

15-213/18-213/15-513 Intro to Computer Systems: Code Style

Just as important as the functionality of your code is your code's readability to others. Therefore, in 15-213/18-213/15-513 (and other CS/ECE courses you will take), we will be paying close attention to your coding style and taking it into consideration when assigning grades.

Each semester the question of "What do you mean by 'good style'?" comes up. The course staff has created this document to try and answer that question. The most basic requirement is a consistent and logical style that makes the purpose of your code clear to the reader. We expect you to pick something that is readable and makes sense, and then stick to that through an entire project. The key points we will be looking for are:

Good Documentation

Good code should be mostly self-documenting: your variable names and function calls should generally make it clear what you are doing. **Comments should not describe what the code does, but why;** what the code does should be self-evident. (Assume the reader knows C better than you do when you consider what is self-evident.)

There are several parts of your code that do generally deserve comments:

- **File header:** Each file should contain a comment describing the purpose of the file and how it fits in to the larger project. This is also a good place to put your name and email address.
- **Function header:** Each function should be prefaced with a comment describing the purpose of the function (in a sentence or two), the function's arguments and return value, any error cases that are relevant to the caller, any pertinent side effects, and any assumptions that the function makes.
- **Large blocks of code:** If a block of code is particularly long, a comment at the top can help the reader know what to expect as they're reading it, and let them skip it if it's not relevant.
- **Tricky bits of code:** If there's no way to make a bit of code self-evident, then it is acceptable to describe what it does with a comment. In particular, pointer arithmetic is something that often deserves a clarifying comment.

Good Use of Whitespace

Proper use of whitespace can greatly increase the readability of code. Every time you **open a block of code** (a function, "if" statement, "for" or "while" loop, etc.), you should indent one additional level.

You are free to use your own indent style, but you must **be consistent**: if you use four spaces as an indent in some places, you should not use a tab elsewhere. (If you would like help configuring your editor to indent consistently, please feel free to ask the course staff.)

Line Length

While there are many different standards for line length, we require that your lines be **no longer than 80 characters**, so we can easily view and print your code. If you indent with tabs please assume a tab size of 4 characters when calculating line lengths. To quickly check that file.c does not exceed 80 characters, run "wc -L file.c" to see its max line length.

Good Variable Names

Variable names should be descriptive of the value stored in them. Local variables whose purpose is self-evident (e.g. loop counters or array indices) can be single letters. Parameters can be one (well-chosen) word. Global variables should probably be two or more words.

Multiple-word variables should **be formatted consistently**, both within and across variables. For example, "hashtable_array_size" or "hashtableArraySize" are both okay, but "hashtable_arraySize" is not. And if you were to use "hashtable_array_size" in one place, using "hashtableArray" somewhere else would not be okay.

Magic Numbers

Magic numbers are numbers in your code that have more meaning than simply their own values. For example, if you are reading data into a buffer by doing "fgets(stdin, buf, 256)", 256 is a "magic number" because it represents the length of your buffer. On the other hand, if you were counting by even numbers by doing "for (int i = 0; i < MAX; i += 2)", 2 is not a magic number, because it simply means that you are counting by 2s.

You should use #define to clarify the meaning of magic numbers. In the above example, doing "#define BUFLen 256" and then using the "BUFLen" constant in both the declaration of "buf" and the call to "fgets".

No "Dead Code"

"Dead code" is code that is not run when your program runs, either under normal or exceptional circumstances. **These include "printf" statements you used for debugging purposes but since commented.** Your submission should have no "dead code" in it.

Modularity of Code

You should strive to make your code modular. On a low level, this means that you should not needlessly repeat blocks of code if they can be extracted out into a function, and that long functions that perform several tasks should be split into sub-functions when practical. On a high level, this means that code that performs different functions should be separated into different modules; for example, if your code requires a hashtable, the code to manipulate the hashtable should be separate from the code that uses the hashtable, and should be accessed only through a few well-chosen functions.

Failure Conditions/Error Checking

When writing a program, we usually only consider the success case. It is equally, if not more, important to consider the failure cases. Many things can fail in your program: the user's input might not match your expected format, malloc might return NULL, the filename the user gave you might not exist, the user might not have permission to read the file they specified, the disk might fill up, the network host you were talking to might be down... the list goes on. It is important to think about what your program can do to resolve these errors, or how it should present them to the user if it can't resolve them. For example, if malloc fails in a crucial part of your program, you might have no choice but to print a fatal error message and exit. But if the user specifies an invalid file to open in an interactive program, it would be much nicer if you told them the file was invalid and gave them a chance to correct the name.

In particular, network servers (and, as you will learn if you take 15-410, operating systems) need to be particularly **robust**. **强健的.** No matter what one client (or process) tries to do, your server (or kernel) should never crash. Error handling is more difficult in such cases, as you need to convert what is a "fatal error" from a client's perspective into something that won't actually kill the server process.

Proper Memory and File Handling

If you allocate memory (`malloc`, `calloc`), you should free it after use. Your program should not have memory leaks. If you use `open` a file, you should close it after use. Closing a file is very important, especially with output files. The reason is that output is often buffered.

Consistency

This style guide purposefully leaves many choices up to you (for example, where the curly braces go, whether one-line "if" statements need braces, how far to indent each level). It is important that, whatever choices you make, you remain consistent about them. Nothing is more distracting to someone reading your code than random style changes.