Welcome to the style guide. Style guides are very common in practice and are often essential when programming in teams of even a modest size. This style guide is a combination of 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 back to CS 136 staff that they appreciated the CS 136 style guide *after* their co-op placements, and that this style guide helped them to develop good habits. Unironically, the phrase *"good habits"* sums up many of the learning objectives in this course. Your personal style and approach will evolve as you become a more experienced programmer, but instilling good habits early in your skill development is essential.

This document may seem overwhelmingly detailed. In our experience, a subset of students are frustrated that they are graded on their style in any capacity. They often object that the marking is too subjective or the style guidelines are too ambiguous. To help alleviate this frustration, we have provided this detailed document so that both students and markers have the same frame of reference.

Unfortunately, we have also seen some students develop a sense of "paranoia" regarding their style marks that can be unhealthy. If you find yourself frequently second guessing your style decisions, or if this style guide is making your programming experience miserable, give yourself permission to step away and take a break from this guide. Remember, style marks in this course are mostly inconsequential (they do not count for a large portion of your grade). Programming should be a mostly enjoyable experience and when it's done right it can be fun, magical and beautiful. Please do not let this style get in the way of your enjoyment. When you are ready to revisit this guide, remember that our objective is to instill good habits. When you were a child you were probably forced to brush your teeth and eat your vegetables and it may have been frustrating. As an adult, you may make your own decisions, but we hope that you still brush your teeth and eat your vegetables.

A few administrative notes before we begin:

- In the rare circumstance where a new style guideline is introduced or reworded after the academic term has started, it will be identified with a NEW or reworded tag and if it is significant, it will mentioned in a discussion post or announcement.
- Many guidelines are identified by the **Section (§)** in the course notes that the guideline relates to (or comes into effect). This is to help identify portions that you may skip over at the beginning of the term and revisit later when you understand the content more. Sections that are not yet in effect are "greyed out", and sections that recently came into effect will be highlighted. The Section corresponds to the Section **in the course notes**, *not* the assignment number.
- A few of the coding examples in the course materials deviate from this style guide to keep the content more condensed.
- As a reminder, in this course "variables" includes both *mutable* variables and

constants.

- For coding examples, good examples are shown in green with a solid border, and bad examples are shown in red with a dashed border.

```
// Good Style
```

```
//BadStyle
```

1. **Clean Code**

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

   1.2 By simply following the guidelines below, you will be half way to making your code clean.

   1.3 The remaining half will improve with experience:

   - judiciously knowing *when* to and *how* to use helper functions
   - finding the "goldilocks zone" (or the "sweet spot") where you do not introduce too many or too few variables
   - designing your control flow to be logical and straightforward
   - avoiding duplicate code patterns (copy and paste is your *enemy*, not your *friend*)

   1.4 The "golden rule" is that you should use care to write code that you would like to see other people write.

2. **Identification**

   2.1 Fill out the integrity statement with your full name and login ID (i.e., your username such as j42smith, *not* your student number). That is the only personal identification needed.

   2.2 You do **not** need to add an additional header and you do **not** need to identify any additional personal information, the course, assignment number or question number.

3. **File Organization**

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

   - header information:
     - a brief purpose for the file
     - integrity statement [only when provided]
     - any `#includes`
     - documentation that applies to the entire file [when needed]
   - non-function definitions: [when needed]

        - global variables
        - structure definitions
    - function definitions

3.2 The order of items within each region is not important, but items must be grouped together in an organized and logical manner (see the section on whitespace below).

3.3 Organize your function definitions so that if function `f` calls function `g`, then `g` should appear *before* (above) `f` in the file. (§ 5) If `f` and `g` are *mutually recursive*, then (in order) declare `g`, define `f`, and then define `g`.

3.4 (§ 5) For interfaces (`.h` files) follow the same organization (with *declarations* instead of *definitions*).

4. **Comments and documentation**

4.1 **Layout**

4.1.1 You must use line comments (`//`) for function documentation, but for any additional documentation you may use either line comments or block comments (`/* */`)

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

4.1.3 For line comments, add at least one space after the two slashes.

```
// at least one space before
```

```
//squished
```

4.1.4 Comments must be aligned horizontally with the code it applies to. For top-level documentation (e.g., function documentation) it must begin in the first column. Comments should appear on the line(s) before the code it applies to, but for short explanations it can alternatively 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
```

```
     // explanation
code_code_code;

code_code_code;
// explanation of previous code

code_code_code;// short explanation
```

4.2 **File (program) documentation**

4.2.1 Every file must have a brief purpose that describes the contents of the file.

4.2.2 There is no formal syntax for this purpose, and a casual sentence or two is sufficient.

```
// This program reads in integers from input
//   and prints them out in reverse order
```

```
// This is a testing client for the killer_robot
module
```

4.3 **Function documentation**

4.3.1 The `main` function does not require a purpose or any other documentation. That is accomplished by the program purpose.

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

4.3.3 In addition, most functions will need additional documentation (sometimes called the "contract") to describe additional information, requirements, (§ 2) side effects and (§ 7) efficiency information.

4.3.4 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
// section: this would be some additional
documentation
//          note that sections are in lower case
//          and are indented if there are multiple
lines
// another_section: this is additional documentation
of a different nature
//                  it has slightly different
indentation
```

4.3.5 For example:

```
// fibonacci(n) prints the Fibonacci sequence from
F_0..F_n
// note: because it starts at F_0, n + 1 items will
print
// examples: fibonacci(0) => 0.
//           fibonacci(10) => 0, 1, 1, 2, 3, 5, 8,
13, 21, 34, 55.
// requires: n >= 0
// effects: produces output
// time: O(n)
```

4.3.6 **purpose statement**

4.3.6.1 The purpose statement has two parts:

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

4.3.6.2 The demonstration is the function name with the parameters names as argument values *with no type information* (the type information is in the definition).

```
// fibonacci(n) ...
```

```
// int fibonacci(int n) ...
```

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

```
// fibonacci(n) loops from 0..n, storing the
previous two values
//   in variables to make the calculation
more efficient
```

4.3.6.4 Do not write the word "purpose".

```
// Purpose: fibonacci(n) ...
```

4.3.6.5 Avoid beginning the purpose statement with *"returns..."*. If possible, use an active verb such as *computes, finds, determines, analyzes, creates, searches, etc.*.

4.3.6.6 When feasible, use the parameter names in the purpose to demonstrate how each parameter is used. Similarly named parameters may be grouped together to avoid redundancy.

```
mean(x1, x2, x3, x4, x5) calculates the
average of x1..x5
```

4.3.6.7 If there are non-obvious units associated with a parameter, provide them (this might be a good use of the *notes* section below).

4.3.6.8 A single-sentence purpose statement does not require a period for punctuation, but a multiple-sentence statement does.

4.3.6.9 If you find there is too much exposition in your purpose statement or it is too wordy, consider adding a *note(s)* section.

4.3.7 **note(s) section [optional]**

4.3.7.1 You may specify additional details that do not fit anywhere else in the function documentation in a section called *"note"* or *"notes"*.

4.3.7.2 Adding a notes section allows you to keep your purpose statement more concise.

4.3.8 **example(s) section [optional]**

4.3.8.1 Sometimes an example is the most effective way to communicate how a function behaves.

4.3.8.2 Examples are **never** required.

4.3.8.3 The syntax is informal, with the aim of being easily understood.

4.3.9 **requirements section [only if applicable]**

4.3.9.1 List any restrictions on the parameters that would prevent the function from behaving as intended.

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

4.3.9.3 The syntax is informal, with the aim of being easily understood.

4.3.9.4 If a requirement cannot be `asserted`, add "`[not asserted]`" afterward.

4.3.9.5 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
// requires: i is a valid integer
//           a newline must be printed
//           the return value is positive
```

4.3.9.6 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
```

4.3.9.7 (§ 4) A practical use of the previous rule would be a global requirement that all pointer parameters must not be `NULL`.

4.3.10 **(§ 2) effects section [only if applicable]**

4.3.10.1 List any side effects that the function causes.

4.3.10.2 You do not need to provide great detail in the effects section. If the main purpose of the function is to produce a side effect, and it is important to provide more detail, that detail can be described in the purpose or a note.

4.3.10.3 Most side effects can be accurately and succinctly described as:

- `produces output`
- `reads input`
- `modifies X`
- (§ 7) `allocates memory`

4.3.10.4 If a side effect does not always occur, add the qualifier word *may* to the side effect (e.g., `may produce output`).

4.3.10.5 Using tracing tools (e.g., `trace_int`) is not considered a side effect (for this course).

4.3.10.6 Only list side effects that affect state *outside* of the function.

- Mutating a local variable within a function is not a side effect of that function.
- (§ 7) Allocating memory and then subsequently freeing that memory before the function returns is not a side effect of that function.

4.3.10.7 (§ 7) For ADT functions (operations), only list side effects that are relevant to the client.

- In the `create` function, indicate that the `destroy` function must be called.

```
// foo_create() creates a new foo ADT
// effects: allocates memory (client
must call foo_destroy)

// foo_destroy(foo) removes memory for
foo
// effects: foo is no longer valid
```

- Allocating (or re-allocating) memory to store data within the ADT is not a side effect that is relevant to the client.

```
// foo_add(item) adds item to foo
// effects: allocates memory
```

- Changing the state of the ADT is a side effect.

```
// foo_add(item) adds item to foo
// effects: modifies foo
```

4.3.11 **(§ 6) time section**

4.3.11.1 State the efficiency if it is not $O(1)$ or if it is $O(1)$ and not obviously $O(1)$.

4.3.11.2 Use n (e.g., $O(n)$) when there is only one element that affects the efficiency.

4.3.11.3 If there is only one array parameter, then it is implicit that n is the length of the array. (§ 8) For example, if there is a single string parameter, it is implicit that n is the length of the string.

4.3.11.4 Specify what n is if there is any ambiguity as to how n is measured.

4.3.12 (§ 5) If function documentation is provided in the *interface* (.h file) do NOT duplicate the documentation in the *implementation*. Having duplication is never a good idea, as changes in one source may not always be reflected in the other. Only add additional documentation to the implementation that may not be appropriate for the interface (e.g., hidden side effects, additional usage notes, efficiency analysis).

```
// see myfile.h for details
// effects: modifies some_global_state_var
```

4.4 **(§ 2) Variable documentation**

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

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

```
// x is the number of widgets per order
const int x = 5;

// number of widgets per order
const int widgets_per_order = 5;
```

4.4.2 In the (very) rare situation where a meaningful name may be ambiguous, document the name.

```
// gravitational constant g
const double g = 9.81;
```

4.4.3 If there are non-obvious units associated with a value, provide them.

```
const double g = 9.81;      // (m/s^2)
```

4.5 (§ 3) Structure documentation

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

4.5.2 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
```

4.5.3 If there are non-obvious units associated with a field, document them as you would a variable.

4.6 Code documentation

4.6.1 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* of the function.

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

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

```
// add one to the variable count
count += 1;

// loop i from 0..n-1
for (int i = 0; i < n; ++i) {
```

4.6.4 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.

4.6.5 A good "rule of thumb" is: if you think a competent peer would not be able to understand your code or would miss an important (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 a maximum of **80 characters per line**. Despite the commonplace 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 is still the industry standard. Note: edX shows a vertical bar at 80 characters and subtly highlights your code when it exceeds 80 characters.

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
//   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 (§ 2) In the rare situation where you need 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 `underscore_style` (also known as `snake_case`) in lower case for all identifiers: functions, structures, variables and parameter names.

```
int process_widgets(int num_widgets, int widgets_per_order)
{
```

```
int Process_Widgets(int numwidgets, int widgetsPerOrder) {
```

6.2 (§ 2) The exception to the above rule is: 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 lower case.

```
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;
```

6.3 Do not call a helper function "`helper`". Find a meaningful name that distinguishes it from the function it is helping.

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

6.5 For straightforward mathematical functions, simple parameter names such as `n, x, y, z` are fine.

6.6 (§ 3) For loop counters, it is perfectly fine to use single variable names such as `i, j, k`.

6.7 (§ 4) For functions with a single array parameter, the names `a` (for *array*) and `len` (for the array *length*) are fine variable names when they occur in sequence and are obvious array parameters.

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

7. **(§ 2) 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 be 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.

```
if (x > 13) {
  if (c == 32) {
    printf("%d", n - 21);
  }
}
```

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, simply to avoid using a magic number.

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

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

- The numbers 0, 1 and -1, especially when used in iteration/recursion or when calculating offsets. This does not apply if the value has 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 (§ 3) Characters are not normally considered magic numbers. The exception is if there would be a more meaningful name that could be used.

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

```
const char char_w = 'w';
const char space = ' ';
```

7.7 (§ 3) Do not use the decimal ASCII values for characters.

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

```
if (c == 32 || c == 13) {
```

7.8 (§ 4) You may use a magic number as the length of an array in a *definition*. Outside of the definition, you should use a constant to represent the length of the array.

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

8. Block indentation

8.1 There are many block indentation styles and many different opinions on them (there is even a Wikipedia page). The variant we use for this course is known as the "One True Bracket Style (1TBS)".

8.2 The key characteristics of this indentation style are:

- A block start (open brace `{`) appears at the end of a line after a single space.

- A block end (close brace `}`) is vertically aligned with the start of the statement that opened the block.
- A block end appears on a line by itself (or is followed by an `else`).
- Conditionals (`if`) and loops (`for`/`while`) use blocks (with braces) even if there is only a single conditional or loop statement.

```
void print_polarity(int n) {
  if (n > 0) {
    printf("positive\n");
  } else if (n < 0) {
    printf("negative\n");
  } else {
    printf("zero\n");
  }
}
```

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

8.3 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);
  }
}
```

```
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 A blank line is *required* between the different "regions" of code and any sub-regions (see file organization above). For example, add a blank line between the `#include` statements and any global constants. Two lines or a more decorative comment (e.g., `////////////////////`) may be used if the regions are large.

9.1.2 A single blank line is *optional* between the function documentation and the function definition. A good rule of thumb is to only add a blank line after the function documentation when the function and/or the documentation is very long.

9.1.3 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.

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

9.1.5 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).

9.1.6 (§ 5) In an interface, there is no need to add a blank line between the function documentation and the function declaration.

9.2 **Horizontal whitespace (in-line)**

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

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

```
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;
```

```
++ x;
```

9.2.3 Add a space around the equal sign (=) in an initialization.

```
int count = 0;
```

```
int count=0;
```

9.2.4 Add a space after commas in function calls/definitions.

```
printf("%d\n", pow(2, 3));

int pow(int base, int n) {
```

```
printf("%d\n",pow(2,3));

int pow(int base,int n) {
```

9.2.5 Add a space after semi-colons in `for` loops.

```
for (int i = 0; i < n; ++i) {
```

```
for (int i = 0;i < n;++i) {
```

9.2.6 Add a space after the conditional (`if`) and loop keywords (`for`/`while`) to help distinguish them from function calls.

```
if (condition) {
  while (i < n) {
```

```
if(condition) {
  while(i < n) {
```

9.2.7 Add a space before and after `else`.

```
} else {
```

```
}else{
```

10. **(§ 2) Scope**

10.1 Variable definitions

10.1.1 Global *mutable* variables are bad style, and should be avoided.

10.1.2 Global *constants* are encouraged and good style.

10.1.3 Only have one variable definition per line.

```
int x = 0;
int y = 0;
```

```
int x = 0, y = 0;
```

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

- it is more convenient for it to be defined 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 together many related constant definitions).
- it is important for value of the constant to be known by the reader (e.g., to draw attention to a constant by placing at the start of the file).

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

10.1.6 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 bad style to follow this style.

10.1.7 Variables must be initialized. If its initial value is unknown when it is defined, either delay its definition until its initial value is known or (if that is infeasible) assign a value of zero(s) or `NULL`.

10.2 (§ 5) 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. (§ 2) 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 (§ 4) 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 return type of the function and the type of all parameters. It is 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 You are never required to use the conditional operator.

12.2.3 Outside of this course, the conditional operator is considered 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 **Functions with no parameters**

12.3.1 Use `void` to indicate a function has no parameters.

```
int foo(void) {
```

```
int foo() {
```

12.4 **Order of operations**

12.4.1 Add parenthesis to clarify the order of operations.

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

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

12.4.2 A good "rule of thumb" is: Add parenthesis if a typical reader would have to stop and think carefully about the order of operations (or possibly consult a table).

12.4.3 Parenthesis are especially important with non-arithmetic operators, but adding parenthesis to arithmetic expressions can reduce the amount of 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;
```

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

12.5 (§ 2) Side effects

12.5.1 When possible, only have **one side effect per statement**.

12.6 (§ 4) Pointers

12.6.1 Use the pointer definition style in the course notes.

```
int *p = &i;
```

```
int * p = &i;
int* p = &i;
```

12.6.2 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;
```

12.7 (§ 5) Modules and declarations

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

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

```
int my_add(int, int);
```

12.7.2 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.

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

12.7.3 An implementation (`.c` file) should always `#include` its own interface (`.h` file) to ensure there are no discrepancies.

12.8 (§ 6) malloc

12.8.1 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));
```