CS 2110 Homework 7
Intro to C

Richard Zhang, Sophie Imhof, Adrien Cao, Eric Stuhr, Dayne Bergman, Makiyah Dee

Version 1.0

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 be the (technologically savvy) manager of your company's user database, using your knowledge of the C programming language to service and manage your company's users.

You will write C functions to add, remove, and update users, aggregate data on users' compensation, as well as sort the users in your database. Finally, you will write C functions to encrypt and decrypt users as your company migrates to a more secure database. For some functions you will have to return whether the function executed successfully or unsuccessfully. See the Detailed Instructions section for more details on the specific requirements for each function.

You will write your C code in three files; `my_string.c`, `cryptography.c`, and `users.c`. We recommend implementing the functions in these files in this order. 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()`, and `my_memset()`. In `cryptography.c`, you will implement the functionality of encrypting and decrypting data. In `users.c`, you will implement the functionality as mentioned above regarding the company database.

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

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.

1.3 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*.

You will notice that many of the arguments in the `my_string.c` and `users.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 `users.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.

2.2 `cryptography.c`

The second part of this homework is to implement several cryptography algorithms in C within the `cryptography.c` file. The cryptography algorithms you will be implementing include RSA. The exact RSA algorithm you will be implementing is in Appendix A. **It is important that you follow the exact algorithm specification in Appendix A**. 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!

What are `struct public_key` and `struct private_key`?

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;
};
```

For decryption, RSA depends on a private key represented by two numbers (n, d). This two-number pair is represented by the `struct private_key` struct defined as:

```
struct private_key {
```

```
    int n;
    int d;
};
```

Functions to Implement:

- `encrypt`: encrypts through the RSA algorithm a plaintext (decrypted) null-terminated string and writes into a ciphertext (encrypted) integer array. It is assumed that `encrypt` will be provided a ciphertext integer array sufficiently large enough to hold the encrypted text of the plaintext null-terminated string. Note that the null terminator does not have to be encrypted as the length of the original plaintext will be passed around when decrypting back to the original plaintext.
  - **IMPORTANT:** Since there is potential overflow with the large numbers being used as exponents, you should use the helper `power_and_mod` function to do the RSA encryption for each character in the plaintext. This helper function does the power and mod calculations without any potential overflow.
  - The ASCII values of the characters in the string will be used for the character-to-number mapping in RSA.
  - Use the provided `struct public_key` struct to do the encryption.
- `decrypt`: decrypts through the RSA algorithm a ciphertext (encrypted) integer array and writes into a string that is null-terminated. Since there is no null-terminator equivalent for integers, the length of the original plaintext must be passed in.
  - **IMPORTANT:** Since there is potential overflow with the large numbers being used as exponents, you should use the helper `power_and_mod` function to do the RSA decryption for each integer in the ciphertext. This helper function does the power and mod calculations without any potential overflow.
  - The ASCII values of the characters in the string will be used for the character-to-number mapping in RSA.
  - Use the provided `struct private_key` struct to do the decryption.

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 `users.c`

The final part of this homework is to implement several C functions within the users.c file. You will be primarily interacting with the global `users` array as well as the global `size` variable. The `users` array must not have gaps between users (must be contiguous). You will use and update the size variable to keep track of the current number of users within the `users` array.

What is a `struct decrypted_user`:

Each user in the array is represented using the `struct decrypted_user` struct defined in `users.h`:

```
struct decrypted_user {
    int id;
```

```
        char name[MAX_NAME_LENGTH];
        char password[MAX_PASSWORD_LENGTH];
        double salary;
        char company[MAX_COMPANY_LENGTH];
        struct public_key pub_key;
    };
```

Each `struct decrypted_user` contains six member variables:

- `id` is an integer that uniquely identifies each user, it is guaranteed that no two users have the same id.
- `name` is a string (represented as a null-terminated array of characters) which stores the name of the user's name (e.g. [‘S’,‘h’,‘a’,‘w’,’n’,‘\0’]).
- `password` is a string (represented as a null-terminated array of characters) which stores the user's password (e.g. [‘p’,‘a’,‘s’,‘s’,‘1’,‘2’,‘3’,‘@’,‘#’,‘#’,‘\0’]).
- `salary` is a double which represents the salary of the user.
- `company` is a string (represented as a null-terminated array of characters) which stores the user's company (e.g. [‘C’,‘o’,‘C’,‘\0’]).
- `pub_key` is a `struct public_key` struct (with members `int n` and `int e`) that stores the user's public key used for RSA encryption.

What is the `users` array:

The `users` array is defined at the top of `users.c`. `users` is an array of all the users in the database, and contains `struct decrypted_user` structs as its elements:

```
    struct decrypted_user users[MAX_USERS_LENGTH];
```

Whenever "`users` array" is mentioned in this pdf, it is referring to this array. All of the functions described below either read or write to this array in some way.

What is `size` v.s. `MAX_USERS_LENGTH`:

`size` is defined at the top of `users.c` as a global integer variable. `size` stores the number of users currently in the `users` array:

```
    int size = 0;
```

Whenever "`size`" is mentioned in this pdf, it is referring to this variable. If any of the functions described below add or remove users from size, you are responsible for also updating `size` to match the new number of users in the `users` array.

What is a `struct encrypted_user`:

When implementing the `encryptUser` and `decryptAndAppendUser` functions, you will be given an `struct encrypted_user` struct defined in `users.h`:

```
    struct encrypted_user {
        int id;
        int name[MAX_NAME_LENGTH];
        int nameLength;
        int password[MAX_PASSWORD_LENGTH];
        int passwordLength;
        double salary;
```

```
    int company[MAX_COMPANY_LENGTH];
    int companyLength;
    struct public_key pub_key;
};
```

Each `struct encrypted_user` contains nine member variables:

- `id` is an integer that uniquely identifies each user, it is guaranteed that no two users have the same id.
- `name` is an integer array which stores the encrypted name of the user represented by integers (e.g. [231,323,93,17,87]).
- `nameLength` is an integer that stores the length of the user's name that was encrypted.
- `password` is an integer array which stores the encrypted password of the user represented by integers (e.g. [12,21,45,23,23]).
- `passwordLength` is an integer that stores the length of the user's password that was encrypted.
- `salary` is a double which represents the salary of the user.
- `company` is an integer array which stores the encrypted password of the user represented by integers (e.g. [134,98,17,12,23]).
- `companyLength` is an integer that stores the length of the user's password that was encrypted.
- `pub_key` is a `struct public_key` struct (with members `int n` and `int e`) that stores the user's public key used for RSA encryption.

It is highly recommended that you use the `encrypt` and `decrypt` functions in `cryptography.c` to encrypt `struct decrypted_user` structs and decrypt `struct encrypted_user` structs.

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_USERS_LENGTH` represents the maximum number of users that can be stored in the `users` array.
- `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_PASSWORD_LENGTH` represents the maximum length of the password `char` array, including the null-terminator. Remember char arrays are one way to represent strings in C.
- `MAX_COMPANY_LENGTH` represents the maximum length of the company `char` array, including the null-terminator. Remember char arrays are one way to represent strings in C.

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`, `MAX_COMPANY_LENGTH`, and `MAX_PASSWORD_LENGTH` represent the maximum lengths of their respective `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 these two macros as being the total amount of characters that have been allocated to represent the name, company, and password, respectively. This means that if the maximum length is 10, for example, then the longest name, company, or password 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, company, or password. This idea is important for truncating strings that are longer than the maximum lengths!

- 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:

- `int addUser(int id, const char * name, const char * password, double salary, const char * company, struct public_key pub_key)`:

  In this function, you will add a `user` struct with the given `id`, `name`, `password`, `salary`, `company`, and `public_key` to the end of the `users` array.

  - 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!
  - Similarly, ensure the new user's password and company do not exceed `MAX_PASSWORD_LENGTH` and `MAX_COMPANY_LENGTH` respectively.
  - If the user's `name`, `password`, or `company` length are 0, or adding the user would make `size` exceed `MAX_USER_LENGTH`, do not create or add the user to `users`.
  - Return `SUCCESS` if you are able to successfully create and add the user, otherwise return `FAILURE`.

- `int updateUserCompany(struct decrypted_user user, const char *company)`:

  In this function, you will update a user so that their company is updated to the `company` parameter passed into this function. The user to be updated will be the user in the users array with the same `id` as the passed in `user`. **It is guaranteed that no two users have the same id.**

  - If the updated company's length (including the null terminator) is above `MAX_COMPANY_LENGTH`, truncate the string to be `MAX_COMPANY_LENGTH` (including the null terminator).
  - If you are able to successfully update the user's company, return `SUCCESS`, if you are unable to find the user based on its id, return `FAILURE`.

- `double averageSalary(const char *company)`:

  In this function, you will calculate the average salary of all the users in your users array with the same company as the passed in `company` parameter so that management can compare them. You should return the average `salary` of these users as a double.

  - If there are no users in `users` who match the passed in company, you should return `0`.

- `int swapUsers(int index1, int index2)`:

  In this function, you will swap the position of a user at `index1` with the position of the user at `index2`.

- If the indices are negative or beyond the current size of the `users` array, do not swap.
- If the users were successfully swapped, return `SUCCESS`. Otherwise, return `FAILURE`;

- `int removeUser(struct decrypted_user user)`:

  In this function, you will remove a user with the same `id` as the passed in `user` struct (note: each user has a unique `id`). **You must maintain array contiguity when removing**. This means that there must be no gaps between users within the `users` array after removing. You must also maintain the current relative ordering of the users array after removal.

  - If there are no users in the `users` array with the same id as the one passed in to this function, do not remove any users.
  - On successful removal, return `SUCCESS`. If you cannot find the user with the given id, return `FAILURE`.

- `int compareName(struct decrypted_user user1, struct decrypted_user user2)`:

  In this function, you will compare two users by which comes first lexicographically by name. If `user1`'s name is less than `user2`'s name, return a negative number. If it is greater, return a positive number. If `user1`'s name equals `user2`'s name, then return 0 (remember that in lexicographic order, "pasture" is less than "pastureland").

- `void sortUsersByName(void)`:

  In this function, you will sort the `users` array using any sorting algorithm of your choice. The result should be in ascending order. A user comes before another user if its name is less than the other's according to `compareName`. If two users have the same name, then the user with the greater `salary` should come first. That means that if every user in the `users` array had the same name, then you're essentially sorting their salaries in descending order. If two users have the same name and salary, then they are considered equal and either can come first (It will be helpful to use `compareName` and `swapUsers` as helper functions).

- `int encryptUser(struct encrypted_user *encrypted_user, int index)`

  In this function, you will take the `struct decrypted_user` struct at the `index` in the `users` array and use RSA encryption to encrypt the `name`, `password`, and `company` with the user's `pub_key`. The encrypted `name`, `password`, and `company` will be written into the `struct encrypted_user` pointed by the `encrypted_user` parameter. The `salary`, `id`, and `pub_key` should be the same and the members describing length in the `struct encrypted_user` struct should be equal to their respective members in the `struct decrypted_user` struct (i.e. `nameLength` in `struct encrypted_user` should be the same as the length of `name` in `struct encrypted_user`). It is highly recommended to use the `encrypt` function you implemented in `cryptography.c`.

  - If the provided `index` is out of bounds or if `encrypted_user` is a NULL pointer, return FAILURE.
  - On successful encryption, return `SUCCESS`.

- `int decryptAndAppendUser(struct encrypted_user *encrypted_user, struct private_key pri_key)`

  In this function, you will take a pointer to an `struct encrypted_user` and decrypt its `name`, `password`, and `company` through RSA. A `struct decrypted_user` struct with these decrypted attributes should then be appended to the end of the `users` array. This appended `struct decrypted_user` struct will have the same `salary`, `id`, and `pub_key` as the original `struct encrypted_user`. For decryption, you will use the provided `struct private_key` struct parameter. It is highly recommended to use the `decrypt` function you implemented in `cryptography.c`.

  - If the provided `struct private_key` struct has a negative `n` or `d` or if `encrypted_user` is a NULL pointer, return `FAILURE`.
  - On successful decryption and append, return `SUCCESS`.

2.4 .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 `users.c` includes `users.h`, its header file.

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

- `struct decrypted_user` and `struct encrypted_user`: These struct definitions are how users are represented in this homework.
- Prototypes for functions like `addUser` in `users.c`. By including these at the top of `users.c`, we prevent errors from using a function before it is defined.
- Macros for constants such as `MAX_NAME_LENGTH`, which is defined as 10, the maximum length a user's name variable can be.
- `UNUSED_PARAM(x)` and `UNUSED_FUNC(x)`: Macros that are used in `users.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:646:F:test_removeUser_basic_1: ...
Example command: make run-gdb TEST=test_removeUser_basic_1
```

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 array within `users.h`, implement it in `users.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. `cryptography.c`
3. `users.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.

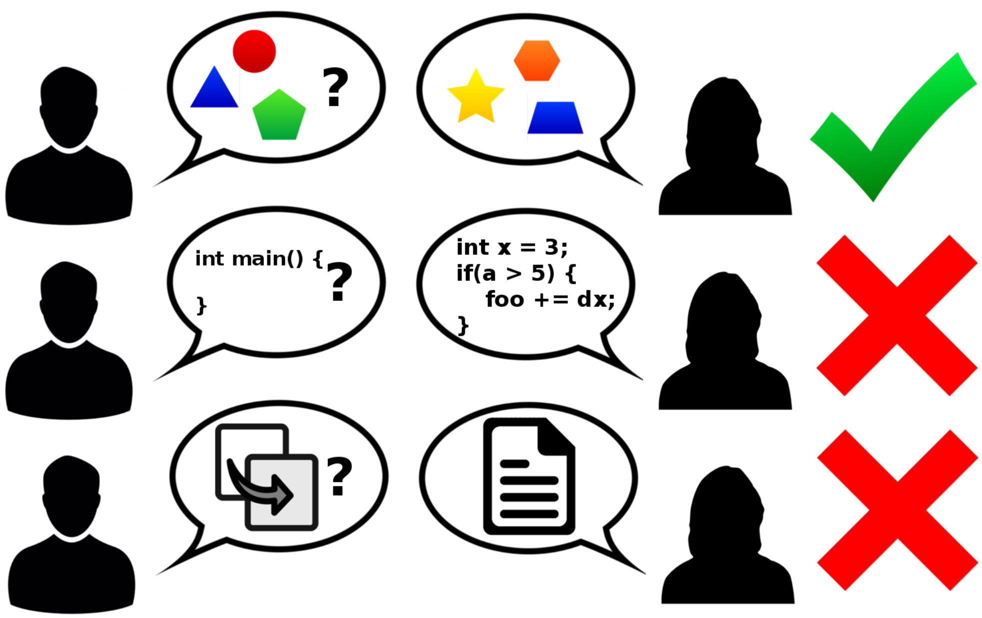

Figure 1: Collaboration rules, explained colorfully