CS 161/162
C/C++ Programming Style Guidelines

Style guidelines exist to help us write code that is easy to read, understand, debug, and maintain. By following these guidelines, you will gain useful skills for future collaboration. You will also be thankful to your past self when you go back to code you wrote a month or a year ago and you can still understand what it does!

1. Comments

File headers

Programs should include a program header at the top of every file, such as:

```
/*********************************************************************
 ** Program Filename:
 ** Author:
 ** Date:
 ** Description:
 ** Input:
 ** Output:
 *********************************************************************/
```

Function headers

Each function should be preceded by a headers block, which describes the purpose, usage, and type of any parameters, as well as any pre-conditions and post-conditions. A pre-condition is a condition that must hold *prior* to beginning the function, and post-conditions are conditions that must hold *upon exiting* the function. As code is updated, these may change, and must be updated accordingly.

```
/*********************************************************************
 ** Function:
 ** Description:
 ** Parameters:
 ** Pre-Conditions:
 ** Post-Conditions:
 *********************************************************************/
```

Explanatory comments

These comments describe what the code does in English. They should precede each major section of code and occur inline whenever needed. General guidelines:

- Comments exist to supplement, not re-state, what is in the code. Avoid writing comments that are already obvious from the code.

- Comments must be consistent with the code. This also means that if you change the code, you may need to change the comments to maintain consistency.
- If someone glancing at your code would think, "What does that do?" then you should add a comment explaining it.
- Short comments should be *what does it do?* comments, such as "`compute mean value`," rather than *how does it do it?* comments such as "`sum of values divided by n`" which literally restates what the code does.
- Putting a comment at the top of a 3-10 line section describing its overall goal is often more useful than a comment on each line describing micrologic.

Longer comments that describe algorithms, data structures, etc., should be in block comment form.

```
/*
 * Here is a block comment.
 * The comment text should be tabbed or spaced over uniformly.
 * The opening slash-star and closing star-slash are alone on a line.
 */

/*
** Alternate format for block comments
*/
```

Block comments inside a function are appropriate, and they should be spaced over to the same indentation setting as the code that they describe. One-line comments alone on a line should be indented to the same indentation of the code that follows.

```
int main() {

  float radius;

  /* Read radius value from user. */
  cout << "Enter the circle's radius value: ";
  cin >> radius;

  return 0;
}
```

Very short comments may appear on the same line as the code they describe, and they should be separated by some whitespace from the statements.

```
if (a == EXCEPTION) {
  b = true;          /* special case */
} else {
  b = isprime(a);    /* works only for odd a */
}
```

2. Whitespace

Use vertical and horizontal whitespace generously to make the code easy to read. Treat code blocks like paragraphs in an essay: group statements together that conceptually go together to achieve a goal, and separate groups with whitespace to indicate that something new is happening.

Indentation and spacing should reflect the block structure of the code; e.g., there should be at least 2 blank lines between the end of one function and the comment header for the next function. For indentation, you can use either tabs or spaces, but be consistent in your choice. If you choose to use spaces, then I suggest you use at least 2 spaces for indenting because it is hard to see only 1 space.

It is a good idea to have spaces after commas in argument or variable lists and after semicolons in `for` loops to separate the arguments/variables and statements visually.

```
int length, width;
int calculate_rectangle_area(int length, int width) { …
for(i = 0; i < MAX_SIZE; i++) { …
```

Operators should be surrounded by a space. For example, use

```
z = x + y
```

rather than

```
z=x+y
```

This greatly enhances readability and makes it significantly easier to spot operators within an expression. Prefix and postfix increment and decrement (e.g., `x++`) are not considered operators in this context.

3. Variable Declarations

Related declarations of the same type can be on the same line, but you should put unrelated declarations of the same type on separate lines. A comment describing the role of the item being declared should be included, with the exception that a list of #defined constants does not need comments if the constant names are sufficient documentation. The names, values, and comments are usually indented so that they line up underneath each other. Use spaces rather than tabs for alignment.

The "pointer" qualifier, '`*`', should occur with the variable name rather than with the type.

```
char *s, *t, *u;
```

declares three `char*` pointers, while

```
char* s, t, u;
```

declares `s` as a `char*` pointer but `t` and `u` as `char` only (not `char*`).

4. Compound Statements

A compound statement is a list of statements enclosed by braces. There are many common ways of formatting the braces. Pick one and use it consistently. When editing someone else's code, always use the style used in that code.

Two common styles:

```
CONTROL {
  STATEMENT1;
  STATEMENT2;
}

CONTROL
{
  STATEMENT1;
  STATEMENT2;
}
```

When a block of code has several labels, the labels are placed on separate lines. The fall-through feature of the C/C++ switch statement (that is, when there is no break between a code segment and the next case statement) must be commented to show that this is an intentional choice, not a forgotten `break` statement.

```
switch(EXPR)
{
  case A:         /* Fall through to case B */
  case B:
    STATEMENT;
    break;
  case C:
    STATEMENT;    /* Fall through to case D */
  case D:
    STATEMENT;
    break;
  default:
    STATEMENT;
}
```

It is good practice to include a `default` case to handle anything not covered in the preceding cases. It does not require a break if it is last.

5. Naming Conventions (for variables, functions, etc.)

Individual projects will no doubt have their own naming conventions. However, there are some general rules.

- Names with leading and trailing underscores (e.g., `_var_`) are reserved for system purposes and should not be used for any user-created names. Function, typedef, and variable names, as well as struct, union, (C++) class, and `enum` tag names should be in lower case, with words separated by an underscore (e.g., `variable_name`).
- Avoid using names in the same program that differ only in case, like `foo` and `FOO`. Similarly, avoid using highly similar variants such as `foobar` and `foo_bar`. The potential for confusion is considerable. Similarly, avoid names that look like each other. On many terminals and printers, the lower-case letter `l`, the number `1`, and the capital letter `I` look very similar.
- `#define` constants should be in all CAPS.
- `enum` constants should be in all CAPS.

6. Constants

Numerical constants generally should not be used directly in the code (this is called "hard-coding" a value). The `#define` feature of the C/C++ preprocessor should be used to give constants meaningful names (and enable them to be updated in one location). Constants with names, like `MIN_SCORE`, make the code easier to read. The enumeration data type is a better way to declare variables that take on only a discrete set of values, since additional type checking is often available. Some specific constraints:

- Constants should be defined consistently with their use; e.g. use 540.0 for a `float` instead of 540 with an implicit `float` cast.
- There are some cases where the constants 0 and 1 may appear as themselves instead of as defines. For example, if a for loop indexes through an array, then

  ```
  for (i = 0; i < ARRAY_LIMIT; i++)
  ```

  is reasonable.

- Always compare pointers to `NULL`, rather than 0.
- Even simple boolean values like 1 or 0 are often better expressed by defining `TRUE` and `FALSE` for later use in the code.

7. Line Length

Lines should be no longer than 80 characters. Any lines longer than this will wrap on many terminals,and reduce readability.

A long string of conditional operators should be split onto separate lines. When separating a statement or conditional operators, make sure the separation is logical and readable. For example:

```
if (foo->next == NULL && totalcount < needed
&& needed <= MAX_ALLOT &&
server_active(current_input)) {
  ...
}
```

is easier to read as

```
if (foo->next == NULL
    && totalcount < needed
    && needed <= MAX_ALLOT
    && server_active(current_input))
{
  ...
}
```