CS137 Style Guide

1. Introduction

Welcome to the CS137 style guide. Style guides are widespread in practice and are often essential when programming in teams of even modest size. This style guide combines many corporate and open-source style guides that were consulted to establish good, industry-supported practices. Any deviations from industry practice are either to reduce the complexity of the guidelines and/or to improve learning outcomes in this introductory course.

Fact: Students have often reported to our course staff that they appreciated the style guide after their co-op placements and that it helped them develop good habits. Unironically, the phrase *"good habits"* sums up many of the learning objectives in this course. Your style and approach will evolve as you become a more experienced programmer, but instilling good habits early in your skill development is essential.

Unfortunately, some students have also developed a sense of "paranoia" regarding their style marks, which can be unhealthy. Therefore, in CS137, there is no hand-mark portion regarding the first three assignments; only feedback will be provided for improvements, but there is a hand-mark for the rest of the assignments. We think programming should be a mostly enjoyable experience; it can be fun, magical, and beautiful when done right. Please do not let this style get in the way of your enjoyment. Remember that our objective is to instill good habits. As a child, you were probably forced to brush your teeth and eat vegetables, which may have been frustrating. As an adult, you may make your own decisions, but we hope you still brush your teeth and eat vegetables.

A few administrative notes before we begin:

- A few coding examples in the course materials deviate from this style guide to condense the content.
- As a reminder, in this course, "variables" include *mutable* variables and *constants*.
- Important guidelines are shown in red and must be followed while coding. We recommend that you also follow the uncolored guidelines, which are not mandatory.

2. Clean Code

2.1. The most subjective measure of code quality is how *clean* it is: readable, understandable, and, if possible, elegant and beautiful.

2.2. You will be halfway to cleaning your code by following the guidelines below.

2.3. The remaining half will improve with experience:

2.3.1. judiciously knowing *when* to and *how* to use helper functions

2.3.2. finding the "Goldilocks zone" (or the "sweet spot") where you do not introduce too many or too few variables

2.3.3. designing your control flow to be logical and straightforward

2.3.4. avoiding duplicate code patterns (copy and paste is your *enemy*, not your *friend*)

2.4. The "golden rule" is to use care when writing code you want other people to write.

3. File Organization

3.1. Your files must be organized into three distinct "regions". From top to bottom, they are:

3.1.1. Header information:

3.1.1.1. any #includes

3.1.1.2. documentation that applies to the entire file [when needed]

3.1.2. Non-function definitions: [when needed]

3.1.2.1. global constants (Global variables are not allowed in this course)

3.1.2.2. structure definitions

3.1.3. Function definitions

3.2. Organize your function definitions so that if function f calls function g, g should appear *before* (above) f in the file. If f and g are *mutually recursive*, then (in order) declare g, define f, and then define g.

3.3. For interfaces (.h files), follow the same organization (with *declarations* instead of *definitions*).

4. Comments and documentation

4.1. Layout

4.1.1. You may use either line comments (//) or block comments (/* */) for any documentation.

4.1.2. Although it's not a *bad style*, it may make your life easier if you use line comments everywhere and reserve block comments when you need to disable large sections of code simultaneously (block comments cannot be nested).

4.1.3. Comments must be aligned horizontally with the code they apply to. Top-level documentation (e.g., function documentation) must begin in the first column. Comments should appear on the line(s) before the code it applies to, but for short explanations, they can be placed to the right of the code with enough white space added to ensure the comment stands out (align such comments when possible).

```
// function documentation
void foo(void) {
  if (some_condition) {
    // this explains the following code
    code_code_code;
  }

  code_code_code;            // short explanation
  more_code_code_code;       // another explanation
```

4.2. Function documentation

4.2.1. The `main` function does not require a purpose or any other documentation. The program's purpose accomplishes that.

4.2.2. **Every function (excluding the `main` function) must have a documented purpose**.

4.2.3. In addition, some functions might need additional documentation (sometimes called the "contract") to describe additional information, such as requirements.

4.2.4. The general layout of function documentation is as follows:

```
// function_name(param1, param2) purpose statement
//   indentation for second line of purpose statement
//   and additional lines
// requires: this is optional. It depends on if the
//           function has any restrictions or assumptions
```

For example:

```
// fibonacci(n) prints the Fibonacci sequence from F_0..F_n
// requires: n >= 0
```

4.3. purpose statement

4.3.1. The purpose statement has two parts:

- a demonstration of how the function is called
- a brief description of what the function does

4.3.2. The purpose statement describes **what** the function does, **not how** it does it.

4.4. requirements section [only if applicable]

4.4.1. List any restrictions or assumptions on the parameters that would prevent the function from behaving as intended.

4.4.2. List any global state requirements that must be satisfied before calling the function (this is rare).

4.4.3. The syntax is informal, intending to be easily understood.

4.4.4. Add "[not asserted]" afterward if a requirement cannot be asserted.

4.4.5. For example:

```
// requires: 0 <= x <= y < 100
//           b is positive
//           n is a prime number [not asserted]
//           s is not empty
//           initialize_map() has been called
```

4.4.6. If a requirement applies to numerous functions in a file and there is no exception to the requirement, a global comment at the top of the file can be added to avoid repetition.

```
// The following applies to all functions:
// requires: all quantity parameters must be non-negative
```

4.5. A practical use of the previous rule would be a global requirement that all pointer parameters must not be NULL.

4.6. Variable documentation

4.5.1. Variables should have **meaningful** names that do **not** require any additional documentation.

```
const int widgets_per_order = 5;
```

4.7. Structure documentation

4.6.1. Like variables, structures, and fields should have meaningful names that do not require additional documentation when possible.

4.6.2. If there are restrictions on the fields, add a requires section after the definition.

```
struct stopwatch {
  int min;
  int sec;
};
// requires: 0 <= min
//           0 <= sec <= 59
```

4.8. Code documentation

4.7.1. If a function:

- uses meaningful variable and parameter names,
- has straightforward and logical control flow, and
- is properly formatted and organized

It should be self-documenting and not require any comments inside the function.

4.7.2. If you believe your function meets the above criteria, you should not feel obligated to add comments to your code.

4.7.3. It is not considered a bad style if you wish to add additional comments to add exposition or break your code into logical steps/sections. However, it is a bad style if your code comments are excessive or verbose in situations where it is not necessary.

4.7.4. If a section of code:

- is of particular note or importance that may not be obvious *to the reader* or
- is potentially confusing or misleading *to the reader*

add a comment to help the reader.

4.7.5. A good "rule of thumb" is: if you think a competent peer would not be able to understand your code or would miss a significant (possibly subtle) detail, either restructure your code (if feasible) or add comments to your code.

5. Line Length

5.1. Limit your code (including comments) to 80 characters per line maximum. Despite the prevalence of widescreen monitors, there are still many good reasons to limit your code width to 80 characters (including accessibility issues for people with reduced vision), and it remains the industry standard.

5.2. For long expressions that extend over multiple lines, end incomplete lines with operators or commas and indent the following line either two spaces or to the matching open parenthesis.

```
// this is a very very very very very very very very very very very very very
//   very very very very very very very very very very very very very very very
//   long comment. Subsequent lines are indented two spaces

return long_variable1 + long_variable2 + long_variable3 + long_variable4 +
  long_variable5 + long_variable6;

return long_variable1 * (long_variable2 + long_variable3 + long_variable4 +
                         long_variable5 + long_variable6);

return long_function_name(long_variable1, long_variable2, long_variable3,
                          long_variable4, long_variable5, long_variable6);
```

5.3. In the rare situation where you need to have a string literal that extends beyond 80 characters, you can split the string literal across multiple lines by closing the double quotes (") on one line and "re-opening" them on the following line.

```
printf("this is a really really really really really really really really "
        "long string: don't forget the space otherwise you get reallylong\n");
```

6. Identifiers (Names)

6.1. use ALL_CAPS for constants when the value of the constant is either:

- meaningless and arbitrary (also sometimes known as a "symbol")
- a software limitation (that is often arbitrary or not obvious)

but if a constant has a meaningful value, continue to use lowercase.

```
const int days_in_leap_year = 366;
const int slices_per_pizza = 8;

const int NORTH = 2;
const int EAST = 3;
const int MAX_QUEUE_LENGTH = 100;
```

6.2. Do not call a helper function "helper". Instead, find a meaningful name that distinguishes it from the function it is helping.

6.3. Do not have multiple variables with the same name in nested scopes. For example, global and local variables do not have the same name. Similarly, do not have variables/parameters with the same name as struct (or a field of a struct).

6.4. Simple parameter names such as n, x, y, z are acceptable for straightforward mathematical functions.

6.5. For loop counters, using single variable names such as i, j, k is perfectly fine.

6.6. For functions with a single array parameter, the names a (for *array*) and len (for the array *length*) are acceptable variable names when they occur in sequence and are obvious array parameters.

```
int sum_array(int a[], int len) {
```

7. Magic numbers

7.1. Do not have *magic numbers* in your code.

7.2. A magic number is a number that appears in your code, and it is not clear where the number came from or what the number represents. If you are unfamiliar with the "magic number" terminology, you can read about them more on wikipedia.

7.3. You may use magic numbers in your testing code.

```
assert(sum_first(11) == 66);
```

7.4. Use a constant instead of a magic number to give your code meaning and context. Do not assign a constant with the name of a number to avoid using a magic number.

```
const int num_dwarves = 7;
```

7.5. The following are generally *not* considered magic numbers:

- The numbers 0, 1, and -1, mainly when used in iteration/recursion or when calculating offsets. This does not apply if the value has a special meaning (e.g., const int max_password_attempts = 1;).
- The number 2 when checking if a number is *even* (e.g., % 2).
- The powers of 10 in most applications (e.g., using % 10 to extract the "ones digit" or using 100 in a percentage calculation).
- Coefficients in formulas where the number has no special meaning (e.g., 3 in the Collatz function: 3 * n + 1).

7.6. Characters are generally not considered magic numbers. The exception is if there would be a more meaningful name that could be used.

```
const char MOVE_UP = 'w';
```

7.7. Do not use the decimal ASCII values for characters.

```
if (c == ' ' || c == '\n') {
```

7.8. You may use a magic number as the length of an array in a *definition*. However, outside of the definition, you should use a constant to represent the array's length.

```
int a[13] = {0};
const int a_len = 13;
```

8. Block indentation

8.1. For indentation, two spaces are recommended, but 3 or 4 are allowed: you **must** be consistent.

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

9. Whitespace

9.1. Vertical whitespace (blank lines)

9.1.1. At least one blank line is *required* between functions. If you have a function with a blank line between the documentation and the definition, use two blank lines before and after the function.

9.1.2. A blank line is never required inside of a function.

9.1.3. Blank lines inside of a function may be helpful to break a function into logical "sub-sections", and a blank line before a comment may help it stand out. Avoid excessive vertical whitespace inside of a function (e.g., two blank lines in a row or code that is "double spaced" with blank lines occurring every other line).

9.2. Horizontal whitespace (in-line)

9.2.1. Add a space around arithmetic, comparison, logical, and assignment operators.

```
if (y > x + 1 || x * x == z) {
  y += 1;
```

9.2.2. The exceptions to the previous rule are the increment and decrement operators, which should not have a space between the variable and the operator.

```
++x;
```

10. Scope

10.1. Variable definitions

10.1.1. Global *mutable* variables are a bad style, and should be avoided in general. They are NOT ALLOWED in CS137.

10.1.2. Global *constants* are encouraged and good style.

10.1.3. Generally, if a constant is only needed for one function, it is a good style to define it locally. However, this rule is easily broken if:

- It is more convenient to define it at the start of the file (e.g., so it can be changed easily).
- It is more cohesive to define it at the start of the file (e.g., to group many related constant definitions).
- The value of the constant needs to be known by the reader (e.g., to draw attention to a constant by placing it at the start of the file).

10.1.4. A local variable should be defined once its initial value is known and close to where it is first referenced/needed.

10.1.5. An *older* popular style (and a requirement in earlier versions of C) is to define all local variables at the start of the function. It is not considered a bad style to follow this style.

10.2. Modular scope

10.2.1. Helper functions and global variables not provided in a module interface must have modular scope. In other words, all non-interface functions and global variables should be defined as static.

11. Const

11.1. Local variables and parameters that are never mutated should be made constants (const). This is beneficial because:

- it communicates the intended use of the variable
- it prevents "accidental" or unintended mutation
- it may help for low-level optimization of the code

11.2. Unfortunately, the course notes and examples do not always follow this practice. This is to keep the slides uncluttered and to avoid drawing focus away from the intended learning objective.

11.3. For pointer parameters to arrays and structures, add const to the pointer parameter if the function does not mutate (modify) the array/structure.

12. Miscellaneous

12.1. Default types

12.1.1. Function definitions must define the function's return type and all parameters' type. It is a bad style to rely on C's behaviour to "default" to int when a type is unknown.

12.2. Conditional operator

12.2.1. The conditional (ternary) operator (?:) may be used *sparingly*. Overuse of the conditional operator can make your code hard to follow.

12.2.2. Outside of this course, the conditional operator is considered a good style when it allows a variable to be const when it would otherwise need to be mutable.

```
// This example is not colour-coded as it is beyond this course:
const int max = a > b ? a : b;   // good

int max = a;                     // fine in this course,
                                 //   but not as good in practice
if (b > a) {
  max = b;
}
```

12.3. Order of operations

12.3.1. Add parenthesis to clarify the order of operations.

```
if ((is_hungry && is_cranky) || (is_sleepy && (time >= (bed_time - time_to_brush
)) && is_dark_out)) {
```

12.3.2. A good "rule of thumb" is to add parenthesis if a typical reader has to stop and think carefully about the order of operations (or possibly consult a table).

12.3.3. Parenthesis is especially important with non-arithmetic operators, but adding parenthesis to arithmetic expressions can reduce the time required to understand the code and clarify the calculation (e.g. when integer division occurs).

```
int num_boxes = (order_size * units_per_order) / units_per_box;
```

12.4. Pointers

12.4.1. If the value of a pointer becomes invalid *and* the pointer variable will remain in scope for some time, assign the value of NULL to the pointer.

```
free(p);
p = NULL;
```

12.5. Modules and declarations

12.5.1. In *declarations*, use the same parameter names as in the definition.

```
int my_add(int a, int b);
```

12.5.2. In *declarations*, only pointer parameters should ever be const. In other words, simple (non-pointer) parameters (e.g., int) should not be *declared* as const parameters in an interface.

12.5.3. An implementation (.c file) should always #include its interface (.h file) to ensure no discrepancies.

12.6. malloc

12.6.1. Always use sizeof when providing an argument to malloc, even when allocating an array of char.

```
ms = malloc(sizeof(struct mystruct));
s = malloc((len + 1) * sizeof(char));
```