments can be aligned like this - i.e. the
                 // comment continuation lines up with the start of the comment
                 // in the line above
} else if (month >= 7 && month <= 9) {
    quarter = 3; // multi-line comments can also be aligned like this - i.e
    // the continuation is lined up with the start of the previous
    // statement
} else {
    quarter = 4;
    /* The starting position for comments that start with /* is the same as
       for comments that start with //, but
                continuation lines
                        can use any level
                                of indenting */
}
```

Example 2: Correct indentation: chained if-else – also showing comment indentation

1.4 Continuation lines

CLANG-FORMAT

Continuation lines (i.e. where a statement doesn't fit on one line) must be indented eight spaces beyond the first line of the statement.

If a split happens at an operator (`=`, `+`, etc.) then the operator will be on the continuation line. This makes it clearer that the line is a continuation line.

Expressions should be grouped together where possible, e.g. if the expression after an assignment operator will fit on the continuation line then the whole expression will be placed on that line rather than splitting in the middle of the expression itself.

If a statement spills on to a third line, the indentation level of the third (and any subsequent) line will be the same as the second line – i.e. eight spaces beyond that of the first line of the statement – unless this is a nested expression as described in the next section.

```c
// If the expression fits on one line then split it like this:
int physAddress
        = page_table_lookup(vAddress / PAGE_SIZE) + (vAddress % PAGE_SIZE);
// rather than like this:
// int physAddress = page_table_lookup(vAddress / PAGE_SIZE)
//         + (vAddress % PAGE_SIZE);
// But if the whole expression doesn't fit on one line then split it like this:
int physicalAddress = page_table_lookup(virtualAddress / PAGE_SIZE)
        + (virtualAddress % PAGE_SIZE);
// Expressions should be split so that operators are at the start of the
// continuation line, not at the end:
int physicalAddress
        = page_table_lookup(virtualAddressVariableWithLongerName / PAGE_SIZE)
        + (virtualAddress % PAGE_SIZE);
// If function parameters can be grouped with the function name then do so
// rather than split them:
int result1
        = function_that_takes_two_fn_parameters(parameter_one, parameter_two);
// i.e. not like this:
// int result1 = function_that_takes_two_fn_parameters(
//         parameter_one, parameter_two);
// But if the parameters can't be grouped with the function name on one line
// but can be grouped with each other, then do that:
int result2 = function_that_takes_three_parameters(
        parameter_one, parameter_two, param3);
// If they can't be grouped either way then don't try:
int result3 = function_that_takes_even_more_parameters(parameter_one,
        parameter_two, parameter_three, parameter_four, parameter_five,
        parameter_six, parameter_seven, parameter_eight);
// Continuation lines for loop expressions are indented by 8 also:
while (!isLoopFinished(argument1) && !isLoopFinished(argument2)
        && !isLoopFinished(argument3)) {
    // but the statements within the loop are indented by 4:
    process(value);
}
```

Example 3: Continuation line indentation

1.5 Continuation lines for nested expressions CLANG-FORMAT 59

Where expressions are nested (i.e. parentheses are used or functions are called in arguments to functions) then 60
continuation lines are further indented to indicate the nesting. 61

```c
int result = function_one(arg1, arg2, arg3,
        function_two(arg4, arg5, arg6, arg7, arg8,
                function_three(arg9, arg10, arg11, arg12)));
if (conditionOne
        && (conditionTwo || conditionThree || conditionFour || conditionFive
                || conditionSix)
        && conditionSeven) {
    printf("true\n");
}
```

Example 4: Continuation lines for nested expressions

1.6 `for` loops

CLANG-FORMAT

Where continuation lines are needed for a **`for`** loop, indents should be made as per the following examples. If this is too confusing to follow, you can just rely on **`clang-format`** to do the formatting for you.

```c
// For loop where parenthesised expressions fit on one line
for (i = 0; i < count; i++) {
    // Statements within the loop are indented by four
    do_stuff();
}
// For loop where expressions do not fit on one line. The start of the next
// expression is indented by 8 characters
for (loopCount = 0; loopCount < numberOfElementsToCheck && !isStopCondition;
        loopCount++) {
    // Statements within the loop are still indented by four
    process(item[loopCount]);
}
// If the initialisation expression (an assignment expression) overflows
// then it is indented by 12. An overflowing test expression is not further
// indented if it is not nested
for (var = initial_value_of_loop_counter(argument1, argument2)
                + adjustment_factor_to_apply();
        var < numberOfElementsToCheck && haveNotFinishedYet()
        && definitelyHaveNotFinishedYet();
        var++) {
    process();
}
// If the initialisation expression uses the comma operator then
// indentation is by 4 characters.
for (loopCountVariableWithVeryLongName = 0,
    anotherVariableWithLongNameInitialisedHere = 2;
        // Following expressions are indented by 8. The subexpression is
        // further indented by another 8.
        loopCount < numberOfElementsToCheck
        && !(isStopConditionA || isStopConditionB || isStopConditionC
                || isStopConditionC);
        loopCount++) {
    process();
}
```

Example 5: Continuation lines for **`for`** loops

1.7 `switch` statements

CLANG-FORMAT

`case` labels must line up with the enclosing **`switch`** statement. The statements associated with a **`case`** label must be on a new line and indented by four spaces beyond the start of the **`case`** label.

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

Example 6: Correct indentation for **`switch`** statement

1.8 Continuation lines for braced initialiser lists

CLANG-FORMAT

Initialiser lists (for arrays or structs) follow the rules above – except where struct member names are used in an initialiser list that does not fit on one line. See the example below.

```c
// Initialiser list fits on the same line as the declaration:
double d1[10] = {1.11111111, 2.2222222, 3.3333333, 4.4444444, 5.555555};
struct Parameters1 param1 = {.initial = NULL, .end = NULL, .runningCount = 0};
// Initialiser list fits by itself on a continuation line:
double d2[10]
        = {1.11111111, 2.2222222, 3.3333333, 4.4444444, 5.555555, 6.666666};
struct Parameters2 param2
        = {.initial = NULL, .end = NULL, .runningCount = 0, .total = 0};
// Initialiser list doesn't fit on a single line (or continuation line):
double d3[10] = {1.11111111, 2.222222, 3.333333, 4.4444444, 5.555555, 6.666666,
        7.777777, 8.888888, 9.9999999};
struct Parameters3 param3 = {NULL, NULL, 0, NULL, 0, false, true, 0, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL};
// Named lists are formatted differently if they don't fit on one line:
struct Parameters4 param4 = {.initial = NULL,
        .end = NULL,
        .runningCount = 0,
        .fileName = NULL,
        .limit = 0,
        .upper = false,
        .lower = true,
        .length = 0,
        .userName = NULL};
```

Example 7: Continuation lines for array and struct initialiser lists

1.9 Continuation lines for string constants

CLANG-FORMAT

Where a string constant does not fit on one line then it may be split over multiple lines. C will concatenate adjacent string constants into a single string constant so `"ab" "cd"` is the same as `"abcd"`. Continuation lines for a string constant must align with the commencing double quote characters (i.e. no indenting by 8).

```c
const char* const longMessageVariableName1
        = "This is an example of a string constant that fits on the next line";
const char* const longMessageVariableName2 = "This approach is also OK even if "
                                             "the string would fit on one line";
const char* const message3
        = "This is another example of a long string constant that must be split"
          " over multiple lines";
```

Example 8: Continuation lines for string constants

2 Horizontal Spacing

2.1 Unary operators

CLANG-FORMAT

There should be no spaces between unary operators and their operands.

```c
i++; // No space between i and ++
int j = -i; // No space between - and i
```

Example 9: Spacing around unary operators

2.2 Binary and ternary operators

CLANG-FORMAT

There should be spaces on both sides of binary or ternary operators.

```c
// Note the spaces around * _ && || ^ & | > ? : in these examples
int y = m * x + c;
bool isValid = exist(filename) && (someCondition || otherCondition);
regValue = (a ^ b) & (c | d);
max = (a > b) ? a : b;
```

Example 10: Spacing around binary and ternary operators

2.3 Control statements and functions

CLANG-FORMAT

There must be a single space between a control statement (e.g. `if`, `for`, `while`, etc.) and the following open parenthesis. There must not be any spaces between a function name and the following open parenthesis. There must not be any spaces between a parenthesis and the following content. There must be no spaces before commas or semicolons but there should be space after each comma or semicolon that is not at the end of a line.

```c
for (int i = 0; i < 10; i++) {
    while (do_something(i, calc_value(i))) {
        do_something_else();
    }
}
```

Example 11: Spacing in control statements and function calls.

2.4 Spaces at the ends of lines

CLANG-FORMAT

There should be no whitespace prior to semicolons at the end of a line, but a semicolon can be on a line by itself if required – indented appropriately. There should be no whitespace at the end of a line (i.e. following a semicolon or other non-whitespace character at the end of the line).

2.5 Pointer declarations

CLANG-FORMAT

When pointers are declared, the `*` should be associated with the type, not the variable name – unless multiple pointers are being declared together.

```c
// When declaring pointers, the * should be next to the type, with a space
// before the variable/argument name:
int fn(int* num, char* str);
char* abc;
// However, if multiple pointers are declared in one statement, then the *
// should be next to the variable name:
char *str1, c, *str2, *str3;
```

Example 12: Spacing in pointer declarations

2.6 Array derefencing

CLANG-FORMAT

There must be no spaces prior to square brackets when dereferencing arrays.

```c
int nums[10];

printf("First num is %d\n", nums[0]);
```

Example 13: No spaces before square brackets

2.7 Spaces before comments

CLANG-FORMAT

Where a comment follows code on a line then there must only be a single space between the code and the comment. See examples 1, 2 and 9 above.

3 Vertical Spacing

3.1 Line termination character

CLANG-FORMAT

Each line (including the last line in a source file) should end with a newline character (`\n` – also known as a linefeed character, i.e. ASCII code 10). Do not use carriage returns (`\r`, ASCII code 13).

3.2 Use of blank lines

CLANG-FORMAT MANUAL CHECK

Function, struct and enum definitions must be separated from other code blocks by exactly one blank line. No separation is required between an immediately preceding comment and such a definition but a blank line must be present before the comment.

See Example 21 below.

Blank lines should also be used to separate groups of statements from each other to make the major steps of an algorithm distinguishable. This requirement is not checked by `clang-format`.

3.3 One statement per line

CLANG-FORMAT

Each statement must be on its own line.

```c
// Not: i++; j++
i++;
j++;
// Not: if(a) b();
if (a) {
    b();
}
```

Example 14: Line breaks between statements

3.4 Multiple assignment operators in the one statement

MANUAL CHECK

Do not use multiple assignment operators in the one statement – split these up into multiple statements.

```c
// Not a = b = expr();
a = expr();
b = a;
```

Example 15: Multiple assignments

3.5 Use of the comma operator

MANUAL CHECK

Do not use the comma operator to separate expressions where a semicolon can be used. This means the comma operator should be restricted to **for** loop initialisers, loop test expressions, etc. if its use is necessary. It should never be used just to reduce the number of lines in a function.

```c
void fn(int a, int b)
{
    int sum, diff; // Commas in variable declarations are fine provided
                   // the variables aren't being initialised with a non
                   // constant expression at the same time
    // Don't do this: int sum = a + b, diff = a - b;
    int product = a * b; // Declaring one variable and initialising it is fine
    int x = 0, y = 1; // Initialisations to constants can be put on one line

    // Dont't use a comma where you can use a semicolon and multiple lines:
    // Not this: sum = a + b, diff = a - b;
    // Do this:
    sum = a + b;
    diff = a - b;
    for (i = 0, j = 0; i < sum; i++, j++) { // Comma operators OK here
        // code ...
    }
}
```

Example 16: Multiple assignments

4 Braces 116

4.1 Function definitions 117

CLANG-FORMAT

The opening brace in a function definition must be placed on a line by itself. The matching closing brace must be at the start of a line also. 118 119

```c
int fn(void)
{
    // statements here
}
```

Example 17: Braces for function definitions

4.2 Control statements 120

CLANG-FORMAT

The statements associated with control statements (**if**, **else**, **for**, **while**, **do**, **switch**, etc.) must be enclosed in braces, even if there is only one such statement. 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. The open brace must be at the end of the line before the enclosed statement. The close brace must appear on a line by itself (unless an **else** clause follows an **if** statement, in which case the **else** must follow the close brace). The closing brace is aligned with the start of the control statement. 121 122 123 124 125 126

```c
// Not if(condition) do_something(); OR if(condition) { do_something(); }
if (condition) {
    do_something();
}
while (a < b) {
    a = do_something_else();
}
if (a) {
    do_this();
} else if (b) {
    do_that();
} else {
    do_other();
}
```

Example 18: Braces for control statements. See example 5 for **for** loop examples and example 6 for a **switch** example.

5 Naming Conventions 127

MANUAL CHECK

Note that all names are expected to be meaningful. This is not checked by `clang-tidy`. 128

5.1 Variable names 129

CLANG-TIDY

Variable names, function parameter 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. 130 131 132 133

```c
int count, wordLength, cursor, fileHandle;
char* htmlMethodName; // Note - lowercase even for a normally capitalised
                      // acryonym
struct Book book; // struct Book is the type name, book is the variable name
const char* courseName = "CSSE2310"; // constants follow the same convention

struct Person {
    char* name; // struct members have camelCase names
    char* homeAddress;
};
```

Example 19: Examples of variable names

[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`.

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`.

5.2 Defined constants and macros

CLANG-TIDY

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 5.1.

Examples: `MAX_BIT`, `DEFAULT_SPEED`

5.3 Function names

CLANG-TIDY

Function names should all be lowercase and use underscores to separate multiple words. Note that function pointers are variables and should use variable naming, as per 5.1.

```c
int main(int argc, char* argv[]);
void reset_secret_string(void);
// The following is a variable declaration for a function pointer
// (sortingFunction) that is initialised to point to the function alpha_sort()
int (*sortingFunction)(void*, void*) = alpha_sort;
```

Example 20: Examples of function names and a function pointer

5.4 Type names

CLANG-TIDY

Type names (i.e. from typedefs), struct 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 5.1.

```c
/* A player within the game - an element within a linked list */
struct PlayerInfo {
    char* name;
    int score;
    struct PlayerInfo* next;
};

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

/* Rectangle - typedef'd structs can be named but don't have to be */
typedef struct Rectangle {
    Point upperLeft;
    Point lowerRight;
} Rectangle;
```

Example 21: Examples of type declarations

5.5 Enumerated Types

CLANG-TIDY

Enumerated types must be named in the same way as any other type (see 5.4). Members of an enumerated type must follow the same naming as constants (see 5.2).

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

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

// Program exit codes - the definitions don't all fit on one line so they must
// be set out as one per line
enum ExtendedExitCodes {
    EXIT_SUCCESS = 0,
    EXIT_ARGS = 1,
    EXIT_FAILURE = 2,
    EXIT_FILE_NONEXISTENT = 3
};
```

Example 22: Examples of `enum` declarations

5.6 File names

MANUAL CHECK

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`

6 Comments

MANUAL CHECK

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.

6.1 Function comments

MANUAL CHECK

Function comments should describe parameters, return values, any global variables modified by the function 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.

[3]See the note about global variables in section 7.3

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.)
 * arg2: ... (repeat for all arguments)
 *
 * Returns: description of what is returned
 * Global variables modified: (if applicable) list of global variable names
 *     modified by this function
 * 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. (Remember that you are not permitted
 *     to copy some code.)
 */
int function_name(char* arg1, int argument2)
{
    // ...
}
```

Example 23: Recommended function comment template

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 etc.). The following example is acceptable – the function comment contains all the required information.

```c
/* Calculates and returns the sum of integers from 1 to n. If n is less than
 * or equal to 0 then 0 is returned. It is assumed that the sum does not
 * overflow - the result will be incorrect if it does.
 */
int sum(int n)
{
    int i, s = 0;
    for (i = 1; i <= n; i++) {
        s += i;
    }
    return s;
}
```

Example 24: Another example of a function comment

See examples 27, 28, and 29 below for further examples of function comments.

6.2 Code references

MANUAL CHECK

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 or learned about something 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. If you are referencing a website then you must provide the full URL, not a shortened URL (e.g. not `bitly` or `tinyURL` links). It is permissible for the URL to be broken into multiple lines so as to not exceed the line length restriction.

See examples of references below:

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

```c
// REF: The following block of code is taken from code submitted by me for the
// REF: CSSE2010 AVR programming assignment in semester two, 2023.
```

Example 25: 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 26: Example code reference – code you have been inspired by

```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
 * REF: function & 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 27: 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 28: Example code reference – in a function comment – for AI generated code. Note that you must separately document your interaction with any AI tool. See the *Documentation required for the use of AI tools* document.

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

Example 29: Example code reference – in a function comment – for AI generated code (2). Note that you must separately document your interaction with any AI tool. See the *Documentation required for the use of AI tools.*

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.**

7 Readability

7.1 Line Length

CLANG-FORMAT

Lines must not exceed 80 columns in length including whitespace. This is based on a historical "standard" screen size of 80 columns, but is still used today to aid readability and printing. If a statement/expression does not fit on one line then it must be split over multiple lines with continuation lines indented as described in section 1 – Indentation.

7.2 Function Length

CLANG-TIDY

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.

7.3 Global variables

CLANG-TIDY MANUAL CHECK

Global and static variables must not be used unless there is no other way to implement the required functionality. For example, a single global variable may need to be used with a signal handler. Global constants, declared using `const` are permitted. Note that `2310stylecheck.sh` will give a warning on the presence of non-`const` global variables, however no warning is given for non-global `static` variables – these must be checked for manually and removed. You can use the `--globalOK` flag to `2310stylecheck.sh` to remove warnings about global variables.

7.4 Magic Numbers

CLANG-TIDY

"Magic numbers" are to be avoided. Defined constants should be used instead of numbers scattered throughout code without context. These can be `#define` constants or global constants declared using `const`. The latter is preferred because this gives you more type safety, but this is not a formal requirement. 0, 1 and 2 are almost never magic numbers. Floating point constants 0.0, 10.0 and 100.0 are also almost never magic numbers.

```c
// Constants can be defined using #define or "const variables"
#define NUM_LETTERS_IN_ALPHABET 26
const int minPasswordLength = 8;

for (int i = 1; i <= NUM_LETTERS_IN_ALPHABET; i++) {
    //...
}
if (strlen(passwd) < minPasswordLength) {
    //...
}
```

Example 30: Use of constants to avoid magic numbers

7.5 Duplicate Includes

CLANG-TIDY

Do not `#include` any header file more than once in a source file.

7.6 `else` after `return`, `continue`, etc.

CLANG-TIDY

Do not use an `else` or `else if` clause if the previous `if` branch(es) result in the control flow being interrupted (e.g. through the use of `return`, `continue`, `break`, etc.

```c
// Don't do it this way
void foo(int value)
{
    int local = 0;
    for (int i = 0; i < 42; i++) {
        if (value == 1) {
            return;
        } else {
            local++;
        }

        if (value == 2) {
            continue;
        } else {
            local++;
        }
    }
}
```

Example 31: `else` after `return` etc. – don't do this

```c
// Do it this way
void foo(int value)
{
    int local = 0;
    for (int i = 0; i < 42; i++) {
        if (value == 1) {
            return;
        }
        // This line can only be reached if the
        // condition above is false anyway
        local++;

        if (value == 2) {
            continue; // Ditto
        }
        local++;
    }
}
```

Example 32: `else` after `return` etc. – do it this way

7.7 Function parameter names

CLANG-TIDY

Parameter names must be present in function declarations and the names must match those used in the function definition.

C requires that parameters be named in function definitions. In the rare circumstance that a function parameter is unused then the `unused` attribute may be applied as shown in the examples below. You should review the code to determine if the function parameter should be present in the first place, but in some cases you have no choice, e.g. with `main()` if one of `argc` or `argv` is used but not the other. (If neither is used then you can omit the parameters to main.)

```c
// Parameter names don't match or are unnamed - this is NOT permitted
void fn1(int a, char* b, double); // declaration - one parameter unnamed - wrong

void fn1(int i, char* s, double d)
{ // definition - names don't match - wrong
  // code here
}

// The following is OK because the parameter names match
// (Though the names should be more meaningful!)
void fn2(int a, char* s, double d); // declaration

void fn2(int a, char* s, double d __attribute__((unused)))
{ // definition, d not used in function, but function should almost certainly be
  // rewritten to omit the parameter.
  // code here
}

int main(int argc __attribute__((unused)), char* argv[])
{ // argv used but not argc
  // code here
}
```

Example 33: Function parameter names

7.8 Use of `const` for pointer function parameters

If the use of `const` with a function parameter of a pointer type will make a function interface safer then it should be used.

```c
// The function:
char first_char(char* p)
{
    return *p;
}

// would be safer written as the following. It makes it clearer that the
// string pointed to by p will not be modified by the function.
char first_char(const char* p)
{
    return *p;
}
```

Example 34: Where `const` should be used

8 Other Issues

8.1 Bug prone code

The `clang-tidy` options used in the `2310stylecheck.sh` tool also check for various bug prone coding approaches. These are not described in detail here but it is your responsibility to fix these issues if they are flagged by `clang-tidy`.

Specifically, the following `clang-tidy` checks are used. You can see more details about these at https://clang.llvm.org/extra/clang-tidy/checks/list.html

- `bugprone-inc-dec-in-conditions`
- `bugprone-macro-parentheses`
- `bugprone-macro-repeated-side-effects`
- `bugprone-misplaced-operator-in-strlen-in-alloc`
- `bugprone-misplaced-pointer-arithmetic-in-alloc`
- `bugprone-misplaced-widening-cast`
- `bugprone-multi-level-implicit-pointer-conversion`
- `bugprone-non-zero-enum-to-bool-conversion`
- `bugprone-not-null-terminated-result`
- `bugprone-posix-return`
- `bugprone-redundant-branch-condition`
- `bugprone-signal-handler`
- `bugprone-sizeof-expression`
- `bugprone-string-literal-with-embedded-nul`
- `bugprone-suspicious-enum-usage`
- `bugprone-suspicious-include`
- `bugprone-suspicious-memset-usage`
- `bugprone-suspicious-semicolon`
- `bugprone-switch-missing-default-case`
- `bugprone-terminating-continue`
- `bugprone-unsafe-functions`
- `bugprone-unused-return-value`
- `misc-confusable-identifiers`
- `misc-redundant-expression`

8.2 Compilation

MANUAL CHECK

Your assignments will use the C99 standard, and must be compiled with the following flags as a minimum: `-Wall -Wextra -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.

8.3 File encoding

MANUAL CHECK

All source files must use an ASCII-compatible encoding.

8.4 Modularity

MANUAL CHECK

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.

8.5 Banned Language Features

CLANG-TIDY MANUAL CHECK

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. Use of `goto` is checked by `2310stylecheck.sh`.
- 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.

9 Updates to this Document

- v3.06 – original version for semester two 2024.