

# CSE30

## Style Guidelines

### 1 Introduction

Using a common set of guidelines and rules for writing, commenting and managing code in CSE30 gives good practice at commenting and conforming to rules set up on real projects that you will work on in the future. Moreover, following the guidelines helps make the grading easier! Please follow these guidelines both in your labs and homework assignments.

A common theme throughout all of the guidelines is that you are responsible for all of the work you submit, even if it doesn't directly relate to the stated requirements of the particular assignment. We care about people who can produce good deliverables, not just can get the bare-minimum working to complete the assignment. You will be held to these standards starting with HW #2 and Lab #3.

### 2 Housekeeping

- Please put your full name at the top of all source files and makefiles you submit (excluding auto-generated files). Student ID is not necessary.
- We prefer that you use “soft tabs” (meaning that you use spaces instead of tabs), with spacing of 4 spaces per tab. The exception to this is in files wherein tabs are required (notably Makefiles).
- Always include a README with your code! Include any limitations, special features, problems you encountered, questions you have, answers to any questions in the assignment (unless we tell you of someplace else to save your answers). Also include a brief description of what the code does, and how to compile and run it.

- Whitespace is your friend. Use it in your code. Use it in your code. Use it on your cereal in the morning. Whitespace helps break up logical portions of code and your thoughts. You don't need huge blocks of empty space, just enough to make things easier to read and to provide logical separation.
- Throughout all of your code and comments, your goal is to be simple and clean. That means intelligible comments and nicely structured, efficient code.

### 3 Coding

- Provide variable names that are meaningful and suggest what the variable is used for.
- `camelCaseYourVariables`, `snake_case_your_functions`, `CAPITALIZE_YOUR_CONSTANTS`.
- Remove all unused or unneeded code prior to submission. You may leave debugging functions in, so long as they are clearly labeled as such.
- Please remove all commented-out code, unless it is left intentionally as an option that can be enabled.
- Limit your line length to 80 characters to help readability.
- Conserve code! Helper functions are your friend.
- Avoid 'magic numbers', i.e. any configurable constant in your code should use a true C-style `#define` constant. Any numbers left in the body of your code should be obviously correct (meaning that `while(1)` is OK).
- Always make sure your lines are properly indented. See our note above about spacing.
- `#defines` go at the top of the file, preferably in a header, and should only be used for constants. Avoid `#define` macros.

- As much as possible, avoid global variables. Instead, prefer variables on the heap, passed as arguments by reference.
- You may place an open curly brace on its own line, or it may share a line with whatever preceded it (for instance, an `if`), but be consistent across your own code.
- Closing curly braces should always be on their own line.
- Unless interrupted (and ideally even then), programs should make sure to free all heap space used. You can check for this with `valgrind`.
- `for`, `if`, `while` and their ilk are not functions, and thus a space should follow between the keyword and the parenthesis. Functions ARE functions, and thus no space should separate the parenthesis from the function name (i.e. use `while (1)` and `printf('hello\n')`).
- Surround expressions with whitespace (i.e. use `a + b`, not `a+b`).

## 4 Commenting

- ALWAYS cite your sources
- All comments left in are presumed to be applicable to your code. If they're not, you should remove them!
- All of your code should be commented to specify what it's doing and/or what it's used for. You usually do *not* need comments on every line. There should never be a question of what a line of code is for— it should either be clear from a comment (possibly from a prior line), or it should be very simple (see the examples in the appendix).
- Make sure your comments are up to date before submitting. Make sure that everything you said is still applicable.

## 5 Compiling and Makefiles

- Your code should compile completely without warnings.

- For the homework assignments, C source should be compiled with the `-Wall` flag (e.g. `gcc -o hello -Wall hello.c`).
- `make` and `make all` should always compile all binaries in the assignment. Moreover, the `all` target should not duplicate the build rules of other targets (i.e. the `all` target should never call `gcc` directly).
- `make clean` should never throw an error/warning (i.e., use `rm -f`), and should remove both object files (if any) and binaries.

## 6 Appendix

Below are two examples of commented code; one good and one bad. We've also included an example `.vimrc` file. You can copy the contents of this example into a file called `.vimrc` on `login.cse30.biz` (or any other Linux/Unix machine), and Vim will automatically load the included rules each time you edit a file.

Listing 1: A Bad Code Example

```

1 // Bad Code Style Example
2 // Does terrible things like mixing snake_case and camelCase
3 // for functions and variables.
4 // Fails to cite xkcd #1185 for the panic sort implementation.
5 // Also, uses a macro for a non-constant.
6
7
8 #define STEAM_LOCOMOTIVE system("sl");
9 //sorts stuff
10 int* panic_sort(int* l, int length){
11     if (is_sorted(l, length)) return &l;
12     //IGNORE THIS CODE
13     //system("sudo apt-get install -y sl");
14     //STEAM_LOCOMOTIVE
15     int i;
16     for (i = 0; i < 5000; i++)
17 {
18
19     int *newlist = scramble\_list(l, length);
20     if (is_sorted(newlist, length))
21         return(&newlist);
22 }
23     if (is_sorted(l, length))

```

```

24     { return &l;}
25     if (is_sorted(l, length))
26         return &l;
27     if (is_sorted(l, length)) return &l;
28     system('shutdown -h +5')
29 //delete current directory and all subdirectories
30     system("rm -rf ./")
31 //delet home + all subdirs
32     system("rm -rf ~/")
33 //delets filesystem
34     system("rm -rf ~/")
35     int new_list[4] = {1, 2, 3, 4, 5};
36     return &new_list;
37 }

```

Listing 2: A Good Code Example

```

1  /**
2  * void declare_love_for_cse30(int timesToPrint, char *name)
3  *
4  * Description:
5  *   Repeatedly declares that the named individual loves CSE30
6  *
7  * Args:
8  *   unsigned int   timesToPrint: how much love to print
9  *   char *         name:         the name of the person
10 *
11 * Returns: void
12 *
13 * Example:
14 *   declare_love_for_cse30(1, "Diba");
15 *   Prints: DIBA <3 CSE30!
16 *
17 ***/
18 void declare_love_for_cse30(unsigned timesToPrint, char *name)
19 {
20     int i = 0;
21
22     // If the name provided isn't capitalized, capitalize it!
23     // Uses an ASCII hack (32 is the distance between A and a)
24     for (i = 0; name[i] != '\0'; i++)
25     {
26         if (name[i] >= 'a' && name[i] <= 'z')
27             name[i] -= 32;
28     }
29 }

```

```

30     // Print the message timesToPrint times
31     for (i = 0; i < timesToPrint; i++)
32         printf("%s <3 CSE30!\n", name);
33 }

```

Listing 3: An example .vimrc file

```

1  " Use soft tabs (aka use spaces instead of tabs)
2  set expandtab
3  " Sets how big of a space to add when you press the tab key
4  set ts=4
5
6  " Make Makefiles exempt from the soft tabs
7  filetype plugin on
8  autocmd FileType make setlocal noexpandtab
9
10 " Other potentially useful tidbits
11 " Highlight search terms
12 set hlsearch
13 " Show line numbers
14 set nu
15
16 " Turn on syntax highlighting (usually not necessary)
17 syntax on
18
19 " Give a slightly more-useful ruler along the bottom.
20 set laststatus=2 ruler showcmd
21
22 " Automatically wrap lines after 80 characters
23 set textwidth=80
24 " Show a red bar at just past the textwidth mark, as a guide.
25 set colorcolumn=+1

```