# CSE 13S Coding Standards

Prof. Darrell Long & Assistants

Winter 2022

It is cold in the scriptorium, my thumb aches. I leave this manuscript, I do not know for whom; I no longer know what it is about: *stat rosa pristina nomine, nomina nuda tenemus.*

—Adso of Melk (Umberto Eco, *Il nome della rosa*)

## 1 Introduction

It is absolutely vital to have coding standards in order to manage complexity since it is much harder to maintain and understand code that is inconsistent and lacks documentation. Code that is unintuitive or lacks obvious meaning requires time to understand. This impairs the process of adding on to the same code later on, wasting time on merely deciphering what the original code did. This document defines the coding standards and guidelines that will be used for CSE 13S. **To get full points on your assignments, the coding standards must be followed completely.** Your assignments are expected to compile using `clang` and must be formatted using `clang-format`.

`clang-format` is a tool that formats source code according to a set of guidelines placed in a file, referred to as the *format file*. The command to format all source code and header files in the *current working directory* is given as follows:

```
$ clang-format -i -style=file *.{c,h}
```

The `-i` option indicates that files should be modifed in-place. The `-style=file` option tells `clang-format` that it should use guidelines specified in a format file named `.clang-format`. `clang-format` will search for this format file first in the current working directory, then the parent directory. The course format file will be added to the root of your course GitLab repositories, so `clang-format` will be able to find the format file if the above command is run in any of the assignment-specific directories. The last argument, `*.c,h`, tells `clang-format` to format anything that ends in either `.c` or `.h`.

More in depth information about the **C** programming language can be found in the books *The C Programming Language, Second Edition* by Brian Kernighan and Dennis Ritchie (often called K&R or Kernighan and Ritchie) and *C in a Nutshell, Second Edition* by Tony Crawford and Peter Prinz.

## 2 General Principles

*I have deep faith that the principle of the universe will be beautiful and simple.*

—Albert Einstein

1. *Simplicity* and *clarity* should be your goals; do not try to be overly clever.
2. *Global variables* are generally discouraged; you may use them only if there is a *very good reason* to do so, such as sharing variables between files, storing "constants" like command line inputs, or making code clearer if it's being obfuscated by an enormous number of parameters being passed from function to function just to avoid the use of global variables. For the purposes of grading in this course, **global variables are *prohibited* unless otherwise stated.**
3. *Static variables* outside of functions are slightly less proscribed than global variables since their visibility is confined to the file where they are defined; that being said, use them only if there is a *very good reason* to do so, such as making a value persist across function calls.
4. The *scope* of a variable should be kept as *small* as possible: defined and used locally.
5. Loop control variables (`for` loops) should be defined in the statement so that you are not tempted to rely on their final value.
6. Variable names, function names, *etc.*, must be sensible and reflect their purposes. Avoid excessively long variable names. The standard variable naming convention for this class will be *snake case* (more on this later).
7. *All* functions require the `return` keyword, even if the function return-type is `void`.
8. *All* functions require a block comment before it. This block comment should describe the function's purpose, its return value, and details about the parameters that it accepts. An example of a block comment is supplied in §9.
9. Avoid excessive and uninformative comments: "set x to 0" is not informative, unless there is an unclear reason why it needs to be set to 0. Comments should explain what the goal is and, if necessary, why. If your code is so complex to the point that an explanation is needed for why it works, consider refactoring it into smaller components. At the same time, *do not* artificially chop it into smaller pieces.
10. Indentation should be done using *four* spaces. **Do not use tabs.** Most, if not all, text editors support converting tabs to spaces, as well as setting the default tab-width.
11. Line width should be limited to 100 characters. This avoids having excessively long lines of code and forces you to be more clear and concise.
12. Line breaks, or blank lines, in code should be used to separate lines of code to elucidate which lines of code are related or are used in conjunction for some purpose.
13. Above all else, be *consistent* in your coding style.

## 3 Compilation, External code, and Makefiles

All assignments are expected to compile using clang on your Ubuntu 20.04 (or later) virtual machine with the following compiler flags: -Wall, -Wextra, -Wpedantic, and -Werror. This set of compiler flags is commonly referred to as the "take no prisoners" compiler flags. -Wall, -Wextra, and -Wpedantic all enable additional warning flags during compilation. -Werror tells the compiler to treat warnings as errors so as to halt compilation in the event that a warning is issued. So, using this set of flags ensures that all possible warnings are dealt with, which is a must for writing robust code.

Code must compile without warnings or errors, return no scan-build errors, and pass a valgrind leak check to recieve full credit. Code obtained from external sources such as stackoverflow.com must be cited using a comment block placed at the top of the file and before the code. The comment must specify the copied lines, the original author, and a link to the source. You will not recieve credit for code that is not your own. Functions and interfaces provided by the **C** standard library may be used unless otherwise specified by the instructors and/or assignment.

A Makefile must be included with each assignment submission (except assignment 0) and should contain the following targets. Additional targets can be included but should be detailed in an assignment's README.

1. all: The default target that builds the program.
2. clean: Removes build artifacts, such as *.o files, and the executable.

You will given a template Makefile for assignment 1. It then becomes your responsibility to submit a proper Makefile for each subsequent assignment. Make sure to attend laboratory section to learn more about Makefiles.

## 4 scan-build

scan-build is a command line utility which allows you to run the clang static analyzer on your code. The static analyzer finds potential bugs and errors during program compilation. You will need a Makefile in order to use scan-build. To invoke scan-build, run the following:

```
$ scan-build make
```

Your code should pass scan-build without errors. That said, scan-build may report false positives: errors that are not actually errors. These false positives should be reported in the assignment README.md accompanied with an explanation why the false positives truly are false positives. Invalid explanations will not be accepted.

## 5 Formatting and Naming

*In hanc utilitatem clementes angeli saepe figuras, characteres, formas et voces invenerunt proposuerunt que nobis mortalibus et ignotas et stupendas nullius rei iuxta consuetum linguae usum significativas, sed per rationis nostrae summam admirationem in assiduam intelligibilium pervestigationem, deinde in illorum ipsorum venerationem et amorem inductivas.*

—Johannes Reuchlin, *De arte cabalistica*, Hagenhau, 1517, III

### 5.1 Braces, Parentheses, and Spaces

Keywords and whatever follows should be separated by a space except for semicolons. Function calls have *no space* between the arguments and the function name. Curly braces surrounding a function or control structure's code block will have the opening curly brace on the same line as the function or control structure's definition. Objects enclosed within parentheses should not include padding spaces.

```c
while (i < 5) {                //control structure
    printf("i is %d\n", i); //function call
    i++;
}
```

Control structure keywords, such as `while`, `if`, `switch`, `for`, *etc.*, which appear at the start of code blocks, should always have opening and closing curly braces. Even if the contained statement is only one line. This provides a more consistent look, avoids possible bugs introduced while editing code, and makes it easier to add code between the braces later on.

```c
if (i < 5) {
    puts("This style is good!");
}

if (i > 5)
    puts("This style is bad!");
```

### 5.2 Variable and Function Naming

Variable and function names must be *snake cased.* This means that all characters are lower case and all words are separated by underscores. Variable and function names should be concise and clearly describe their function and/or purpose. Exceptions are for iteration control variables, in which they can be single characters (such as `i`, `j`, and `k`), and temporary variables (such as `temp`).

```c
// Example of snake case variable naming with a loop.
uint32_t odd_count = 0;
for (uint32_t i = 0; i < 4096; i += 1) {
    if (i % 2 != 0) {
        odd_count += 1;
    }
}
```

In the case of pointers, the `*` is directly followed by the variable name. `malloc()` returns type (void *), so we cast it to type `(char *)`.

```c
char *first_name = (char *)malloc(10 * sizeof(char));
char *last_name = (char *)malloc(10 * sizeof(char));
```

### 5.3 Structure Naming

Structure names will be *Pascal cased.* Pascal case is a subset of *camel case* in which the first letter is also capitalized. Each `struct` should be `typedef`'d to create opaque, exportable types.

```c
// Typedef struct TreeNode to be just TreeNode.
typedef struct TreeNode TreeNode;

// Definition of struct TreeNode, a binary tree node.
//
// left:   The node's left child.
// right:  The node's right child.
// data:   The node's data.
typedef struct {
    TreeNode *left;
    TreeNode *right;
    uint32_t data;
} TreeNode;
```

## 6 Variables and Types

*When my code has tons of trouble*
*Wesley Mackey comes to me*
*Speaking words of wisdom:*
*"Code in C"*

### 6.1 Booleans

Booleans should be of type `bool` and not `int`. All boolean values in **C** are technically integers where 0 is `false` and anything else is considered `true`. Although `bool` and `int` are functionally interchangeable when evaluating conditionals, using these different types communicates a specific variable's purpose and therefore clarifies otherwise vague code. The `bool` type is provided by the header file `stdbool.h`.

For example, assume a function named `pop()` that pops an item off a stack. Depending on the implementation, `pop()` could either return the removed value or an exit status. For this example, assume the purpose of the function is to remove the item and return true if the item was removed successfully and false if it wasn't.

```c
// Not using a boolean makes the purpose of the function
// unclear.
int pop(Stack *s, int *top);

// Using a boolean makes the purpose of the function clearer.
bool pop(Stack *s, int *top);
```

## 6.2 Integers

Include `stdint.h` or `inttypes.h` to use fixed-width integer types for integer variables. These are defined as `intN_t` and `uintN_t`, where `N` is either 8, 16, 32, or 64, and refers to the width of the integer in bits (*e.g.* `uint64_t` refers to a 64-bit unsigned integer). You should only use signed integers only if you need negative values.

To maximize portability of passing integers to functions that operate on `printf`-like format strings, include `inttypes.h` instead of `stdint.h` to use defined format specifier macros. These macros are defined like `PRIxN`, where, `x` is a `printf` format specifier like `d` or `c`, and `N` is the width in bits of the variable.

```c
// Example of printing an uint64_t with unsigned integer
// formatting.
uint64_t n = 42;
printf("n equals %" PRIu64 "\n", n);

// Printing an int32_t with decimal formatting.
int32_t m = 360;
printf("m equals %" PRId32 "\n", n);

// Printing an integer in hexadecimal.
printf("%" PRIx64 "\n", n);

// Printing an integer in octal.
printf("%" PRIo64 "\n", n);
```

## 6.3 Floats and Doubles

Both floats and doubles, denoted by `float` and `double` respectively, are used to represent floating point numbers. You can think of floating point numbers as real numbers with limited precision. A `double` has twice the precision of a `float`. However, since we are basically squeezing infinitely many real numbers into a finite number of bits, floating point numbers can only give an approximate representation, and thus are rounded. Due to this rounding, there can be errors in certain calculations and those errors can add up and snowball into very large errors. So how do you pick between using a `double` and a `float`? *Use as much precision as is needed and nothing more.* A `float` can represent up to 7 decimal digits, and a `double` up to 15 decimal digits. If you do not need more than 7 digits, use a `float`. When declaring a floating point number, add a '`.0`' after the significant to indicate that the value is not an integer.

```c
// Example of declaring pi as a double and float.
#define PI 3.141592653589793238462643383279502884197

double life = 42.0 // Include the '.0' to show it's not an integer.
float pi_float = PI;
double pi_double = PI;

printf("Pi as float: %0.20f\n", pi_float);
// This gives up to 7 decimal points of precision.
// Output: "Pi as float: 3.14159274101257324218"

printf("Pi as double: %0.20lf\n", pi_double);
// This gives up to 15 decimal points of precision.
// Output: "Pi as double: 3.14159265358979311599"
```

## 6.4 Constants

The keyword `const` serves as a type *qualifier*. In the type `int const`, `int` is the type *specifier*, and `const` is the type *qualifier*. You should always write the qualifier after the type in which it qualifies. The purpose of `const` is to restrict the value of a variable to solely what it is initialized to. You can't assign a value to a `const` value after it has already been declared, even if the value to assign it is the same value it was declared with. You also can't assign a value to a `const` value even if it wasn't initialized to anything when it was first declared.

```c
int const foo = 5;  // This is fine.
foo = 6;            // This is not fine.
foo = 5;            // This is still not fine.
```

`const` pointers are a bit more difficult. Where the `*` appears in the declaration determines what is considered constant, the pointer or the data it is pointing two. In the case of `int const *foo` we have a pointer to a `int const` type. Here the pointer can be modified but the data it points to cannot. In the case of `int *const foo` we have a constant pointer to an `int`. The data can be modified but the pointer cannot. For this reason one must be careful when declaring `const` pointers.

Constants are described in more detail in §2.4 of Kernighan and Ritchie and Chapter 11 of *C in a Nutshell*.

```c
int const *foo; // pointer to an int const, the pointer
                // can be modified.
int *const foo; // const pointer to an int, can
                // modify the data.
```

## 6.5 `static`

There are two uses for the `static` keyword in **C**. A static variable is declared *inside* a function if the value of the variable needs to persist across function calls and should only exist within the scope of the function it's declared in. A static variable is declared *outside*, if the value of the variable needs to persist across function calls and should only exist within the scope of the file where it's declared. Functions and global variables that are not accessed outside of the file where they are defined should be declared `static`.

The example shown below demonstrates how `static` maintains the value of statically declared variables across function calls.

```c
static.c
static uint32_t x = 0; // Can only be accessed within static.c

void increment() {
    // This value will persist across function calls.
    static uint32_t y = 0;

    x += 1;
    y += 2;

    printf("%d ", x);
    printf("%d\n", y);
}

void main() {
    increment(); // 1 2 should be printed.
    increment(); // 2 4 should be printed.
}
```

More information about `static` variables and functions can be found in §4.6 of Kernighan and Ritchie and Chapters 7 and 11 of *C in a Nutshell*.

## 6.6 `extern`

The `extern` keyword extends the visibility of variables and functions so that they can be called from any one of the program's files, *provided that the declaration is known.* By default, functions in **C** implicitly prepend an `extern` keyword and hence extern is not used in function declarations and definitions. Explicit use of the `extern` keyword generally revolves around variables that to for variable declarations. These variables are essentially, global variables. Variables can be declared any number of times but they can be defined only once. By defining a variable we mean that memory is allocated to the variable. In the following example, `util.c` defines a variable declared as `extern` in `util.h`. A file that includes `util.h` will be able to see and modify the `extern` variable.

```c
util.h
#pragma once

#include <stdbool.h>

extern int counter; // External counter declaration.
```

util.c
```c
#include "util.h"
#include <stdio.h>

int counter = 42; // Counter definition.

int decrement(void) {
    return counter--;
}
```

`extern` declarations and scoping rules are described more in §1.10 and §4.4 of Kernighan and Ritchie and Chapters 7 and 11 of *C in a Nutshell*.

## 6.7 Type Casting

When an operation has operands of different types, **C** will try to convert the operands into a common type. In general, these automatic type conversions are those that convert a "narrower" type into a "wider" type. An example of this would be converting a `uint8_t` into a `uint32_t`, the former of which is smaller than the latter. Another example is in the case of integer division. In **C** integer division truncates, or returns an integer and discards the remainder. Casting to a `float` or a `double` as shown in the example below will prevent truncation and return an accurate value. One must also take care to not cast from a double to a float as this will result in lost accuracy.

In **C**, explicit type conversions can be forced using the unary *cast* operator. Type casting is done through the construction of (*type-name*) *expression*. The following are examples of explicit type casting:

```c
// Converts a char into a uint8_t.
char a = 'a';
uint8_t c = (uint8_t) a;

// Avoiding truncation with integer division
double f;
f = 1/2;
// This results in f=0.0. Since the operation was performed
// as integer division the quotient was truncated and only
// stored as a double.

f = (double) 1/2;
// This gives us f=0.5 as one of the two operands were cast
// to a double.
```

Type casting should be used whenever you allocate memory using `malloc`-like functions, like in the second example above. This is because `malloc`-like functions return `(void *)`'s. Other than for casting `(void *)`'s, type-casting should be done rarely and you should be careful with operator precedence if you must type-cast.

```c
// Allocating memory for an int array.
int *arr = (int *) malloc(10 * sizeof(int));
```

Type conversions and casting is described in §2.7 and §A6 of Kernighan and Ritchie and Chapter 4 of *C in a Nutshell*.

### 6.8 Enumeration

Enumeration of constants, or assigning names to constants, is done using `enum`. Enumeration is typically used for a set of related symbols to give a unique value. A good example would be enumerating status codes. You should never enumerate magic numbers; always define magic numbers using #define. Enumerations should be `typedef`'d. If the enumeration tokens aren't manually given a value, the first token will by default be assigned the value of 0, the next token assigned the value of 1, and so on and so forth. You can also specify what value to start enumerating from, and if you wish, manually assign values for each `enum` token.

```c
// Example of enumerating a week.
// Tue == 3, Wed == 4, etc.
typedef enum { Mon = 2, Tue, Wed, Thu, Fri, Sat, Sun } Week;

// How to use the enumeration.
Week day = Mon;
printf("%d\n", day); // This prints out 2.
```

Enumerations are described in §2.3 of Kernighan and Ritchie and Chapter 2 of *C in a Nutshell*.

## 7 Control Structures

*Code in C, Code in C, Code in C, ...*
*There will be an answer, Code in C.*

### 7.1 If Statements

If-else statements directly follow one another. This means that that an `else` or `else if` statement following an `if` statement should be on the same line as the `if` statement's closing brace.

```c
if (first condition) {
    // Statements if first condition is true.
} else if (second condition) {
    // Statements if second condition is true.
} else {
    // Statements if neither are true.
}
```

If-else and Else-if are described further in §3.2 and §3.3 of Kernighan and Ritchie and Chapter 6 of *C in a Nutshell*.

### 7.2 Loops

Infinite loops are to be written using `while` loops. Infinite loops should be exited using `break`. In reality, `goto` can also be used to since `goto`, like `break`, alters the normal execution of code, but using it without very good reason to do so is considered bad practice as explained in Dijkstra's letter, "Go To Statement

Considered Harmful"[1]. Kernel programming sometimes uses goto for error handling, but you are not programming in the Kernel.

```c
// Make sure to use the stdbool.h constant "true".
while (true) {
    printf("Deja vu");
}

// Example of a for loop.
for (int i = 0; i < 5; i++) {
    foo++;
}

// Example of a do-while loop.
do {
    foo++;
} while (i < 5);
```

More information on loops can be found in §3.5 of Kernighan and Ritchie and Chapter 6 of *C in a Nutshell*.

## 7.3 Switch Statements

Switch statements are an exception to the previously describe brace conventions. Although the switch statement, denoted by `switch`, does require braces like a loop or `if` statement. The encapsulated `case` statements *do not* require them. Adding extra braces for clarity does not hurt and feel free to do so as long as you are consistent. The `break` statement must follow *all* statements for a specific case, unless the goal is to fall through more than one case. Fall-through is the act of not including the `break` statement so that more than one case can share code. You should generally avoid having a fall-through in your switch statements. All `switch` statements should also have a `default` case, which is invoked if no case conditions are met, even if it is empty. *We will mark you down if you don't!*

```c
switch (i % 2) {
case 0:
    printf("Variable i (\%d) is even!\n", i);
    break;
case 1:
    printf("Variable i (\%d) is odd!\n", i);
    break;
default:
    puts("How is it neither even nor odd?!");
    break;
}
```

Switch statements and their specifics are more thoroughly described in §3.4 of Kernighan and Ritchie and Chapter 6 of *C in a Nutshell*.

[1] Edsger W. Dijkstra. 1968. Letters to the editor: go to statement considered harmful. Communications of the ACM 11, 3 (March 1968), 147–148.

## 8 Includes and Defines

*Eleanor Rigby*
*Sits at the keyboard and waits for a line on the screen*
*Lives in a dream*
*Waits for a signal*
*Finding some code that will make the machine do some more.*
*What is it for?*
*All the lonely users, where do they all come from?*
*All the lonely users, why does it take so long?*

The first step of compiling a **C** program is running the preprocessor. The major tasks for the preprocessor are to include header files specified by `#include` and to perform token replacement with `#define`. The features of the preprocessor are more thoroughly described in §4.11 of Kernighan and Ritchie and Chapter 15 of *C in a Nutshell*.

The primary way to connect multiple files together into one program is through the use of *header files* which are specified with the `.h` file extension. The contents of these files are included in your program through the `#include` command. Header files should include prototypes and declarations for functions, structs, *etc.* that will be visible outside of their encapsulating `.c` file. The use of header files is more thoroughly described in §10.2 of this document.

The `#define` macro tells the preprocessor to replace a string of text with another before the code is actually compiled. This can be used to give names to "magic" numbers, or numbers who's purpose is not obvious to the reader. It is also useful to give names to constants that appear multiple times in a program such as block sizes or maximum values for a variable. Lastly, some basic functions or repeated commands can be represented through macros but this should be used with caution. The `#define` macro also enables the programmer to modify all instances of a specific string or number while only editing a single line of code.

Includes and defines are to be formatted the same way: no space after the '#' character. Thus, includes and defines are formatted as shown in the following example. The names of `#define` macros must be in snake case and all caps.

```c
// Example of including standard I/O and boolean libraries.
#include <stdio.h>
#include <stdbool.h>
// Example of including one of your own header files.
#include "foobar.h"

// Example of defining a block size macro.
// Every instance of BLOCK_SIZE is replaced with 4096.
#define BLOCK_SIZE 4096

// Example of preventing duplicate macro definitions using #ifndef.
// Macros that go between the #ifndef and #endif are guarded.
#ifndef MAGIC
#define MAGIC 0xdeadd00d
// ... other defines ...
#endif
```

In general, you must be careful with macros. They use *text substitution* and do not work through function calls. Some avoid using them as much as possible. It is our opinion that they should be used with care. A better alternative to macros that are more complex than a simple value is an *inline function* as shown below.

```c
// Dangerous under the right circumstances.
#define MAX(X, Y) (((X) > (Y)) ? (X) : (Y))

// A call like this.
MAX(a++, b++)
// Will be replaced with this, which increments the returned
// value twice.
(((a++) > (b++) ? (a++) : (b++))

// A better macro using compiler builtins,
// a bit complex and difficult to read.
// Portability is also of concern.
#define BETTER_MAX(X,Y) \
    ({ __typeof__(X) _X = (X); \
     __typeof__(Y) _Y = (Y);  \
     _X > _Y ? _X : _Y; })

// The best option is an inline function.
inline int max(int a, int b) {
    if (a > b) {
        return a;
    }

    return b;
}

// Another inline funciton.
inline int other_max(int a, int b) {
    return a > b ? a : b;
}
```

The first macro can produce what is called a side effect, an unintended changed to a variable. In this case it can lead to the returned value being modified twice. The second macro is a bit better but uses some arcane compiler incantations to create temporary copies of the two variables and therefore can be difficult to understand. The best solution that you should use is an *inline function*, which performs the same task with better protections and similar behavior when compared to a macro. The last example uses the ternary operator or ?, which helps write clean and simple code. Two ternary operator examples are shown below.

```c
// condition ? true : false;
x = a > b ? a-- : b++;
return_code = a == b ? foo() : bar();
```

When in doubt, seek to be easier to understand than clever.

## 9 Comments

*Life should not be a journey to the grave with the intention of arriving safely in a pretty and well preserved body, but rather to skid in broadside in a cloud of smoke, thoroughly used up, totally worn out, and loudly proclaiming "Wow! What a Ride!"*

—Hunter S. Thompson, *The Proud Highway: Saga of a Desperate Southern Gentleman, 1955–1967*

You will be using "//" style comments, with a single space separating the "//" comment prefix and the comment itself. These are line comments and only comment out characters that follow it on the same line. We will *not* be using the other bracket style comment "/* ...*/" as it is error-prone. Again, avoid unnecessary and excessive comments. Commenting is a balancing act. You do not want to comment too much or too little. The goal of is to explain what is being done or what is not immediately evident.

*All functions and structs must also provide a comment block right above their declarations, briefly explaining their usage, return value, and details of the parameters they accept.* Any *preconditions* or *post-conditions* should be established as well. Preconditions are conditions that must be upheld prior to calling the function. Postconditions are conditions that are upheld after calling the function. Here is example of a properly commented function:

```c
// Prints a string to an opened file.
// Returns true if string is printed successfully.
// Returns false otherwise, or if the string/file is NULL.
//
// outfile: The file to print to. Must already be opened.
// str: The string to print.
bool strprint(FILE *outfile, char *str) {
  if (!outfile || !str) {
    return false;
  }
  fprintf(outfile, "%s", str);
  return true;
}
```

## 10 Abstract Data Types (ADTs)

*The public knows that human beings are fallible. Only people blinded by ideology fall into the trap of believing in their own infallibility.*

—Freeman Dyson

You will be learning many data structures and making abstract data types using them this quarter. An ADT is a type, or class, for objects whose behavior is defined by a set of functions. These functions are split into three categories: *constructor/destructor*, *accessor*, and *manipulator*. An ADT constructor handles allocating memory for a new instance of the ADT, its destructor frees any allocated memory associated with the ADT. Accessor functions are mainly used in ADT implementations where the design of the ADT is not made explicit and used to get ADT fields. In this class, struct definitions of ADTs will go in header files, so this is not an issue. Manipulator functions serve to alter values stored in an ADT. ADTs should always have a source file (**C**-file) and a corresponding header file.

### 10.1 ADT Functions

All functions must have a comment block preceding it that gives a brief overview on the purpose and usage of not just the function, but also the input parameters, followed by the output parameters. Return types of functions must be explicitly stated, even if it is a `void`-type function. ADT functions must be written *noun* then *verb* style, where *noun* refers to the ADT and *verb* the action to perform on the ADT object.

### 10.2 ADT Header Files

Header files serve as ADT interfaces and contain function prototypes and definitions of functions implemented in their respective corresponding source files.

Header files must have a header guard at the top in order to prevent duplicate header file inclusion. Header guards are defined using the `#pragma once` preprocessor macro. As the name might suggest, preprocessor macros are processed during the compiler's preprocessor phase. If the file has already been included at this phase, then the contents of the file are skipped over and not preprocessed. If the file hasn't yet been included, then its contents are preprocessed.

A header file will include the declarations for structs, global variables, function prototypes, *etc.* that will be visible to other `.c` files. Variable definitions, function source code, and the like do not belong in a header file with the exception of inline functions.

### 10.3 ADT Source Files

These source files contain the implementations of all functions prototyped in their corresponding header files as well as definitions for their variables. Declarations of structs and variables and function prototypes that are not to be visible to other `.c` files also appear in a source file. Excluding the header guard, all code in source files are commented and formatted the same way as in header files.

## 11 Example BitVector ADT

*Vector: You wanted to see me dad.*
*Mr. Perkins: Oh yes, come in Victor.*
*Vector: Actually, Victor was my nerd name. Now its Vector!*
*Mr. Perkins: Sit down!*

*Despicable Me*

In the given example, the *bv_create* function is an example of a constructor function and *bv_delete* function is an example of a destructor function.

bv.h

```c
#pragma once

#include <stdint.h>

// A BitVector is a dynamically allocated vector of bits.
typedef struct BitVector BitVector;

// Creates and returns a BitVector of specified length.
// Returns NULL if memory allocation fails during creation.
//
// length: The number of bits the BitVector represents.
BitVector *bv_create(uint32_t length);
```

bv.c

```c
#include "bv.h"

struct BitVector {
    uint8_t *vector;
    uint32_t length;
};

// Creates and returns a BitVector of specified length.
// Returns NULL if memory allocation fails during creation.
//
// length: The number of bits the BitVector represents.
BitVector *bv_create(uint32_t length) {
    BitVector *bv = (BitVector *)malloc(sizeof(BitVector));
    if (bv != NULL) {
        bv->length = length;
        bv->vector = calloc(length / 8 + 1, sizeof(BitVector));
        if (!bv->vector) {
            free(bv);
            bv = NULL;
        }
    }
    return bv;
}
```