Coding Guidelines and Standards

Daniel Bittman Darrell Long Ethan Miller

CSE 130 Fall 2019

A vital goal of computer programming and system implementation is *managing complexity*; therefore, you should strive to reduce complexity at all levels, including in the code itself. This document lays out the standards that we expect you to adhere to for code, development practices, and documentation. **To get full points on your assignments, you are required to follow each item in this document**, except items labeled "optional". This will help you as you implement assignments, and will help the graders return grades in a timely fashion.

Your assignments are expected to compile on `clang` version `7` on a standard `Ubuntu` 18.04 installation[1], and must compile without warnings using the following flags: `-Wall -Wextra -Wpedantic -Wshadow -std=gnu++11 -O2`. We will test your code under these flags, on the aforementioned system using the aforementioned compiler.

1 The Code

1. **C++**: We're allowing the use of C++ to allow you to use existing container types and some useful abstraction features. We are *not* allowing it because we want to see who can use template metaprogramming the best. You may use any features present in `C++-11` and the associated standard library (unless otherwise specified by the assignment), but do not over-do it. If the grader doesn't understand your code, you can't get points for it.

   You are not required to use C++! We will not dock points if you decide to implement your own vector type or hash table instead of using the standard library.

2. **Complication**: Unnecessarily over-complicated code will be marked down. Part of the goals of this course are to teach you to not over-engineer your solutions; therefore, adding unnecessary complication will earn fewer points. If you have concerns over your system design, you may discuss your design with the course staff.

3. **External libraries and code**: You are allowed to use the standard C library, the standard C++ library (including the STL), and any additional libraries specified in the assignment. **However, individual assignments may bar you from using particular library functions, even from the standard C and C++ library.** If you wish to use an external library, please ask the course staff before doing so. If you include code from external sources, you must document where it came from. Note that copying code from online may lose you points; if the point of the assignment was to implement a hash table, and you copied the code for one from `stackoverflow.com`, you didn't *really* do the assignment, did you?

4. **Makefiles and building**: We expect your project to build by simply typing `make` in the assignment's subdirectory in your repo. This means that each assignment will need to supply a `Makefile`. We're not requiring it to be complex, but it must support the targets "`all`" (as default; must build the project), "`clean`" (must remove built artifacts except for the final executable), and "`spotless`" (must do `clean`, then remove the executable). We will supply a `Makefile` template on canvas that contains instructions on how to modify it.

1.1 Style

1. **Run your code through `clang-format`**. This will ensure that your code uses a uniform style throughout. We will provide a `clang-format` config file on canvas, and the `Makefile` template will contain a target which runs `clang-format`.

2. **Self-documenting code**. Variable names, function names, class names, *etc.*, must be sensible, and reflect the purpose. Loop-counter variables are the only variables that may be a single letter, but avoid excessively long variable names too (do not look to Java for inspiration on how naming is done: `create_table` is fine, `createTheTableWithAllocation` is not). Avoid excessive and uninformative comments: "set x to 0" is not helpful; instead, use comments to explain what the code aims to achieve and, if necessary, why. If you code is so complex that you must explain *how* it works, consider breaking it into simpler components.

[1] We will provide instructions on how to set-up a VM running this OS on canvas.

3. **Source organization**. Break up your code into logical units (as `.cpp` files). If your prior classes did not teach multi-compilation-unit programming, we will cover it in section. Each source file should be well-organized.

   Header files should be minimal; they must contain only the interfaces that you define for your system components. Do not include system header files in your headers, except for those that are necessary for that file. Do not #include `.cpp` files, they must be compiled separately. Header files must use either header guards or (preferably) #pragma once.

4. **Avoid global variables**. Encapsulate or use local variables instead. Imagine creating an implementation of a hash table that uses a global variable for the table itself. Now imagine if you want two hash tables in the same program. Functions must *always* take as arguments any data on which they operate, and classes should define methods that operate on their state. The exceptions to this rule are:

   (a) Global constants (variables declared with `const`).

   (b) "Mostly-immutable" variables (variables set once on startup, perhaps a result of command-line arguments).

5. **Scope**. Variables must be declared in the *smallest* scope possible, and must be declared as close to their use in the function as possible. This means that,
   `size_t foo, i; for (i=0;i<len;i++) { foo = bar(i); use_foo(foo); }`
   is bad form. Instead, it should be:
   `for (size_t i=0;i<len;i++) { size_t foo = bar(i); use_foo(foo); }`

6. **Boolean.** Functions which return boolean values should be declared returning `bool`, not `int` (as is often done).

1.2 Avoiding Mistakes

1. **Compile your code with warnings enabled**. Use the flags: `-Wall -Wextra -Wpedantic -Wshadow`. Your code must compile without warnings on the standardized compiler (and version) using the standardized compile options for the class. In addition to the warning flags above, we will compile your code with `-std=gnu++11 -O2`.

2. **Avoiding bugs**. If your code is buggy and crashes, is unstable, or fails to correctly pass tests, you will lose points. Getting the logic of the program right is, of course, up to you. But you can help track down bugs in your code. Part of this is making sure your code compiles without warnings, but in addition:

   (a) *OPTIONAL* use static analysis tools.

   (b) *OPTIONAL* use run-time tools (*e.g.* the address sanitizer, the undefined behaviour sanitizer, `valgrind`, *etc.*).

   (c) *OPTIONAL* use a debugger (*e.g.* `gdb`, `lldb`).

3. **Standard integer types**. Include `cstdint`, and use the fixed-width integer types for integer variables. These are defined as `intN_t` and `uintN_t`, where the `N` is either 8, 16, 32, or 64, giving you signed and unsigned integers with `N` bits (*e.g.* `uint32_t` is an unsigned 32-bit integer). Some specifics:

   (a) Do **NOT** use basic built-in integer types like `long` and `int`. The only exceptions here are for `int argc`, the return type of `main`, and interacting with functions (parameters and return values) defined by the standard library that take these kinds of arguments (in which case you must match the types specified in the prototype).

   (b) Use the signed versions *only when you need negative values.* This means your variables should be mostly `uintN_t`.

   (c) Variables that refer to sizes must be declared `size_t` (provided by the `cstddef` header). Indexes must be `size_t`, unless you are certain the number will be small (and will fit into another unsigned type).

   For example, a good loop that accesses array elements looks like:
   `for (size_t i=0;i<len;i++) { foo(arr[i]); }`
   Since we're unsure of how big `len` could be, and we are indexing with `i`, we use `size_t`.

   Some functions return `ssize_t` when they may return either a length or an error. If you're storing the result of a function like this (*e.g.* `read`), you should use `ssize_t`.

   (d) Never store pointers in integers. If you need to do integer arithmetic on a pointer, if allowed by the assignment, cast to a `uintptr_t`, never a `long` or an `int`.

(e) For maximizing portability, you should include `cinttypes` and use the provided format specifier macros when passing integers to functions that operate on a `printf`-like format string. These are used in place of "`%lld`", and are formed like `PRI`**`x`**`N`, where `x` is a `printf` format specifier type (like `d` or `x`), and the `N` is the bit width of the variable. For example, use `PRIu64` to print a `uint64_t`, like so:
`printf("x is " PRIu64 "\n", x);`. More information is available here.

4. **Magic numbers**. "Magic numbers" in your code must be specified as named constants, not as bare numbers. These are numbers that have a meaning, occur in one more more locations in the code, and which could be replaced by a named constant to clarify the code and prevent errors later. Some examples include:

   (a) The size of a buffer; prefer using the `sizeof` operator instead of manually specifying the size.

   (b) Possible return values of function to indicate different types of errors or results. Use an `enum` instead.

   (c) Other magic values, like indicators, offsets into files, *etc.*. Use named constants instead.

   Magic numbers hurt readability (and thus grade-ability), and also encourage errors; consider reading data into a buffer. If you change the buffer size, but not the amount of data read, it may be a buffer overflow. Instead, if the read call uses `sizeof(buffer)`, the read call is updated "automatically".

   Exceptions include numbers that have obvious meaning (such as zero, one, *etc.*), or are numbers that have no meaning beyond their value.

5. **Static.** Functions and globals that are not accessed outside of the file they are defined in should be declared `static`.

Note that some of these points may require you to make a second (or third) pass over your code once you have a basic, working solution in order to *clean up* your code. This is expected, and is what you should be doing anyway. Think of it like going from a second draft of an essay to a final draft.

1.3 Some Common Mistakes

There are *many* common mistakes people make, especially in an introductory systems class. This is totally okay; you're here to learn. But we might be able to save you a headache later on with this list. It's not exhaustive, but contains some of the most common mistakes we've seen.

1. **Not checking errors.** Most functions in C return 0 on success or -1 on error (exception: functions which return pointers return NULL on failure). You should check the return value of almost every function you call. When you're using a function, you should check its `man` page and look at the possible errors that can occur and the return values to expect.

2. **Special-case errors.** In general, you should not have separate error handling for each possible error unless you really know you need to. As an example, `open` can fail because of "file not found" or because of "access denied". Both cause open to return -1 and set the global variable `errno` to the error code. They can be handled the same way:
`if(open("foo", O_RDWR) == -1) {fprintf(stderr, "open failed: %s\n", strerror(errno)); exit(1);}`

3. **Storing return values incorrectly.** The function `read` returns a `ssize_t` that indicates the number of bytes read on success and -1 on error. Storing this value to check the error code *must* be put in a *signed* type. If you store it in a `size_` (an unsigned type), a failure will be interpreted as a huge *positive* integer (remember CSE 12!) and will certainly break your code.

4. **Overly-complex string processing.** Instead of building up a string via a combination of `strcat` and `strcpy`, use more advanced string functions. Here are some useful ones: `snprintf, dprintf, asprintf`. Here are some useful string functions for parsing: `strtol, strstr, strtok_r, strchr`.

2 Development Practices

1. Commit your code through `git` regularly, and in small, logical chunks. It's okay to commit things that don't compile or work while you're developing (we won't take off points for this)! Of course, your program must compile to be graded and should work when you submit it.

2. Use well-written commit messages. They should describe what your commit does. For example, "Make server respond only to clients that present correct authorization token", not "asdf".

3. Your repo will contain a file named `.gitlab-ci.yml` in the root. DO NOT delete or modify this file, as it will prevent GitLab@UCSC from auto-checking basic submission requirements, and will result in your assignment not being graded.

3 Documentation

1. As stated above, avoid excessive commenting. However, you should include comments for each part of your interface (for example, each method of your data structure) that explains the arguments and return values, if/when the function may or may not be called, discussions on interactions between parts of your program, and **anything that would help the graders understand your code better**.

2. Each assignment must include a design document, a write-up document, and a readme document. The design document **MUST** be a PDF named `DESIGN.pdf`. The write-up **MUST** be a PDF named `WRITEUP.pdf`. The README file **MUST** be a Markdown file named `README.md`. Note that plain ASCII text is valid Markdown, so you're not *required* to use Markdown tags in your `README.md` file.

3. The `README.md` file must contain your name and email, instructions for building and running your program, known bugs, and any information you want the graders to know ahead of time.

4. Your `DESIGN.pdf` must clearly explain the design of your system. This will include describing any interfaces you create or use, explaining how you're composing them, and how your code achieves the goals of the assignment.

   The design document is what the graders will refer to if your code is unclear, so it behooves you to explain clearly.

5. Each assignment will ask a number of conceptual questions relating to it. Your `WRITEUP.pdf` will contain your answers to the questions posed in each assignment, must be written in complete sentences and be well-thought out. The answers don't have to be long; unless otherwise specified, they should be a few sentences, and show you understand the reasoning. You will not get extra points for too-long answers.