Introduction
This document shall serve as a brief introduction to C coding style to be used in CIS 212. As with other languages, proper style is not enforced by the compiler; proper style is necessary in order to write clear and human-readable code. All support code provided with your project assignments will follow these conventions. Your submitted code is expected to conform to these conventions, as well.

Version of C and standard arguments to gcc
While there have been several subsequent versions of C standardized in the recent past, all supplied code will conform to the c89 standard. Your submitted code must compile without errors or warnings using the gcc compiler supplied with the Arch Linux virtual machine provided to you. Finally, we will always use the “-W -Wall” warning flags when compiling your code to test it.

You are strongly advised to modify your code to eliminate any warnings generated by the compiler with the “-W -Wall” compiler flags.

Functions Without Arguments
Function arguments in C are passed as a comma-separated list enclosed by parentheses. When declaring a function that does not accept any arguments, it should be declared as

```
type function(void);
```

To invoke the function in your program, you simply use an expression of the form

```
v = function();
```

where “v” is a variable of type “type”.

This is required, as a declaration of the form “type function();” means that the function can take any number of arguments. This is due to backwards compatibility with earlier versions of the C standard.

Declaring Pointers
The following declares a variable and a pointer to a variable of a particular type. The second line shows how we assign a value to the pointer.

```
type v, *p;
p = &v;
```

The declaration above indicates that “*p” can be used anywhere a value of “type” can be used.

Layout and Spacing
Functions should be of a reasonable length; you should not have to scroll down through your editor of choice to view an entire function body. Sometimes this may be unavoidable; in such situations, ensure
that your functions are easily broken up into logically discrete units. **You should not sacrifice readability for length.**

Specifically, your `main()` function should have no more than 200 lines of code. Having a long `main()` function indicates a lack of abstraction, implying that your code has not been sufficiently refined into smaller, logical functions.

Line lengths should be no more than 80 characters, both for code and comments. Indentation should be consistent – we strongly recommend a 4-space indent. **Do not indent by a single space; this makes indentation extremely hard to follow.**

Be sure to use a reasonable amount of horizontal spacing. For example, you should put space between operators:

\[(x+11)/(y\%5)-z\] is far less readable than \[(x + 11) / (y \% 5) - z\]

**Brace Positioning for Blocks**

There are two standard styles for the positioning of the opening curly brace around a block of code (body of a function, conditional, loop):

```c
while (...) {
    /* the body */
}
or
while (...) {
    /* the body */
}
```

Both of these are legal; generally, the leftmost style above is the preferred style, since it saves one line of vertical space; it will be used in this course.

**Conditional and Looping Constructs**

The scope of a conditional construct, or for a looping construct, is a single statement; thus, it is perfectly legal to write code like this:

```c
if (...) {
    a statement that does something
}
```

In C, we can use a block (enclosed by curly braces) anywhere a single statement can appear. Thus, the above expression can also be written as:

```c
if (...) {
    a statement that does something
}
```

Why would this latter form be preferred? Suppose we wanted to add a second statement to the code that should be executed if the condition is TRUE? If we modified the former expression as

```c
if (...) {
    a statement that does something
    second statement
}
```
it would not achieve our desired outcome. We would still execute “a statement that does something” if the condition is TRUE, but we would always execute “second statement” regardless of the truth of the condition. The equivalent modification of the latter expression yields

```
if (...) {
    a statement that does something
    second statement
}
```

does the right thing, since everything in the block delimited by the curly braces is the “statement” associated with the if clause.

It is so common to augment the work associated with a conditional or looping construct that you are encouraged to always use curly braces to delimit the actions associated with these constructs.

How should we format a conditional if there is also an else clause?

```
if (...) {
    what we should do if true
} else {
    what we should do if false
}
```

This simply conforms to our previously stated rule.

**Naming Conventions**

The textbook and all of the examples in the slides use the following naming convention for function and variable names: for single word variable or function names, use all lower case; for multi-word names, use “camel case” – i.e., capitalize the first letter of the 2nd and subsequent words (e.g., modificationTime).\(^1\)

For symbolic constants (which are defined using `#define`), the constants are all upper case, with underscores separating words (e.g., BUFFER_SIZE).

Finally, for data types, the textbook and the slides use mixed case with all words having their first letter capitalized (e.g., PriorityQueue).\(^2\)

**Commenting**

**Header Comments**

In Python, you have learned to create a docstring at the top of each function, describing what the function does, a description of each input argument, and an explanation for the output of the function.

\(^1\) Note that the C standard libraries conform to a different convention – variable and function names are entirely in lower case; for multi-word names, an underscore is used to separate the words (e.g., modification_time).

\(^2\) Note that the C standard libraries conform to a different convention – data type names are entirely in lower case; for multi-word names, an underscore is used to separate the words; and “_t” is appended to the name (e.g., priority_queue_t).
C Coding Style and Conventions

It is good style to provide header comments for your C functions, as well. Consider the following example:

```c
/*
 * Count the number of instances a particular integer
 * appears in an array
 *
 * item - the integer to be found in the array
 * arr - the array to be searched
 * size - the number of elements in the array
 * returns the number of times `item' appears in `arr'
 */
long countItemInArray(long item, long *arr, long size) {
    * * *
}
```

Inline Comments

It is also good practice to write inline comments to help explain particularly complicated sections of code. Inline comments are all text contained within “/*” and “*/” tokens. As with the header comment example above, comments can be continued over multiple lines, if necessary.

You should not find yourself writing inline comments for every other line of code – such behavior usually indicates that you need to re-evaluate your logic to find a clearer way to write the code.

Extraneous Comments

There are some comments that you should not leave in your code. If you have used calls to `printf()` or other functions to debug your code, these should be removed from your submitted code (not just commented out). If any supplied code has TODOs in comments to help guide your work, these comments should also be removed.

Your Report File

If the project assignment indicates that you must provide a Report file, it should provide a high-level overview of your program structure, should enumerate those students who helped you debug, and discuss any known bugs in your program.

Logic Misuse to Avoid

1. Do **not** use “while (1) { ... if (condition) break; ...}”; instead use

   ```c
   while (!condition) { ... }
   ```

2. Same with a for loop
3. Do **not** have an empty action clause in an if expression, such as “if (...) {} else {...}”
4. Do **not** create a variable just to return it as the function value; instead of

   ```c
   int ret = 3 * mul + 7; return ret;  use  return (3 * mul + 7);
   ```
5. Do **not** use an if/else statement to return the value of a condition; instead of

   ```c
   if (cond) return 1; else return 0;  use  return cond;
   ```
6. Do **not** use global variables at all.
7. Do not duplicate code; if you find yourself repeating the same code sequence two or more times, consolidate into a helper function.
8. Do not use goto at all.

**Pointer Misuse**
1. C enables you to pass copies of structs as arguments; except in very rare situations, this is the wrong thing to do. Pass pointers to structs instead.
2. Do not use (*p).val instead of p->val. Manually dereferencing the pointer and then accessing its field is more work than simply using the -> syntax.
3. Do not use *(arr + i) instead of arr[i]. The former syntax is much less clear.