Style Guidelines

Overview:

Your style will be graded on the following items.

Failure to follow anything listed in this document will result in deductions from your programming assignments!

File Headers:

Every file needs a file header (yes, even the README). This header needs to have the format of:

```
/*
 * Filename: filename.extension
 * Author: Your name
 * UserId: Your cs30x account (you will lose points if this is incorrect!)
 * Date: Date when file was created
 * Sources of help: All sources of help you used (tutors, Piazza posts, books,
 *                  websites, etc.). Other students should not be listed here as
 *                  other students should not be looking at your code.
 *                  If you did not use any sources of help, you MUST put "None".
 */
```

[See PA0 starter code for examples]

Function Headers:

We have two different type of function headers. Those for C functions and those for Assembly routines. We are looking for the headers to be EXACTLY in the format we describe.

C Function Header

```
/*
 * Function Name: function name()
 * Function Prototype: Definition of the function. For example, for main this
 *                     would be int main( int argc, char * argv[] );
 * Description: Description of how the function behaves
 * Parameters: Name any parameters passed into the function, and how they are
 *             used. If no parameters are used, say None
 * Side Effects: Any behaviors the function might exhibit that are not
```

```
 *               immediately apparent (related to the return value). Examples
 *               include updating a value pointed to by a parameter,
 *               printing things to stdout/stderr, reading/writing from file
 *               parameters, etc (ask on Piazza if you are unsure if an action
 *               you are taking has a side effect). If there are no side
 *               effects, say None
 * Error Conditions: Explain any potential errors/exceptions that may occur if
 *                   your function is used incorrectly. If there are no error
 *                   conditions, say None
 * Return Value: What does the return value of this function represent/what
 *               will it be used for?
 */
```

Assembly Routine Header

Assembly routine headers require everything that the C Headers do, in addition to listing any registers used, as well as any variables allocated on the stack. Stack variables include local variables **and** formal parameters. Note that if no variables are stored on the stack, you do not need a “Stack variables” section in the function header for that function.

```
/*
 * <Full C Header>
 *
 * Registers used:
 *     <register> - <use> -- <description of what the value represents>
 * example:
 *     r0 - arg 1 -- number of values to add together
 *     r1 - arg 2 -- starting value to start adding from
 *     r2 - local var -- keeps track of loop index
 *
 * Stack variables:
 *     <name> - <fp offset> -- <description of what the value represents>
 * example:
 *     sum - [fp - 8] -- holds the sum of values
 */
```

[See the PA0 starter code for examples]

Comments:

Well documented code is good code. Commenting is a skill that you will develop in time, but here are some guidelines to help jumpstart the process.

Assembly Files

For Assembly files, you’ll want to comment almost every line. Instructions can be pretty cryptic and it is easy to forget what you were intending on doing so comment as you go.

Back to Top

If your Assembly is not commented, tutors will NOT help you in the lab. Sometimes it can also be helpful to include the C code you are translating in your comments.

Use inline comments after your instructions, marked with @

```
mov   r2, r0          @ Copies the input parameter into register 2.
add   r3, r0, r1      @ Adds first and second argument.
```

Make sure to align all your inline comments vertically.

C Files

For C files, comments can be more infrequent. You don't need to comment every line, but you'll need to comment logical blocks. If there are a few lines of code that work towards the same goal, start the block off with a comment explaining what will happen in the next few lines. Your comments should be meaningful and should give a high level description of what your code is doing.

In addition, make sure to comment complicated lines of code to make them easy to understand at first glance. Avoid commenting trivial lines of code that can be quickly understood at first glance just by looking at the code.

bad

```
if (arr[i] == NULL) {
  i++; // "Increment i" - bad comment, it's obviously incrementing i
  continue;
}
```

better

```
if (arr[i] == NULL) {
  i++; // "Skip to next element" - high level idea of what i++ is doing
  continue;
}
```

Magic Numbers:

No Magic Numbers! Any numbers other than 0, 1, and -1 are magic numbers. All character literals (except '\0' which is ASCII value 0) are considered as magic as well! In the event you need to use a number or character literal (it happens a lot), define a constant at the top of your file. The name of the constant should be in SCREAMING_SNAKE_CASE, and should indicate the meaning of the number (what it is used for/what it represents). Using constants in this way (instead of magic numbers) will improve the readability and maintainability of your code.

Example:

C:
```
#define MIN_RANGE 3
```

Assembly:
```
        .equ    MIN_RANGE, 3
```

Note the indentation:
C -- not indented
Assembly -- indented

IMPORTANT: Give your constants meaningful names. Simply saying `#define TWO 2` is not good enough; `TWO` gives no indication of what the constant is used for, and does nothing to improve the readability or maintainability of your code.

Note: Strings (e.g. "hello") do not count as magic numbers, so this section does not apply to them. **However**, it is good practice to define constants for them anyway, especially if you use the same value in multiple places.

Back to Top

Variable Names:

Use reasonable names for your variables and constants. The names of variables/constants should indicate their purpose/what they represent. This greatly improves readability and helps others (and yourself) more easily follow your code.

Capitalization:

	Convention	Example

Variables	camelCase (each word after the first begins with an uppercase letter)
	- OR -
	snake_case (each word in lowercase separated by underscores)
	
	Choose **one** and stay consistent.
		`numOfArguments`
		
		`num_of_arguments`

Constants	SCREAMING_SNAKE_CASE (all uppercase with underscores between words)	`NUM_OF_ARGUMENTS`

Spell out words fully:

The words used in variable names should be spelled out fully. Common abbreviations are fine (e.g. "coord" for "coordinate", "num" for "number"), but don't aggressively abbreviate.

good

```
#define TABLE_WIDTH 10
#define OUTFILE_INDEX 1

unsigned char byteMask;
char * linePtr; // "ptr" is a common abbreviation of "pointer"
```

bad

```
#define TW 10
#define IDO 1

unsigned char byMsk;
char * lp;
```

Don't use single-letter variable names:

Single-letter names don't provide any information to the reader and should **not** be used. However, there are some exceptions:

good

```
for (int i = 0; i < length; i++) { ... }  // Common iterator name "i"

float x;  // Common coordinate names
float y;
float z;
```

Don't use incremented variable names:

Incremented variable names are variables that use the same base name with an additional number that usually increments. If you find yourself resorting to names like this, try to think of better names that describe what the variable is used for.

Back to Top

```
int var1; // Just "var" with numbers after it. Not informative at all!        bad
int var2;
int var3;
```

Use of Whitespace:

Indentation:

	Indentation Size	Rules
C	2 spaces	No tabs!
Assembly	1 tab (tab width = 8 spaces)	No spaces!

In C, when a long line is split among 2 lines, make sure to indent the second line farther than the first to indicate that it is a continuation of the previous line. There are many different ways to do this depending on the situation, however, the overall goal is to improve readability. The following are some examples.

Long argument lists:

Break the line on any of the commas, then line up the second line with the first argument provided.

```
variable = reallyLongFunctionName(arg0, arg1, arg2, arg3,
                                  arg4, arg5);
```

Long argument names:

Break the line on the opening paren of the argument list, then indent the subsequent lines in a way that enhances readability (two levels of indentation is a good rule of thumb).

```
variable = reallyReallyLongFunctionName(
    thisIsTheFirstArgBeingPassed,
    thisIsTheSecondArgBeingPassed);
```

Long expression using infix operators:

Break the line on the infix operators and line up the second line with to be on the right hand side of the equals sign.

```
variable = long + string * of -
           infix % operations;
```

Newlines:

Whitespace is your friend! Separate logical blocks of code with a blank line in between. It is a lot harder to debug a wall of text than it is to work with something that is broken up. However, don't go overboard. You don't need to have a newline between every two unrelated lines of code, and definitely don't use too many at once. Make sure to remove any excessive newlines at the end of each file.

Back to Top

Example code with good newline usage:

good

```
int parse(char * str) {
  char * endptr; // Pointer to the first non-digit character
  int value;     // Stores the parsed value to return

  // Convert str into an int
  errno = 0;
  value = strtol(str, &endptr, BASE);

  // Check for ERANGE error
  if (errno != 0) {
    return -1;
  }

  // Check for non-digit character
  if (*endptr != '\0') {
    return -1;
  }

  return value;
}
```

Example code with too few newlines:

bad

```
int parse(char * str) {
  char * endptr; // Pointer to the first non-digit character
  int value;     // Stores the parsed value to return
  // Convert str into an int
  errno = 0;
  value = strtol(str, &endptr, BASE);
  // Check for ERANGE error
  if (errno != 0) {
    return -1;
  }
  // Check for non-digit character
  if (*endptr != '\0') {
    return -1;
  }
  return value;
}
```

Back to Top

Example code with too many newlines and extra newlines at the end of the file:

bad

```
int parse(char * str) {

  char * endptr; // Pointer to the first non-digit character

  int value;     // Stores the parsed value to return

  // Convert str into an int

  errno = 0;

  value = strtol(str, &endptr, BASE);


  // Check for ERANGE error

  if (errno != 0) {

    return -1;
  }

  // Check for non-digit character
  if (*endptr != '\0') {

    return -1;
  }

  return value;

}
```

Binary Operators:

Make sure to use spaces around binary operators--this greatly improves readability. For example:

good

```
for (int index = foo; index < (foo * sizeof(bar)); index += STEP_AMT) {
  // ...
}
```

bad

```
for(int index=foo;index<(foo*sizeof(bar));index+=STEP_AMT){
  // ...
}
```

[note that this applies to binary operators in general, not just binary operators in for-loops]

---

80 Characters Per Line:

No lines over 80 characters! If you have a line that is 81+ characters, break it into two lines (following the rules on indentation above). Be very careful when writing comments as those tend to be where students get

Back to Top

deducted. Also keep in mind that whitespace characters (spaces, tabs) count towards the 80-character rule. Your assignments will be graded with the default tab width of 8 spaces.

Add the following lines to your .vimrc file to easily detect characters beyond the 80 character limit. (Lines beginning with quotation marks are just comments to explain what the following command does.)

```
" Highlight the 80th column (can change to 81st column if you prefer)
set colorcolumn=80

" Highlight text over 80 characters in red
autocmd BufEnter * match ErrorMsg /\%81v.\+/
```

Use the following command to detect lines over 80 characters long in a file:

```
expand -t 8 <file> | grep -ne '.\{81\}'
```

This will expand tabs inside `<file>` to 8 spaces, then look for lines that have 81+ characters. The expansion step is important, since tab (`\t`) characters count as a single character in `grep`, so using the grep command alone will miss lines that are pushed over 80 characters by tabs.

Other:

Curly braces:

You MUST use curly braces for **all** control-flow structures, even if the body is a single line.

good

```
if (condition) {
  printf("hello world!\n");
}

for (int i = 0; i < length; i++) {
  printf("%d\n", arr[i]);
}
```

bad

```
if (condition)
  printf("hello world!\n");

if (condition) printf("hello world!\n");

for (int i = 0; i < length; i++)
  printf("%d\n", arr[i]);
```

Printing Output:

Make sure you print different types of output to the correct output stream:

Error output	`stderr`
Normal output	`stdout` (unless the assignment specifies otherwise)

Assembly Looping Construct:

Stick to the Assembly looping construct covered in class (if this doesn't make sense it means it hasn't been covered yet). We will be checking for this and failure to follow our directions will result in a deduction.

Back to Top

Assembly Syntax:

Use the new unified ARM assembly syntax. This means that all instructions and registers should be written in lowercase, and there should be no # symbols written before constants.

```
.syntax unified                                                    good

.equ  FP_OFFSET, 4

mov   r3, 420
```

```
.EQU  FP_OFFSET, #4                                                bad

MOV   R3, #420
```

When in doubt, refer to the lecture notes handed out in class for the ARM assembly style you should be writing in.

Commented-out Code:

Remove any and all commented-out code before turning your assignments in. Leave only what is relevant.

TODO's:

Once a TODO is "TODONE" (meaning you completed the task that needed to be done), remove the TODO from your code. You must remove all TODO's from your code before turning your assignments in.

Back to Top