CS 2110 Homework 7
Intro to C

Richard Zhang, Sainitin Daverpally, Max Ko,
Thanasis Taprantzis, Alex Whitlock, Samarth Kamat

Version 1.1

Contents

1 Overview

1.1 Purpose

The purpose of this assignment is to introduce you to basic C programming along with the tools you need to succeed as a C programmer. This assignment will familiarize you with C syntax and how to compile, run, and debug C. Furthermore, you will gain exposure to common practices such as man(ual) pages and standard libraries which are utilized in real world C programming.

You will become familiar with how to work with strings, arrays, pointers, and structs in C. You will understand the relationship in C between arrays and pointers, including pointer arithmetic (think about how arrays are stored in memory in assembly). You will also become familiar with how to use a Makefile to automate the compilation of your program.

1.2 Task

In this assignment, you will put yourself in the shoes of your fellow TAs and with your knowledge of the C programming language, you will design a queue system for CS2110 Office Hour.

First, you will familiarize yourself with the C programming language by implementing several string functions. Next, you will write C functions to add and remove students, update student's topic, as well as group and remove students by topic. See the Detailed Instructions section for more details on the specific requirements for each function. You will write your C code in two files; `my_string.c`, `oh_queue.c`.

IMPORTANT NOTE:

- **You need to complete 'string.c' before 'oh_queue.c' as you will be using the implemented functions from 'string.c' in 'oh_queue.c'.**
- **The assignment is due by 11:59 PM on March 28th, 2023. You could also submit the assignment by March 29th, 2023 for a late penalty of 25% of your overall grade.**

In `my_string.c`, you will write your own implementations of common C library functions for working with strings:

- `my_strlen()`
- `my_strncmp()`
- `my_strncpy()`
- `my_strncat()`
- `my_memset()`
- `remove_first_instance()`
- `replace_character_with_string()`
- `remove_first_character()`

In `oh_queue.c`, you will implement the new Office Hour queue system. More specifically, the following functions:

- `push()`
- `pop()`
- `group_by_topic()`
- `hash()`
- `update_student()`
- `remove_student_by_name()`
- `remove_student_by_topic()`
- `OfficeHoursStatus()`

Take a look at the sections on Makefiles and Autograder for more info on how to compile and test your program.

1.3 TA Tips

While doing the homework, you may find it helpful to draw diagrams of memory locations, as you did with assembly programming. How are arrays represented in memory? How can you use pointers to find the address of `array[i]`? How are arguments passed to a function and results returned using the stack frame? Remember, C functions are pass by value (copies of the values of arguments are pushed onto the stack), just like subroutines in assembly.

1.4 Criteria

Your C code must compile without errors or warnings, using the provided Makefile. Your array of structs should be populated correctly at the end of the program. Your helper functions in `my_string.c` must all be implemented correctly, producing the same behavior for test cases as the equivalent library functions from `string.h`.

2 Detailed Instructions

2.1 `my_string.c` functions

The first part of this homework is to implement three very common C string library functions found in `string.h`. The caveat is that you must implement these functions **using only pointer notation**. That is, you cannot use array indexing notation, such as `str[i] = ‘a’`. This restriction only applies in the `my_string.c` file. We recommend implementing these functions first so you are able to use these functions as you move on with the assignment. Make sure to read all the information in this section to ensure you have all the information you need to succeed!

- `my_strlen`: calculates the length of the string pointed to by s, upto and excluding the terminating null terminator (`‘\0’`). Consider the following examples:
  - if `char *pet = "otter"`, then `my_strlen(pet) == 5`. Remember the null terminator is implicitly added here.
  - if `char letters[] = [‘a’, ‘b’,‘c’, ‘\0’, ‘d’]`, then `my_strlen(letters) == 3`
  - If a string with no null terminator is passed into `my_strlen` the string standard library defines the output to be undefined behavior. *Consider why this might be the case?* Therefore, this function should do nothing extra to handle inputs of invalid strings (ie: unterminated strings) because the programmer is expected to pass in valid strings.
- `my_strncmp`: compare two strings, up to at most n characters. Comparisons should be made between each character between the ASCII values of each pair of corresponding characters. Comparisons should be character by character until there is a difference between the corresponding pair of characters, or both strings terminate at the same character, or until we have completed `n` comparisons. Return any arbitrary negative integer if the first string is less than the second string, zero if equal, and positive if greater. You should also remember that we need to remember to account for null terminators. Some examples of string comparisons are below:
  - `my_strncmp("bazz", "bog", 3) < 0`
  - `my_strncmp("rainforest","rain", 4) == 0`
  - `my_strncmp("rainforest","rain", 5) > 0` recall, `‘f’ > ‘\0’`
  - `my_strncmp("\0aa", "\0ab", 3) == 0`, notice how and where an additional null terminator is placed mid-string
- `my_strncpy`: copy at most n characters from a source string to a destination memory location. If the null terminator is not included in the first n characters of source, **the destination string should not be automatically null terminated by this function**. If the length of source is less than n, you should fill in the remaining n characters with null terminators. Remember, `my_strncpy` does not operate based on the size of the destination string. Therefore, it has to be assumed `my_strncpy` was provided a destination buffer sufficiently large enough to receive the copy, otherwise there could be a buffer overflow error. Furthermore, it is not required that n be the exact same size as the destination buffer.
- `my_strncat`: appends at most n bytes from *src* string to the *dest* string, overwriting the null terminator at the end of *dest*, and then adds a null terminator at the end of the concatenated string. If the length of the *src* string is less than n, copy up to and excluding the null-terminator. Assume that the *dest* string will have enough space for the result.
- `my_memset`: fills the first n bytes of the memory area pointed to by *str* with the constant int c. The function returns a pointer to the memory location *str*.
- `remove_first_instance`: finds the first instance of the character c and remove it from the *str* string. You will need to modify *str* directly.

  - if `str = "bazz"`, then `remove_first_instance(str, 'a')` will change `str` to be `"bzz"`.
  - if `str = "bazaz"`, then `remove_first_instance(str, 'z')` will change `str` to be `"baaz"`.
  - If no instance of the character was found, the function does nothing to `str`.

- `replace_character_with_string`: finds the first instance of the character *c* in string *str* and replace it with the character pointed by *replaceStr*. You will need to modify *str* directly.
  - if `str = "bazz"`, then `remove_character_with_string(str, 'a', "horse")` will change `str` to be `"bhorsezz"`.
  - if `str = "bazaz"`, then `remove_character_with_string(str, 'z', "2110")` will change `str` to be `"ba2110az"`.
  - If no instance of the character was found, the function does nothing to `str`.

- `remove_first_character`: removes the first character of the string in **ONE line**. You will be given a pointer to a string (`char **`) and will need to change the pointer to point to a string with the first character removed.

You will notice that many of the arguments in the `my_string.c` and `oh_queue.c` files are of type `const char *`. This is a pointer to a char that is constant. This means that you cannot edit any of the characters to which a `const char *` points to. If you attempt to do so, using something like `*pointer = ‘c’`, you will get a compile error. If `const` does not precede a `char *` declaration, you can edit the characters to which it points to.

In order to understand the functionalities of these three library functions, you may also take a look at their man page (i.e. manual page). See the section on Man Pages for more info.

What is `size_t`:

`size_t` is an unsigned integer datatype in C that represents the size of a variable in bytes. `size_t` is used in the string library functions.

Some notes:

1. You should complete `my_string.c` before moving on to `oh_queue.c`.
2. **You are not allowed to use array notation in this file (e.g. s1[0]). All functions should be implemented using pointers and pointer arithmetic only!** Think about how arrays and pointers correlate with each other in C. Again, this restriction only applies to this file. If you do use array notation in your code, the makefile will indicate this error:
`make: *** [Makefile:30: check-array-notation] Error 1` along with the instances of array notation that it encountered.
3. You are not allowed to use any of the standard C string libraries (e.g. `#include <string.h>`).
4. Your string functions should not assume the length of any arguments passed in.
5. For `my_strncmp`, you do not need to return a specific number, as long as it follows the description in the `strncmp` man page.
6. `size_t` is an unsigned integer datatype typically used to represent the size or length of data, or any other unsigned quantity.
7. For `remove_first_character`, you are not allowed to use loops or conditionals. Think back to Homework 1, where you implemented several functions with just one line of code.

2.2 oh_queue.c

The second part of this homework is to implement several C functions within the oh_queue.c file. You will primarily be interacting with the global oh_queue struct.

What is the struct Queue:

The Office Hour queue is represented using the struct Queue struct defined in oh_queue.h:

```
struct Queue {
    struct OfficeHoursStats stats;
    struct Student students[MAX_QUEUE_LENGTH];
    };
```

The struct Queue contains two member variables:

- stats is a struct OfficeHoursStats that stores the the following information:
  - no_of_people_in_queue is an integer that keeps tracks of the number of students currently in the queue.
  - no_of_people_visited is an integer that keeps track of the cumulative number of students that have visited Office Hours (but not currently in the queue as they are already popped).
  - currentStatus is a char * that identifies whether office hours is "Completed" (no students are in the queue) or "InProgress" (there are still students in the queue)
- students is an array of struct Student which represent the actual office hours queue.

Like a real queue in real life, the students array should not have any gaps between students (it must be contiguous). You will use and update the no_of_people_in_queue to keep track of the current number of students in the students array.

What is a struct Student:

Each student in the array is represented using the struct Student struct defined in oh_queue.h:

```
struct Student {
    int customID[30];
    int queue_number;
    struct StudentData studentData;
};
```

The struct Student contains three member variables:

- customID is an integer array that stores hashed ID of the student name (e.g. [231,323,93,17,87]).
- queue_number is an integer representing when the student joined the queue.
- studentData is a struct containing the data of the student

What is a struct StudentData:

The information of each student is represented by a struct StudentData struct defined in oh_queue.h:

```
struct StudentData{
    char name[MAX_NAME_LENGTH];
    struct Topic topic;
};
```

The `struct Student` contains three member variables:

- `name` is a string (represented as a null-terminated array of characters) which stores the name of the user's name (e.g [‘S’,‘o’,‘p’,‘h’,’i’,‘e’,‘\0’]).
- `topic` is a `enum struct Topic` that stores the `subject topicName` and `float questionNumber`. The question topic that the students could ask are stored in the `enum subject`.

What is `struct public_key`?

In this homework, there is one RSA encryption algorithm that you will need to implement. The exact RSA encryption you will be implementing is in Appendix A. **It is important that you follow the exact encryption specification in Appendix A** (Don't worry about RSA decryption in the appendix. That was from a previous semester, but it is there in case you want to learn more about RSA). We recommend implementing these functions first so you are able to use these functions as you move on with the assignment. Make sure to read all the information in this section to ensure you have all the information you need to succeed!

For encryption, RSA depends on a public key represented by two numbers (n, e). This two-number pair is represented by the `struct public_key` struct defined as:

```
struct public_key {
    int n;
    int e;
};
```

Useful Macro Definitions in `users.h`:

- `SUCCESS` is an alias for the integer value to return when a function operation succeeds.
- `FAILURE` is an alias for the integer value to return when a function operation fails. Think of this as an error code.
- `MAX_NAME_LENGTH` represents the maximum length of the name `char` array, including the null-terminator. Remember char arrays are one way to represent strings in C.
- `MAX_QUEUE_LENGTH` represents the maximum length of the `students` array.

Hints:
In regards to the functions to be implemented below, here are some common mistakes and reminders for how to go about solving them.

- `MAX_NAME_LENGTH` represent the maximum lengths of the name `char` arrays. However, length in this case doesn't necessarily mean the length from `my_strlen`, which by design choice, does *not* include the null-terminator. Think of the maximum lengths from this macros as being the total amount of characters that have been allocated to represent the name. This means that if the maximum length is 10, for example, then the longest name string we could have comprises of 9 non-null characters (letters) and 1 null-terminator. These characters all make up the 10 maximum characters that makes up the maximum name. This idea is important for truncating strings that are longer than the maximum lengths!
- The `queue_number` is dependent on when the student was added to the queue, not the current position of the student in the queue. Think: ed post's number.
- Remember you can index into strings for reading or writing characters. For example, `some_string[3]` is the 4th character of `some_string`.

- When comparing strings to each other, be wary of the `n` argument you pass in to `my_strncmp` that tells how many characters you will be comparing. Strings that should be considered equal may return false if you force characters to be compared when they shouldn't be compared. For instance, comparing "apple" to "apple" but passing in a `n` that will force characters past their respective null-terminators to be compared, which are likely to be different and hence return false. However, we should have only been comparing the characters prior to reaching the null terminator.
- Remember all strings are to be null-terminated! Otherwise, we would not be able to tell what characters are actually part of a string.

Functions to Implement:

- `void hash(int *ciphertext, char *plaintext, struct public_key pub_key)`:

  In this function, you will create a hashing function used to generate a unique ID for the student. The hashing function utilizes the RSA algorithm, taking the plaintext (decrypted) null-terminated string and writes into a ciphertext (encrypted) integer array. It is assumed that `hash` will be provided a ciphertext integer array sufficiently large enough to hold the encrypted text of the plaintext null-terminated string.

  - For each character in `plaintext`, you will need to apply the below formula with the ASCII value of the character m and the members of the `pub_key` (n, e):

    $$(m^e) \bmod n = c$$

  - c is the cipher number that corresponds to the character m that you would put into the `ciphertext` array.

- `int push(char *studentName, enum subject topicName, float questionNumber, struct public_key pub_key)`:

  In this function, you will add a `Student` struct with the given `studentName`, `topicName`, and `questionNumber` to the end of the `oh_queue`.

  - If the given name's length (including the null terminator) is above `MAX_NAME_LENGTH`, truncate the name to be `MAX_NAME_LENGTH` (including the null terminator). Be mindful of how strings are represented in C!
  - The `customID` of the `Student` struct should be the hashed value returned by the `hash` function. You will need to use this function's `studentName` and `pub_key` to perform the hash.
  - The `queue_number` is dependent on when the student was added to the queue, not the current position of the student in the queue. For example, a student with `queue_number` 67 would be the 67th student ever to be pushed onto the queue. Think about how you can use the members in `oh_queue.stats` to calculate this number.
  - If the student's `name` is NULL, or adding the student would make `no_of_people_in_queue` exceed `MAX_QUEUE_LENGTH`, do not create or add the student to the `oh_queue`.
  - After the operation, update the stats appropriately in `oh_queue.stats`. Refer to the specification of `struct OfficeHoursStats` at the beginning of this section for details. For the `currentStats`, implementing and using the `OfficeHoursStatus` function first may be useful.
  - Return `SUCCESS` if you are able to successfully create and add the student, otherwise return `FAILURE`.

- `int pop()`:

  In this function, you will remove first student from the front of the `oh_queue`.

  - The students in the array remain contiguous in the array after the operation and should still start at index 0.

  - If there are no students in the `oh_queue`, return `FAILURE`.
  - On successful removal, return `SUCCESS`.
  - After the operation, update the stats appropriately in `oh_queue.stats`. Refer to the specification of `struct OfficeHoursStats` at the beginning of this section for details. For the `currentStats`, implementing and using the `OfficeHoursStatus` function first may be useful.

- `int group_by_topic(struct Topic topic, struct Student *grouped[])`:

  In this function, you will group together students in the queue by the given `topic` and add them to the provided `Student` array. Assume that the provided array will have enough allocated buffer space to contain all the students with the `topic`.

  - Return the number of students with the given topic.

- `int update_student(struct Topic newTopic, int customID[])`:

  In this function, you are given a `newTopic` and a `customID`. You will find the student using the given `customID` and update the student's `topic` to the `newTopic`.

  - If the student can not be found in the queue, return `FAILURE`.
  - If the student topic was successfully updated, return `SUCCESS`.

- `int remove_student_by_name(char *name)`:

  In this function, you will be removing the student with the given name from the queue.

  - The remaining students in the array remain contiguous after the operation and should still start at index 0.
  - If the student can not be found in the queue, return `FAILURE`.
  - If the student with the name was successfully removed, return `SUCCESS`.
  - After the operation, update the stats appropriately in `oh_queue.stats`. Refer to the specification of `struct OfficeHoursStats` at the beginning of this section for details. For the `currentStats`, implementing and using the `OfficeHoursStatus` function first may be useful.

- `int remove_student_by_topic(struct Topic topic)`:

  In this function, you will be removing all students from the queue with the given `topic`.

  - The remaining students in the array should remain contiguous after the operation and should still start at index 0.
  - If no students were removed, return `FAILURE`.
  - On successful removal, return `SUCCESS`.
  - After the operation, update the stats appropriately in `oh_queue.stats`. Refer to the specification of `struct OfficeHoursStats` at the beginning of this section for details. For the `currentStats`, implementing and using the `OfficeHoursStatus` function first may be useful.

- `void OfficeHoursStatus(struct OfficeHoursStats* resultStats)`:

  In this helper function, you will update the current Office Hour status.

  - If there are no one left in the queue, the Office Hour status is `Completed`, otherwise the status is `InProgress`

What is `power_and_mod`?

The `power_and_mod` helper does the calculation of the power and modulus calculation without any potential overflow. It takes a base `b`, an exponent `e`, and a modulus `n` and calculates the following:

$$(b^e) \bmod n$$

Since b^e can be very, very big, the `power_and_mod` function does this calculation with some modulo tricks to prevent overflow. Note that it is not necessary to understand how the helper function works.

2.3 .h files

A header file is a C file (by convention with the extension .h) that contains function prototypes, struct definitions, as well as macros. Header files are useful so that we can separate these declarations and definitions from our main C code and later include them in other files. You can see in the code that `oh_queue.c` includes `oh_queue.h`, its header file.

Before getting started with this homework make sure to get familiar with what's provided in `oh_queue.h`. Here is some of what's defined in `oh_queue.h`:

- `struct Student`: This struct definition is how students are represeted in the queue system.
- Prototypes for functions like `push` and `pop` in `oh_queue.c`. By including these at the top of `oh_queue.c`, we prevent errors from using a function before it is defined.
- Macros for constants such as `MAX_QUEUE_LENGTH`, which is defined as 30, the maximum length the office hour queue can be.
- `UNUSED_PARAM(x)` and `UNUSED_FUNC(x)`: Macros that are used in `oh_queue.c` as placeholders so that you can compile the file without needing to complete every function. You may remove these once you've completed a function.

3 Useful Tips

3.1 Man Pages

The `man` command in Linux provides "an interface to the on-line reference manuals." If you are unsure about the specifics of a my_string.c function, you should look up the exact details using its man page. This is a great utility for any C and Linux developer for finding out more information about the available functions and libraries. In order to use this, you just need to pass in the function name to this command within a Linux (in our case Docker) terminal.

For instance, entering the following command will print the corresponding man page for the strlen function:

```
$ man strlen
```

Additionally, the man pages are accessible online at: http://man.he.net

NOTE: You can ignore the subsections after the "RETURN VALUE" (such as ATTRIBUTES, etc) for this homework, however, pay close attention to function descriptions.

3.2 Debugging with GDB and printf

We highly recommend getting used to "`printf` debugging" in C early on.

Moreover, If you run into a problem when working on your homework, you can use the debugging tool, GDB, to debug your code! Former TA Adam Suskin made a series of tutorial videos which you can find here.

Side Note: Get used to GDB early on as it will come in handy in any C program you will write for the rest of 2110, and even in the future!

When running GDB, if you get to a point where user input is needed, you can supply it just like you normally would. When an error happens, you can get a Java-esque stack trace using the backtrace(bt) command which allows you to pinpoint where the error is coming from. For more info on basic GDB commands, search up "GDB Cheat Sheet."

4 Checking Your Solution

4.1 Makefiles

Make is a common build tool for abstracting the complexity of working with compilers directly. In fact, the PDF you're reading now was built with a Makefile! Makefiles let you define a set of desired targets (files you want to compile), their prerequisites (files which are needed to compile the target), and sets of directives (commands such as gcc, gdb, etc.) to build those targets. In all of our C assignments (and also in production level C projects), a Makefile is used to compile C programs with a long list of compiler flags that control things like how much to optimize the code, whether to create debugging information for gdb, and what errors we want to show. We have already provided you a Makefile for this homework, but we highly recommend that you take a look at this file and understand the `gcc` commands and flags used to understand how to compile C programs. If you're interested, you can also find more information regarding Makefiles here.

Since your program is connected to an autograder with multiple files that need to be compiled using particular settings, it's a little difficult to compile it by hand. The Makefile allows us to simply type the command `make` followed by a target such as `hw7`, `tests`, `run-case`, or `run-gdb` to compile and run your code.

4.2 Autograder

To test your code manually, compile your code using `make` and run the resulting executable file with the command-line arguments of your choice.

Keep in mind that you should run all commands inside the Docker terminal in the same directory as the Makefile. You may also run the usual script with -it to get a terminal inside Docker without needing to use the browser window:

```
./cs2110docker.sh -it
```

If you use your own Linux distribution/VM, make sure you have the `check` *unit test framework installed. However, keep in mind that your code will be tested on Docker.*

To run the autograder locally (without GDB):

```
# To clean your working directory (use this instead of manually deleting .o files)
$ make clean

# Compile all the required files
$ make tests

# Run the tester executable
$ ./tests
```

The above commands will run all the test cases and print out a percentage, along with details of the **failed test cases**. If you want to debug a failed test case, see below.

Commands to run/debug a specific failing test case:

- To run specific tests without gdb:

  ```
  # Run all tests
  $ make run-case
  ```

```
# Run a specific test
$ make run-case TEST=testCaseName
```

- To run specific tests with gdb:

```
# Run all tests in gdb
$ make run-gdb

# Run a specific test in gdb
$ make run-gdb TEST=testCaseName
```

```
Example error message: suites/hw7_suite.c:960:F:test_compareUser_basic_equal: ...
Example command: make run-caseTEST=test_compareUser_basic_equal
```

TA Tip: Since C autograders can sometimes print out a lot of info, it might be a good idea to pipe the output to a file (`./tests > output.txt`) and investigate the content of the file instead! Use Gradescope for a cleaner output or run tests individually when debugging as mentioned above.

4.3 Manual Testing

If you want to write manual tests of your functions, you are allowed to modify `main.c` to treat it as your own driver program. For example, you may want to create a new helper method that will print out the contents of the student array within `oh_queue.h`, implement it in `oh_queue.c`, and call it in `main.c`. However, if you choose to create extraneous helper methods to help debug **make sure to remove them when submitting to the autograder**. Editing `main.c`, however, should not interfere with the autograder.

Here is how to manually run your `main.c` code.

```
# Clean up all compiled output
$ make clean

# Recompile the hw7 executable
$ make hw7

# Run the hw7 executable
$ ./hw7
```

Important Notes:

1. The output file will **ONLY** be graded on Gradescope.
2. All non-compiling homework will receive a zero (with all the flags specified in the Makefile/Syllabus).
3. **NOTE: DO NOT MODIFY THE HEADER FILES.**

   You must place any code elements you define (structs, macros, function declarations, etc.) in the C FILES. Usually placing those definitions in `.h` files would be good practice, but for this assignment you are not turning them in, and so the declarations would be lost when submitting.

Many test cases are randomly generated and your code should work every time we run the autograder on it, however, there's no need to submit to Gradescope multiple times once you get the desired grade.

We reserve the right to update the autograder and the test case weights on Gradescope or the local checker as we see fit when grading your solution.

5 Deliverables

Please upload the following files to Gradescope:

1. `my_string.c`
2. `oh_queue.c`

Note: Please do not wait until the last minute to run/test your homework; history has proven that last minute turn-ins will result in long queue times for grading on Gradescope.

6 Appendix

6.1 Appendix A: The RSA Algorithm

The RSA algorithm is a public-key encryption scheme where a public key is used to encrypt a plaintext message and a private key is used to decrypt a ciphertext message. The plaintext is a string of characters while the ciphertext is a sequence of numbers that each corresponds to a character in the plaintext. We will denote the public key to be (n, e) and the private key to be (n, d). Through some discrete math magic (that is not required to know), a public key must follow certain math properties such that only one corresponding private key can be used for decrypting a message encrypted with that public key.

To encrypt a plaintext message, follow these steps:

1. Convert each of the characters to a corresponding number. In this homework, you will use **the ASCII value of the character** as the mapping, so 'A' would correspond to 65, 'B' would correspond to 66, etc.
2. Use the public key (n, e) to apply power and mod to each of the numbers you got in the previous step. If we have a number m corresponding to the plaintext, we do the following:
$$(m^e) \bmod n = c$$
3. The new values c that we got for each character in the previous step is the encrypted number sequence (the final ciphertext). Each number in this encrypted number sequence corresponds to the character value m in the plaintext.

To decrypt a ciphertext message, follow these steps:

1. Use the private key (n, d) to apply power and mod to each of the numbers in the ciphertext. If we have a ciphertext number c, we do the following:
$$(c^d) \bmod n = m$$
2. The new value m that we got in the previous step is the decrypted number that corresponds to the ciphertext number c.
3. Convert each of the numbers m to their corresponding characters. In this homework, you will treat the new value m as **an ASCII value** and convert it back to the character value.

Lets go through an example. We have a plaintext 'Cat' that we would like to encrypt with the public key $(143, 23)$. We first convert each of the characters in the message to their corresponding number (the ASCII value in this case), so we get 67 97 116. With the public key $(143, 23)$, we then apply the encryption formula to each number.

$$(67^{23}) \bmod 143 = \mathbf{111}$$
$$(97^{23}) \bmod 143 = \mathbf{102}$$
$$(116^{23}) \bmod 143 = \mathbf{51}$$

With the public key $(143, 23)$, the encrypted message for 'Cat' is 111 102 51. We can then use the private key $(143, 47)$ to decrypt this encrypted message back to 'Cat' by applying the decryption formula for each number.

$$(111^{47}) \bmod 143 = \mathbf{67}$$
$$(102^{47}) \bmod 143 = \mathbf{97}$$
$$(51^{47}) \bmod 143 = \mathbf{116}$$

The decrypted number sequence is 67 97 116. Converting this number sequence to characters with ASCII, we get the original message: 'Cat'.

6.2 Appendix B: Rules and Regulations

6.2.1 General Rules

1. Starting with the assembly homeworks, any code you write should be meaningfully commented for your benefit. You should comment your code in terms of the algorithm you are implementing; we all know what each line of code does.
2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.
3. Please read the assignment in its entirety before asking questions.
4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
5. If you find any problems with the assignment, it would be greatly appreciated if you reported them to the TAs. Announcements will be posted if the assignment changes.

6.2.2 Submission Conventions

1. Unless otherwise noted, all files you submit for assignments should have your name somewhere near the top of the file as a comment.
2. When preparing your submission, you may submit the files individually to Canvas/Gradescope. You can create an archive by right clicking on files and selecting the appropriate compress option on your system. Both ways (uploading raw files or an archive) are exactly equivalent, so choose whichever is most convenient for you.
3. Do not submit compiled files (`.class` files for Java code or `.o` files for C code). Only submit the files we ask for in the assignment.
4. Do not submit links to files. The autograder will not understand it, and we will not manually grade assignments submitted this way, as it is easy to change the files after the submission period ends.

6.2.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes accounting for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time, and provide documentation (i.e. note from the dean, doctor's note, etc.). Extensions will only be granted to those who contact us in advance of the deadline, and no extensions will be made after the due date.
2. You are responsible for ensuring that you have turned in the correct files. After submitting, be sure to download your submission into a brand new folder and test that it works. What you turn in is what we grade; there are no excuses if you submit the wrong files. In addition, your assignment must be turned in via Canvas/Gradescope. Under no circumstances whatsoever will we accept any email submissions of assignments (Note: if you were granted an extension, you will still turn in the assignment over Canvas/Gradescope).

6.2.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative. In addition, many, if not all, homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be programatically examined to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be written by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be referred to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply emailing it to them so they can look at it. If you supply an electronic copy of your homework to another student, and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code, including, but not limited to, public repositories (GitHub), Pastebin, etc. If you would like to use version control, use `github.gatech.edu`.

6.2.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code. What you shouldn't be doing, however, is pair programming, where you collaborate with each other on a single instance of the code. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, because it is frequently the case that the recipient will simply modify the code and submit it as their own.

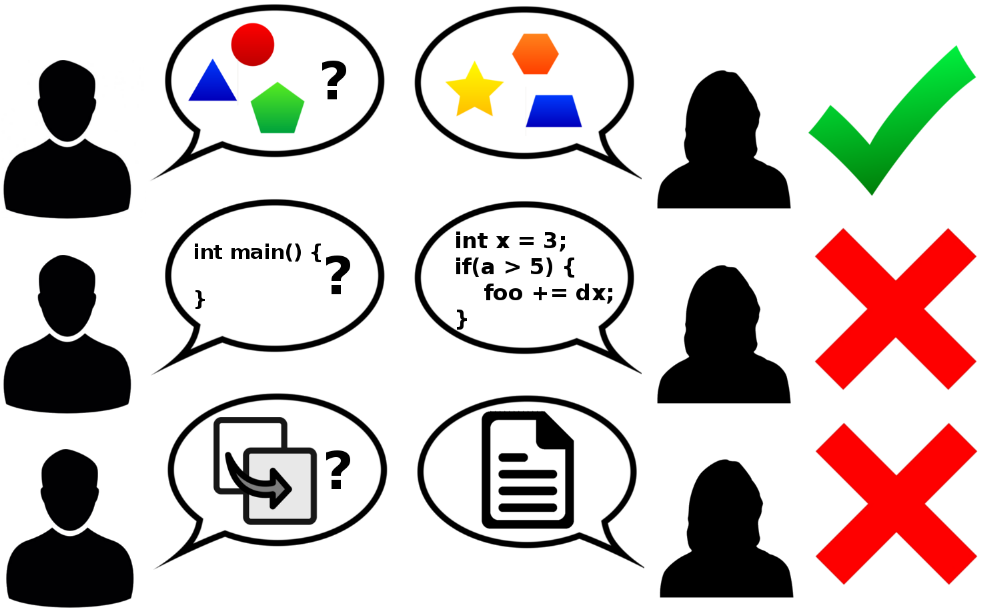

Figure 1: Collaboration rules, explained colorfully