Computer Science 2211b
Software Tools and Systems Programming
Winter 2024

Assignment 3 (7%)

Due: Monday February 26 at 11:55 PM

Objectives

In this assignment, you will gain experience with:

- Using control flow statements such as conditional statements and loops
- Writing simple functions
- Using arrays in C
- Following style conventions and good programming practices

PART 1 - Compression (25%)

In Parts 1 and 2, you'll write two simple programs to compress and decompress text.

Write a program `compress.c` that meets the requirements below:

1. Declare a character array of size 1024 to hold the user input. This should be declared in your `main` function and passed in to other functions. No global variables.

2. Write a function `read_input` with the following signature:

   ```
   int read_input(char buffer[1024])
   ```

   The function takes a character array passed from `main`. It then reads user input as follows:

   - Read the next character. **Hint**: look up the `getchar` function or use `scanf`

   - If the character is any whitespace character (space, tab, newline, etc.), then skip it. **Hint:** look up the `isspace` function

   - Otherwise, store the character in the next available element of the array

- Keep reading until `getchar` (or `scanf`) returns `EOF`, indicating the end of the input stream

The function should return the number of non-whitespace characters read into the buffer.

3. Write a function `compress` with the following signature:

```
void compress(char buffer[1024], int length)
```

The function should iterate over the buffer, writing compressed data to standard output (the screen) in the format shown below. Don't forget to handle cases where the user entered only 1 character (or no characters, in which case you would print nothing).

Note: Your program should print **NO** prompt. It should just wait for input.

For example, if the user entered the following input:

```
ccccc b c c ccc cc cs 2211
```

Your program would print the following output:

```
5 c
1 b
8 c
1 s
2 2
2 1
```

You may assume that user input will always be less than 1024 characters long.

When entering your input interactively, you will need to press **Enter** at the end of your input and then press **Ctrl+D** to input the `EOF` character.

Alternatively, a handy way to test your program is with `echo`. Here, we *pipe* the output of `echo` into our program:

```
$ echo "ccccc b c c ccc cc cs 2211" | ./compress
5 c
1 b
8 c
1 s
2 2
2 1
```

PART 2 - Decompression (10%)

Write a program `decompress.c` that meets the requirements below:

1. Read input in the form output by the `compress` program. **Hint**: use `scanf` to read the count and character on each line

2. Decompress the data, printing each character separated by spaces. You do not need to reproduce the original format of the data: just separate each character by spaces.

3. Keep reading lines until `scanf` returns `EOF`

Note: Your program should print **NO** prompt. It should just wait for input.

Once again, when entering input interactively, you will need to press **Enter** and then press **Ctrl+D** to input the EOF character.

If we input the output from the previous part, the `decompress` program should print the following:

```
c c c c c b c c c c c c c c s 2 2 1 1
```

Once again, for convenience in testing, you can pipe output from one command into your program. For instance, you can pipe the output of your `compress` program into your `decompress` program:

```
% echo "ccccc b c c ccc cc cs 2211" | ./compress | ./decompress
c c c c c b c c c c c c c c s 2 2 1 1
```

For Part 2, you are not required to implement any functions other than `main` (but are welcome to do so, if you wish).

PART 3 - Tic-Tac-Toe (65%)

Write a program `tictactoe.c` that implements a game of Tic-Tac-Toe and meets the requirements below:

1. Declare a multi-dimensional array for the game in your `main` function:

   ```
   char board[3][3];
   ```

   Here, we declare a 3x3 array to be used to store the current state of the game. Don't forget to initialize your board (e.g., with spaces)!

2. Implement a function to print the board with the following signature:

   ```
   void print_board(char board[3][3])
   ```

   This function takes the board and prints it out with row and column separators, along with row and column numbers as shown in the sample output below.

3. Implement a function to prompt the user for a move with the following signature:

   ```
   void prompt_move(char board[3][3])
   ```

   The user will be `X` (the computer will be `O`). The function should prompt the user to enter a move in the form `row col`. For instance, to put an `X` in the top-right corner of the board, the user would enter `0 2` (row 0, column 2).

   The function should print an error and re-prompt the user for a move if the user:

   - Enters an invalid number
   - Selects a position that is already taken

4. Implement a function to check for a win with the following signature:

   ```
   bool check_win(char board[3][3])
   ```

   The function should check the board to see if anyone has won by getting:

   - A sequence of three symbols in a row
   - A sequence of three symbols in a column
   - A sequence of three symbols in one of the two diagonals

If a win is found, it should print a message (e.g., `X wins!`) and return `true`. Otherwise, it should return `false`. You can find `bool`, `true`, and `false` in `stdbool.h`.

5. Implement a function to move for the computer with the following signature:

```
void computer_move(char board[3][3])
```

We will use very poor AI for this function by simply choosing a random position. You can use the `rand` function in `stdlib.h` to get a random number. For instance, to get a random number between 0 and 2, you would use:

```
int num = rand() % 3;
```

Your function should continue to select a random row and column until it finds a position that has not already been taken. Once it finds one, it should print the computer's move to the screen, as shown in the sample output below.

IMPORTANT: The random number generator must be *seeded* at the beginning of your program. Otherwise, it will return the same sequence every time you run the program and will therefore not be random. You should seed it **once** at the beginning of your `main` function with the following code:

```
#include <stdlib.h>
#include <time.h>

int main()
{
    // Use the current time as the seed for
    // the random number generator
    srand((unsigned int)time(NULL));
```

6. After each move (player or computer), you should print the board and then check for a win (or for a "cat's game" where no moves are left and no one has won)

7. Once a win occurs, the program should exit. Also, if all board positions have been taken with no win, the program should print `Cat's Game!` and exit

8. The functions described in this part are the minimum functions required, however you are welcome and encouraged to implement other custom functions to further modularize your code as you see fit

Sample Output:

```
$ ./tictactoe

    0   1   2
  -------------
0 |   |   |   |
  -------------
1 |   |   |   |
  -------------
2 |   |   |   |
  -------------

Enter row and column for your move: 1 1

    0   1   2
  -------------
0 |   |   |   |
  -------------
1 |   | X |   |
  -------------
2 |   |   |   |
  -------------

Computer moves to row 0, column 1

    0   1   2
  -------------
0 |   | O |   |
  -------------
1 |   | X |   |
  -------------
2 |   |   |   |
  -------------

Enter row and column for your move: 0 0

    0   1   2
  -------------
0 | X | O |   |
  -------------
1 |   | X |   |
  -------------
2 |   |   |   |
  -------------
```

```
Computer moves to row 2, column 1

    0   1   2
  -------------
0 | X | O |   |
  -------------
1 |   | X |   |
  -------------
2 |   | O |   |
  -------------

Enter row and column for your move: 2 2

    0   1   2
  -------------
0 | X | O |   |
  -------------
1 |   | X |   |
  -------------
2 |   | O | X |
  -------------

X wins!
```

Submitting Your Assignment

Your programs must follow the **CS 2211 Style Guide** introduced in Topic 5, including:

- Snake case for identifiers (variables and function names)
- 4 spaces per indentation level -- no tabs
- Clear comments throughout the program
- Commented variable declarations
- A file header comment
- Function header comments indicating the purpose of each function
- The use of constants rather than magic numbers

Hint: Your editor can be set to expand tabs to spaces and can be configured to use a specific indentation level. You can Google how to do this for your preferred editor.

After completing this assignment, your `asn3` directory should look as follows:

```
$ tree ~/courses/cs2211/asn3
[redacted]
 ── compress.c
├── decompress.c
├── tictactoe.c
├── accommodation.txt        (if you requested academic accommodation)

1 directory, 4 files
```

If not, go back and ensure that your directory matches this structure.

For full details on submission, see the **Assignment Submission Instructions** from the course web site. As a quick overview, submitting will involve the following commands:

```
cd ~/courses/cs2211/asn3
git add .
git commit -m "Submitting assignment 3"
git push
git tag asn3-submission
git push --tags
```

Check your repository at [redacted] to confirm that you have pushed your files successfully.

Then, check your Western email for an email confirmation from the system. If you don't receive one, check your junk mail folder. If you still don't receive an email within 10-15 minutes, follow the steps to resubmit your assignment.

This involves first deleting your `asn3-submission` tag:

```
git tag -d asn3-submission
git push origin :refs/tags/asn3-submission
```

Then, resubmit:

```
git tag asn3-submission
git push --tags
```