```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

#define MAX_INPUT 100

int main() {
    char input[MAX_INPUT];  // array of characters that stores the entire command,
but limits the amount of char to MAX_INPUT
    char *word[10];         // Array of pointers that points to each word in a
command
    int running = 1;

    while (running)
    {
        printf("> ");

        if (fgets(input, MAX_INPUT, stdin) == NULL) // stdin = user input from
keyboard
        {
            break;
        }

        input[strcspn(input, "\n")] = 0; // fgets function automatically adds "/n"
when stored, so this replaces it with \0

        // Splits the words in input
        int i = 0;
        char *token = strtok(input, " ");
        while (token != NULL && i < 9) // args has 10 slots, but last one is
reserved for NULL --> "i < 9" keeps loop in bounds and args can store up to 9 words
        {
            word[i++] = token;
            token = strtok(NULL, " ");
        }
        word[i] = NULL; // Necessary for execvp()

        if (word[0] == NULL) // Emprty input
        {
            continue;
        }

        // Handle built-in commands: "close", "help", and "subtract"
        if (strcmp(input, "close") == 0)
        {
            printf("Exiting shell...\n\n");
            running = 0;
            continue;
        }
        else if (strcmp(input, "help") == 0)
        {
            printf("\nBuilt-in commands:\n");
            printf("- close: Terminates the shell.\n");
            printf("- help: Displays this message.\n");
            printf("- subtract <num1> <num2>: Subtracts two numbers and prints the
result.\n\n");
            continue;
```

```
        }
        else if (strcmp(word[0], "subtract") == 0) // args[0] args[1] args[2] =
subtract value1 value2 --> if args[0] = subtract, then do command
        {
            if (word[1] != NULL && word[2] != NULL)
            {
                int num1 = atoi(word[1]);
                int num2 = atoi(word[2]);
                printf("Result: %d\n\n", num1 - num2);
            }
            else
            {
                printf("Usage: subtract <num1> <num2>\n\n");
            }
            continue;
        }

        // Fork a child process for simple commands from /usr/bin
        pid_t ch_pid = fork();

        switch (ch_pid)
        {
            case 0:
                // replaces current program with the one specified by word[0]
                execvp(word[0], word); // Execute the command from user input
                printf("Command not found: %s\n\n", word[0]); // Only prints if
exec fails
                exit(1);
                break;

            case -1:
                printf("Fork failed.\n\n");
                break;

            default:
                waitpid(ch_pid, NULL, 0); // waits for child process to complete,
then parent process prints prompt and waits for command
                break;
        }
    }

    return 0;
}


/*
// strcmp(s1,s2) = string comparison between two strings s1 and s2
//      - Equal: return 0
//      - Not Equal: returns - or + # based on size if s1 is < or > s2
//
// strcspn(string, reject) = searches for the first occurence of "reject" and
returns the index of where "reject" is at in the string
//
// atoi() = converts strings into intergers
//
// tokens - seperated words that make up an entire command
//
// strtok(string, delimiter) = string tokenization for splitting input
//      - delimiter: the phrase in the string that you want strtok() to check for
```

```
//      - once it finds the delimiter, it replaces it with \0, tokens the substring
that came before it, and returns a pointer to the token
//
// fgets(*str, n, FILE *stream) = fgets reads a line of text from a file or input
(char array where input is stored, max # of char, input source)
//
// execvp(char *file, char argv[]) - replaces the current program with a new one
stated by *file parameter
//      - *file is the new program you want to run and char argv[] is the array of
arguements you'll pass within it as the command-line array
//      - end of args MUST be NULL to define the end of arguments list
//      - similar to execl except...
//          -execl requires arguements to be passed individually and requires full
path of program
//              - full path = /usr/bin/ls
//              - indivudally passed arguments = execl(/usr/bin/ls, -l, -a,...)
//          - execvp passes arguments in array and SEARCHES for the program
//
// "continue;" is necessary after each if statemen because it skips the rest of the
loop, bypassing the forking and executing logic
//      -otherwise, it will proceed within case statement, execvp will search for
the built-in command you typed, fail to find it, and return the error message
//
*/
```