CS 2110 Homework 7
Intro to C

Prabhav Gupta, Henry Bui, Saloni Bedi,
Alex Whitlock, Richard So, Vy Mai

Spring 2024

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

You will write your C code in two files: `my_string.c`, `pkmn_gym.c`.

See the Detailed Instructions section for more details on the specific requirements for each function.

IMPORTANT NOTE:

- **You need to complete 'string.c' before 'pkmn_gym.c' as you will be using the implemented functions from 'string.c' in 'pkmn_gym.c'.**

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()`
- `is_palindrome_ignore_case()`
- `caesar_shift()`
- `deduplicate_str()`
- `swap_strings()`

In `pkmn_gym.c`, you will establish a new Pokémon gym! Your Pokémon gym will have the following functions:

- `catch_pokemon()`
- `release_pokemon()`
- `count_species()`
- `trade_pokemon()`
- `register_trainer()`
- `unregister_trainer()`
- `battle_trainer()`
- `find_champion()`

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!

For this file, you may assume that all inputs are valid (i.e. not NULL).

- `size_t my_strlen(const char *s)`:

  This function calculates the length of the string pointed to by `s`, up to and excluding the null terminator (`'\0'`). Consider the following examples:

  - If `char *pet = "otter"`, then `my_strlen(pet) == 5`. Remember, the null terminator is automatically 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 (i.e., unterminated strings) because the programmer is expected to pass in valid strings.

- `int my_strncmp(const char *s1, const char *s2, size_t n)`:

  This function compares two strings, checking a maximum of `n` characters of both strings. Note that:

  - Comparisons should be made between each character between the ASCII values of each pair of corresponding characters.
  - Comparisons should be character by character until one of the following occurs:
    * There is a difference between the corresponding pair of characters. (This includes if one of the strings terminates before the other one!)
    * Both strings terminate at the same character.
    * We have completed `n` comparisons.

  This function should return an arbitrary negative integer if the first string is lexicographically less than the second string, zero if equal, and positive if greater. You should also 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("a\0a", "a\0b", 3) == 0` (notice how and where an additional null terminator is placed mid-string!)

- `char *my_strncpy(char *dest, const char *src, size_t n)`:

  This function copies at most `n` characters from a source string (`src`) into a destination memory location (`dest`), returning a pointer to `dest`. Note that:

  - If `src` ends before `n` characters have been copied to `dest`, then continue writing null characters to `dest` until `n` characters have been written.

- If `src` has not ended after `n` characters have been copied to `dest`, a null terminator should not be added to `dest`.
- As an implementor, you can assume `dest` is sufficiently large enough for `n` characters to be written into it (note that if you *use* this function, you have to ensure the `dest` argument you enter is sufficiently large enough!).

Consider the following example. Let us define the following variables:

```
char src[]  = "hob";
char dest[] = "ribbit";
```

Then,

- `my_strncpy(dest, src, 3)` would set dest to `{'h', 'o', 'b', 'b', 'i', 't', '\0'}`.
- `my_strncpy(dest, src, 4)` would set dest to `{'h', 'o', 'b', '\0', 'i', 't', '\0'}`.
- `my_strncpy(dest, src, 5)` would set dest to `{'h', 'o', 'b', '\0', '\0', 't', '\0'}`.

- `char *my_strncat(char *dest, const char *src, size_t n)`:

  This function appends at most `n` bytes from the `src` string to the `dest` string, returning a pointer to `dest`. This function starts writing characters over `dest`'s null terminator and stops writing to `dest` once:

  - a null terminator is found in `src`, or
  - `n` characters have been written from `src`. In this case, a null terminator is added to the end of `dest`.

  Note that, like `my_strncpy`, as the implementor of this function, you can assume `dest` is sufficiently large enough for the result. (If you use this function, you must provide a sufficiently large enough `dest` to hold the result of this function!)

  Consider the following example. Let us define the following variables:

  ```
  char src[6]    = "defgh";
  char dest[100] = "abc";
  ```

  Then (assuming we set `src` and `dest` back to these inputs after every execution),

  - `my_strncpy(dest, src, 3)` sets dest to `{'a', 'b', 'c', 'd', 'e', 'f', '\0' }`
  - `my_strncpy(dest, src, 5)` sets dest to `{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', '\0' }`
  - `my_strncpy(dest, src, 6)` also sets dest to `{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', '\0' }`
  - `my_strncpy(dest, src, 9)` also sets dest to `{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', '\0' }`

- `void *my_memset(void *str, int c, size_t n)`:

  This function fills the first `n` bytes of the memory area pointed to by `str` with the constant int value `c`. The function returns a pointer to the memory location `str`.

  Note that this function does not copy the int value `c` directly into `str`, but rather, truncates it to a byte before copying it.

  As an implementor, you can assume the `str` buffer has enough space to write `n` bytes into (If you use this function, you must ensure you provide a `str` buffer that has enough space to write `n` bytes into). This function does not add a null terminator at the end.

  Consider this example. Suppose you had a destination buffer `char str[] = "i love aardvarks!"`:

- `my_memset(str, 'a', 1)` updates `str` to `"a love aardvarks!"`
- `my_memset(str, 'a', 7)` updates `str` to `"aaaaaaaaardvarks!"`
- `my_memset(str, 0x9961, 16)` updates `str` to `"aaaaaaaaaaaaaaaa!"`. Note here that `0x9961` truncates to `0x61` which encodes the character `'a'`.

- `int is_palindrome_ignore_case(const char *str)`:

  This function checks whether a string is a palindrome, ignoring differences in case. It should return 1 if the input string `str` is a palindrome, and it should return 0 if it is not.

  Consider the following examples of palindromes: racecar, madam, ABBA, Ho-Oh, Eevee, Girafarig. Note that with the last three words, they are only palindromes if you ignore case!

- `void caesar_shift(char *str, int shift)`:

  This function applies a Caesar shift of size `shift`, updating the input *str* with the encrypted text.

  A Caesar shift or Caesar cipher is a type of encryption method where each letter of a message is substituted for a letter that is a fixed distance away in the alphabet.

  For example, with a shift of +2:

original	A	B	C	D	E	F	...	U	V	W	X	Y	Z
shifted	C	D	E	F	G	H	...	W	X	Y	Z	A	B

  With this table, we could convert the sentence "I love LC-3 assembly!" into "K nqxg NE-3 cuugodna!".

  Note that uppercase letters should be transformed into other uppercase letters, lowercase letters should be transformed into other lowercase letters, and other types of characters (e.g., numbers and punctuation) remain unchanged.

- `void deduplicate_str(char *str)`:

  This function removes consecutive duplicate characters in the string. This function is case-sensitive, so characters that are considered "duplicate" must also have the same case.

  Consider these examples:

  - if `char foo[] = "mississippi"`, then calling `deduplicate_str(foo)` should update `foo` to `"misisipi"`.
  - if `char bar[] = "bookkeeper"`, then calling `deduplicate_str(bar)` should update `bar` to `"bokeper"`.
  - if `char baz[] = "Zzyzx"`, then calling `deduplicate_str(bar)` should keep `baz` as `"Zzyzx"`, as there are no consecutive duplicate characters.

- `void swap_strings(char **s1, char **s2)`:

  This function swaps two string variables. You should not use loops (i.e., `for`, `while`, `do-while`, or `goto`) to implement this function. Note that you are given pointers to strings (`char **`).

  Here is an example of how this function would be used:

```
char *right_opinion = "CS 2110 is a terrible, horrible, no good, very bad class.";
char *wrong_opinion = "CS 2110 is such a fun class!";

swap_strings(&right_opinion, &wrong_opinion);

printf("%s\n", right_opinion); // CS 2110 is such a fun class!
printf("%s\n", wrong_opinion); // CS 2110 is a terrible, horrible,
                               // no good, very bad class.
```

You will notice that many of the arguments in the `my_string.c` and `pkmn_gym.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 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 is large enough to store the result of `sizeof(...)`. The `size_t` type is used throughout the C standard library (which, of course, includes the string library functions).

Some notes:

1. You should complete `my_string.c` before moving on to `pkmn_gym.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 `swap_strings`, 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 pkmn_gym.c

The second part of this homework is to implement several C functions within the `pkmn_gym.c` file.

You are a Pokémon gym manager in the region of WeAreTakingSignificantCreativeLibertiesToMakeThisIdeaWorkForTheHomeworkAssignmentia (or Watsia for short)! You need to keep your gym working and functional in case a 10-year-old prodigy with the name of a color wants to battle at your gym! You will primarily be interacting with `Trainer`s in the gym and with the global `gym` struct to facilitate the operations of your gym.

In your gym, you have trainers that are registered to your gym to train! Your `struct Gym` is defined as such in `pkmn_gym.h`:

```
struct Gym {
    struct Trainer trainers[MAX_TRAINER_LENGTH];
    int num_trainers;
};
```

Your gym keeps track of the following information:

- `trainers` is an array which keeps track of all of the trainers currently registered at the gym,

- `num_trainers` is the size of the array, keeping track of how many trainers are currently registered at the gym.

The trainers of your gym are represented by the following struct:

```
struct Trainer {
    char name[MAX_NAME_LENGTH];
    struct Pokemon party[MAX_PARTY_LENGTH];
    int party_size;
    int num_wins;
};
```

Each `struct Trainer` tracks the following information:

- `name`: The name of the trainer,
- `party`: The Pokémon the trainer has (maximum of 6!),
- `party_size`: The number of Pokémon the trainer has in their party,
- `num_wins`: The number of wins they've had against other trainers.

Finally, each Pokémon is represented by the following struct:

```
struct Pokemon {
    char species[MAX_NAME_LENGTH];
    int level;
};
```

Each Pokémon (`struct Pokemon`) tracks the following information:

- `species`: The name of the species of Pokémon (e.g., Girafarig, Ho-Oh, Alomomola),
- `level`: The level of the Pokémon (ranges from 1 to 100 (inclusive)).

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/species `char` arrays, including the null terminator. Remember char arrays are one way to represent strings in C.
- `MAX_PARTY_LENGTH` represents the maximum length of the `party` array.
- `MAX_TRAINER_LENGTH` represents the maximum length of the `trainers` 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 definition, 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!

- Remember you can index into strings for reading or writing characters. For example, `some_string[3]` is the 4th character of `some_string`. In this file, you are allowed to use array notation.
- When using any of the functions that write to a destination buffer (e.g., `my_strncpy`, `my_strncat`, `my_memset`), ensure that your `dest` buffer has enough capacity for that function. As examples,
  - For `my_strncpy` and `my_memset`, `n` should never exceed `sizeof(dest)` (supposing that `dest` is a char array).
  - For `my_strncat`, `n` should never exceed `sizeof(dest) - strlen(dest)` (also supposing that `dest` is a char array).
- 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.
- Remember all strings are to be null-terminated! Otherwise, we would not be able to tell what characters are actually part of a string.
- **Always check the validity of your parameters.** Possible things to check for:
  - `NULL` values
  - Strings that are too long
  - Values that are outside the acceptable range

  If you have an invalid input, the function should not do anything, and return `FAILURE`.

Functions to Implement:

Now that you've (hopefully) read all of the gym descriptions, here's all the functions you must implement for your gym!

- `int register_trainer(const char *name)`:

  You need to allow trainers to join your gym. Implement a function that adds a trainer to the trainer array in your gym.
  - The trainers in the gym's trainer array should remain contiguous before and after the operation and should start at index 0.
  - Since this trainer is new to the gym, they should not have any wins or Pokémon.
  - If the gym is full (i.e., if there are `MAX_TRAINER_LENGTH` trainers in the gym), do not add the new trainer and return `FAILURE`.
  - Otherwise, add the trainer and return `SUCCESS`.
- `int unregister_trainer(const char *name)`:

  You also need to allow trainers to leave your gym. Implement a function that finds the trainer with the provided `name` and removes them from the trainer array in your gym.
  - The trainers in the gym's trainer array should remain contiguous before and after the operation and should start at index 0.
  - If no trainer in the gym has the specified name, do not remove any trainer and return `FAILURE`.
  - If only one trainer in the gym has the specified name, remove the trainer and return `SUCCESS`.
  - If more than one trainer in the gym has the specified name, remove the first trainer in the array with the specified name and return `SUCCESS`.
- `int catch_pokemon(struct Trainer *trainer, const char *species, int level)`:

  The trainers from your gym really like catching Pokémon. You must implement a function to add a Pokémon with the given `species` and `level` to the given `trainer`'s party.

- The Pokémon in the given trainer's party array should remain contiguous before and after the operation and should start at index 0.
- If a trainer's party is full (i.e., they have `MAX_PARTY_LENGTH` Pokémon in their party), do not add the Pokémon and return `FAILURE`.
- Remember that the valid range of any Pokémon's level is between 1 and 100 (inclusive). If the argument does not match this constraint, return `FAILURE`.
- If the Pokémon is successfully added, return `SUCCESS`.

- `int release_pokemon(struct Trainer *trainer, const char *species)`:

  It hurts to say goodbye... and some of your trainers will do it anyway. You must implement a function to remove a Pokémon with the given `species` from the given `trainer`'s party.

  - The Pokémon in the given trainer's party array should remain contiguous before and after the operation and should start at index 0.
  - If no Pokémon in the given trainer's party has the specified species, do not remove any Pokémon and return `FAILURE`.
  - If only one Pokémon in the trainer's party has the specified species, remove the Pokémon and return `SUCCESS`.
  - If more than one Pokémon in the given trainer's party has the specified species, remove the first Pokémon in the array with the specified species and return `SUCCESS`.

- `int count_species(const char *species)`:

  You want to compile useful statistics for your gym! Implement a function to count the total number of Pokémon of a given species in your gym.

  - This should count and return the total number of Pokémon of the given species in each trainer's party.

- `int trade_pokemon(struct Trainer *t0, int party_index_0, struct Trainer *t1, int party_index_1)`:

  Your trainers want to be able to exchange their Pokémon! Implement a function to swap the Pokémon in one trainer's party with the Pokémon in another trainer's party.

  - The first Pokémon to be traded is the Pokémon at index `party_index_0` of trainer `t0`'s party. If this index is out of bounds for the party array, do not perform a trade and return `FAILURE`.
  - The second Pokémon to be traded is the Pokémon at index `party_index_1` of trainer `t1`'s party. If this index is out of bounds for the party array, do not perform a trade and return `FAILURE`.
  - Trainers unfortunately can't trade with themselves. If `t0` and `t1` are the same trainer, do not perform a trade and return `FAILURE`.
  - After all conditions have been checked and verified, perform the trade and return `SUCCESS`.

  For example, suppose `t0`'s party consisted of:

Index	0	1	2	3	4	5
Pokémon	Eevee	Ho-Oh	Girafarig	Alomomola	Farigiraf	-
Level	20	100	25	11	26	-

  and suppose `t1`'s party consisted of:

Index	0	1	2	3	4	5
Pokémon	Dunsparce	Haunter	Trubbish	Shellos	Vulpix	Rotom
Level	14	22	11	15	17	14

If `t0` wished to trade their Ho-Oh for `t1`'s Trubbish, we would call `trade_pokemon(t0, 1, t1, 2)`.

- `int battle_trainer(struct Trainer *challenger, struct Trainer *opponent)`:

  The trainers of your gym LOVE to battle. Implement a function that allows a `challenger` to initiate a battle with an `opponent`.

  In a battle, the two trainers complete a set of "matches" between Pokémon of the each index of both `challenger` and `opponent`'s parties. The trainer with higher-leveled Pokémon wins the match.

  The winner of the most Pokémon matches win the battle!

  For example, suppose the `challenger` has this party:

Pokémon	Magikarp	Magikarp	Magikarp	Magikarp	Magikarp	Magikarp
Level	100	14	29	75	44	91

  and suppose the `opponent` has this party:

Pokémon	Kyogre	Ho-Oh	Zekrom	Dialga	Mewtwo	Yveltal
Level	96	50	34	73	40	80

  Then, the battles commence like so:

  1. **Magikarp (lv. 100)** vs. Kyogre (lv. 96)
  2. Magikarp (lv. 14) vs. **Ho-Oh (lv. 50)**
  3. Magikarp (lv. 29) vs. **Zekrom (lv. 34)**
  4. **Magikarp (lv. 75)** vs. Dialga (lv. 73)
  5. **Magikarp (lv. 44)** vs. Mewtwo (lv. 40)
  6. **Magikarp (lv. 91)** vs. Yveltal (lv. 80)

  Since the challenger won 4 matches and the opponent only won 2 matches, the challenger wins the battle!

  Some additional notes:

  - If one trainer has more Pokémon than another, the trainer with more Pokémon automatically wins in matches where they do not have competition. For example, if the `challenger` has 4 Pokémon and the `opponent` has 6, the `opponent` automatically wins the 5th and 6th matches of the battle.
  - In a match, if the Pokémon have the same level, treat it as if neither trainer won the match.
  - If the challenger and the opponent win an equal number of matches, the challenger must swallow their pride and give the win to the opponent (i.e., the opponent wins the battle in case of a tie).
  - This function should return the winner of the battle (0 if the challenger wins, and 1 if the opponent wins).
  - This function should also update the respective trainer's `num_wins` field.

- `struct Trainer *find_champion(void)`:

  You, as delusional as you are, believe the next **POKÉMON LEAGUE CHAMPION** of Watsia is in your very gym! Implement a function to find this champion!

  - Find and return a pointer to the trainer that has the most wins in your gym.
  - If several trainers have the maximum number of wins, pick the first in the gym trainer array with the maximum wins.
  - If your gym is empty (oh no!), return `NULL`.

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 `pkmn_gym.c` includes `pkmn_gym.h`, its header file.

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

- `struct Trainer`: This struct definition is how trainers are represented in the gym system.
- Prototypes for functions like `catch_pokemon` and `release_pokemon` in `pkmn_gym.c`. By including these at the top of `pkmn_gym.c`, we prevent errors from using a function before it is defined.
- Macros for constants such as `MAX_PARTY_LENGTH`, which is defined as 6, the maximum number of Pokémon in a trainer's party.
- `UNUSED_PARAM(x)` and `UNUSED_FUNC(x)`: Macros that are used in `pkmn_gym.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. First, enter into our provided docker container with the following commands:

```
# macOS or Linux
./cs2110docker.sh

# Windows
cs2110docker.bat
```

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.*

Navigate to your working directory for HW7, then 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 trainers array within `pkmn_gym.h`, implement it in `pkmn_gym.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.**
   Since you are not turning them in, any changes to the .h files will not be reflected when running the Gradescope autograder.

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. `pkmn_gym.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 Rules and Regulations

1. Please read the assignment in its entirety before asking questions.
2. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
3. If you find any problems with the assignment, please report them to the TA team. Announcements will be posted if the assignment changes.
4. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency please reach out to your instructor and the head TAs **IN ADVANCE** of the due date with documentation (i.e. note from the dean, doctor's note, etc).
5. You are responsible for ensuring that what you turned in is what you meant to turn in. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Gradescope. Email submissions will not be accepted.
6. See the syllabus for information regarding late submissions; any penalties associated with unexcused late submissions are non-negotiable.

6.1 Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work. Homework assignments will be examined using cheat detection programs to find evidence of unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student. If you supply a 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 platform which would allow other parties to it (public repositories, pastebin, etc). If you would like to use version control, use a private repository on github.gatech.edu

Homework collaboration is limited to high-level collaboration. Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment.

High-level collaboration means 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, or providing other students any part of your code.

Submissions that are essentially identical will receive a zero and will be sent 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.

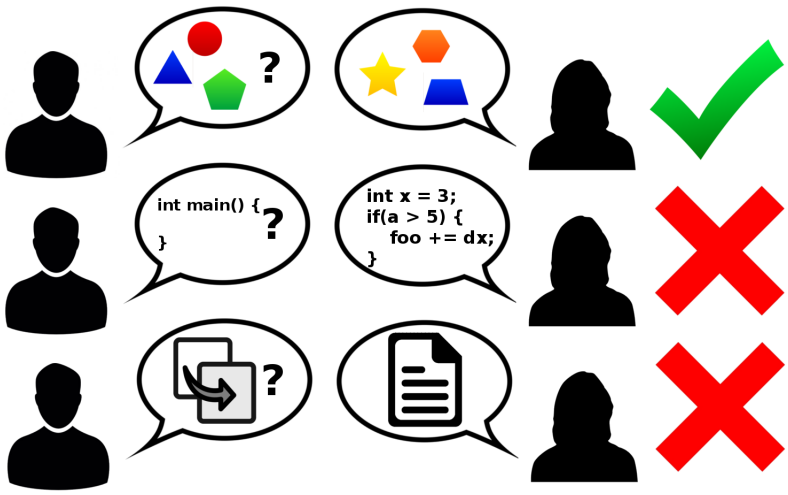

Figure 1: Collaboration rules, explained colorfully