CS 3: Introduction to Software Design

Practicum 1: `strlib`

strlib.h

```
1  char *strlib_strcat(char *dest, const char *src);
2  int strlib_strcmp(const char *str1, const char *str2);
3  char *strlib_strcpy(char *dest, const char *src);
4  char *strlib_strncpy(char *dest, const char *src, int n);
5  size_t strlib_strlen(const char *str);
```

strlib.c

```
1  #include "strlib.h"
2
3  size_t strlib_strlen(const char *str) {
4     int i = 0;
5     while (str[i++] != '\0');
6     return i - 1;
7  }
8
9  char *strlib_strcat(char *dest, const char *src) {
10    int strlen_of_dest = strlib_strlen(dest);
11    int i = 0;
12    for (i = 0; i < strlib_strlen(src); i++) {
13       dest[strlen_of_dest + i] = src[i];
14    }
15    dest[strlen_of_dest + i] = '\0';
16 }
17
18 int strlib_strcmp(const char *str1, const char *str2) {
19    if (strlib_strlen(str1) < strlib_strlen(str2)) {
20       return -1;
21    }
22    if (strlib_strlen(str1) > strlib_strlen(str2)) {
23       return 1;
24    }
25    if (strlib_strlen(str1) == strlib_strlen(str2)) {
26       int i = 0;
27       while (str1[i] != '\0' && str1[i] == str2[i]) {
28          i++;
29       }
30       if (strlib_strlen(str1) == i) {
31          return 0;
32       }
33       else {
34          return str2[i] - str1[i];
35       }
36    }
37 }
38
```

```
char *strlib_strcpy(char *dest, const char *src) {
   int i = 0;
   while (src[i] != '\0') {
      dest[i] = src[i];
      i++;
   }
   dest[i] = '\0';
   return dest;
}

char *strlib_strncpy(char *dest, const char *src, int n) {
   int i = 0;
   while (src[i] != '\0' && i < n) {
      dest[i] = src[i];
      i++;
   }
   if (i < n) {
      dest[i] = '\0';
   }
   return dest;
}
```

CS 3: Introduction to Software Design

Practicum 1 Code Quality Items

- **commenting::description**
  - A comment should describe the actual behavior that the piece of code has
  - A comment should not be unclear, useless, or uninformative
  - All public functions should be docmented ONLY in the header file, NOT in the .c file!
  - Make sure to specify the type and purpose of each parameter and return value, as well as what the function does at a high level.
  - Code should not be "overcommented"

- **design::encapsulation** – Encapsulation is, at a high level, hiding data from clients of structures in the .c files. It is the process of hiding the implementation of a module from the code that uses it. This is important because if we need to change our underlying implementation, code that we have written that uses it should still be able to function. Encapsulation also makes things more readable and understandable for the user.

- **design::expensive-function-calls** – Function calls (especially for expensive ones) should be minimized while still preserving the functionality of the code.

- **design::extra-control-flow** – Additional control structures should not be used if they can be simplified.

- **design::includes** – Code should not include a ".c" file rather than a header

- **formatting::one-liner** – Code should be broken down into multiple lines, with one statement per line. Control loops and statements should be expanded into multiple lines instead of having their braces in one line.

- **functions::code-duplication-multiple-functions** – Decompose significant amount of code duplication, by factoring out common code from into helper functions. (this is usually any more than 10 lines) and or lack of decomposition of functions into helper functions

- **functions::code-duplication-single-function** – Lines of code should not be duplicated within a single function.

- **variables::declaration-location** – Declare variables in the narrowest possible scope, as close to its usage as possible.

- **variables::description** – A variable is properly descriptive and not overly descriptive

- **variables::magic-numbers** – Use a const variable to appropriately abstract magic numbers from the code

- **variables::type** – Use a variable type that specifies the size explicitly (e.g., `int32_t`, `size_t`) rather than an `int` or `long`