```
/* lab4.c
 * Christopher Brant
 * cbrant
 * ECE 2220, Fall 2016
 * MP4
 * Subject: ECE222-1,#4
 *
 * Purpose: A simple file editor that can handle files with characters
 *          that are not printable.
 *
 *          The editor can find and replace any byte in the file.  In
 *          addition it can find any string.
 *
 * Assumptions:
 *     input file is read in as bytes
 *
 * Command line argument
 *   name of file to read
 *
 * Bugs:
 *
 * Notes:
 *
 * See the ECE 222 programming guide
 *
 * Format with
 *     astyle --style=kr lab3.c
 *
 * Replace "kr" with: bsd, ansi, java, gnu, linux, vtk, or google.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXLINE 128

/*-------------------------------------------------------------*/
/* Here is a sketch for a start to searching for a byte
 *
 * Search for a matching byte starting after position.  Wrap around
 *    to beginning of memory until found or return to starting position
 *
 * input:
 *     byte is a string with the hex characters for the byte to find
 *     mem_start is starting address of the data
 *     mem_size is the number of bytes in the memory block
 * input/output:
 *     position is the location of the cursor.  If the byte is found
 *     then position is updated to the memory offset for the match
 *     If the byte is NOT found, then position is not changed
 * return value:
 *     true if byte found
 */
int find_next_byte_match(int *position, char *byte, char *mem_start, int mem_size)
{
    // This function changes two ascii nibbles into a single hex value
    int found = 0;
    int i;
    char byteval;

    for (i = 0; i < 2; i++)
    {
        if (byte[i] >= 'a' && byte [i] <= 'f')
            byte[i] -= 32;
    }

    if (byte[0] >= '0' && byte[0] <= '9')
        byteval = (byte[0] - '0') * 16;
    else if (byte[0] >= 'A' && byte[0] <= 'F')
        byteval = ((byte[0] - 'A') + 10) * 16;

    if (byte[1] >= '0' && byte[1] <= '9')
        byteval += byte[1] - '0';
    else if (byte[1] >= 'A' && byte[1] <= 'F')
        byteval += (byte[1] - 'A') + 10;

    i = *position + 1;

    while (byteval != mem_start[i] && i != *position)
    {
        if (i == mem_size)
            i = -1;
        i++;
    }

    if (byteval == mem_start[i] && i != *position && i < mem_size)
    {
        *position = i;
        found = 1;
    }

    return found;
}
/*-------------------------------------------------------------*/
/* Here is a sketch for a start to searching for a string
 *
 * Search for a matching string starting after position.  Wrap around
 *    to beginning of memory until found or return to starting position
 *
 *    Note: the string does not match if it overlaps both the characters
 *          at the end and the beginning of memory.
 *
 * input:
 *     str      : string to find.  It has already been processed to remove escapes
 *     wild_pos : -1 if no wildcard, or position in str with the wildcard
 *     mem_start: starting address of the data
 *     mem_size : the number of bytes in the memory block
 *
 * input/output:
 *     position is the location of the cursor.  If the string is found
 *     the position is updated to the memory offset for the first byte
 *     that matchs the string
 * return value:
 *     true if string found
 *
 */
int find_next_string(int *position, const char *str, int wild_pos,
        char *mem_start, int mem_size)
{
    int found = 0;
    int slen = strlen(str);
    int i = *position + 1;
    int j;
    int match_start;

    // If there is no wild card, search for a string
    if (wild_pos == -1)
    {
        while (i < mem_size && found == 0)
        {
            while (str[0] != mem_start[i] && i != *position)
            {
```

```
            if (i == mem_size)
                i = -1;
            i++;
        }

        // If the function finds no match and wraps back to its
        // original position, break from the loop
        if (i == *position)
        {
            match_start = i;
            i++;

            for (j = 1; j < slen; i++, j++)
            {
                if (str[j] != mem_start[i])
                    break;
            }

            if (j == slen)
            {
                found = 1;
                *position = match_start;
            }

            break;
        }

        match_start = i;
        i++;

        for (j = 1; j < slen; i++, j++)
        {
            if (str[j] != mem_start[i])
                break;
        }

        if (j == slen)
        {
            found = 1;
            *position = match_start;
        }
    }
}

else if (wild_pos == 0)
{
    while (i < mem_size && found == 0)
    {
        while (str[1] != mem_start[i] && i != *position)
        {
            if (i == mem_size)
                i = -1;
            i++;
        }

        if (i == *position)
        {
            match_start = i;
            i++;

            for (j = 2; j < slen; i++, j++)
            {
                if (str[j] != mem_start[i])
                    break;
            }

            if (j == slen)
            {
                found = 1;
                *position = match_start;
            }

            break;
        }

        match_start = i;
        i++;

        for (j = 2; j < slen; i++, j++)
        {
            if (str[j] != mem_start[i])
                break;
        }

        if (j == slen)
        {
            found = 1;
            *position = match_start - 1;
        }
    }
}

else if (wild_pos > 0)
{
    while (i < mem_size && found == 0)
    {
        while (str[0] != mem_start[i] && i != *position)
        {
            if (i == mem_size)
                i = -1;
            i++;
        }

        // If the function finds no match and wraps back to its
        // original position, break from the loop
        if (i == *position)
        {
            match_start = i;
            if (wild_pos != 1)
                i++;

            for (j = 1; j < slen; i++, j++)
            {
                if (j == wild_pos)
                {
                    i++;
                    j++;
                }

                if (str[j] != mem_start[i])
                    break;
            }

            if (j == slen)
            {
                found = 1;
                *position = match_start;
            }

            break;
        }

        match_start = i;
        i++;
```

```
            for (j = 1; j < slen; i++, j++)
            {
                if (j == wild_pos)
                {
                    i++;
                    j++;
                }

                if (str[j] != mem_start[i])
                    break;
            }

            if (j == slen)
            {
                found = 1;
                *position = match_start;
            }
        }
    }

    return found;
}
/*-------------------------------------------------------------*/
/* Here is a sketch for a start to replacing a string
 *
 * Search for a matching string starting after position.  Wrap around
 *    to beginning of memory until found or return to starting position
 *
 *    Note: the string does not match if it overlaps both the characters
 *          at the end and the beginning of memory.
 *
 * input:
 *     str      : string to find.  It has already been processed to remove escapes
 *     wild_pos : -1 if no wildcard, or position in str with the wildcard
 *     rstr     : string for replacement
 *     mem_start: starting address of the data
 *     mem_size : the number of bytes in the memory block
 *
 * input/output:
 *     position is the location of the cursor.  If the string is found
 *     the position is updated to the memory offset for the first byte
 *     that matchs the string
 * return value:
 *     true if string found
 */
int replace_next_string(int *position, char *str, int wild_pos, char *rstr,
        char *mem_start, int mem_size)
{
    int found = 0;
    int slen = strlen(str);
    int i = *position + 1;
    int j, k;
    int match_start = 0;

    // Exactly the same as find next string except for
    // the string is then replaced with rstr before the function ends
    if (wild_pos == -1)
    {
        while (i < mem_size && found == 0)
        {
            while (str[0] != mem_start[i] && i != *position)
            {
                if (i == mem_size)
                    i = -1;
                i++;
            }

            if (i == *position)
            {
                match_start = i;
                i++;

                for (j = 1; j < slen; i++, j++)
                {
                    if (str[j] != mem_start[i])
                        break;
                }

                if (j == slen)
                {
                    found = 1;
                    *position = match_start;

                    for (i = *position, k = 0; k < slen; i++, k++)
                        mem_start[i] = rstr[k];
                }

                break;
            }

            match_start = i;
            i++;

            for (j = 1; j < slen; i++, j++)
            {
                if (str[j] != mem_start[i])
                    break;
            }

            if (j == slen)
            {
                found = 1;
                *position = match_start;

                for (i = *position, k = 0; k < slen; i++, k++)
                    mem_start[i] = rstr[k];
            }
        }
    }
}

else if (wild_pos == 0)
{
    while (i < mem_size && found == 0)
    {
        while (str[1] != mem_start[i] && i != *position)
        {
            if (i == mem_size)
                i = -1;
            i++;
        }

        if (i == *position)
        {
            match_start = i;
            i++;

            for (j = 2; j < slen; i++, j++)
            {
                if (str[j] != mem_start[i])
                    break;
            }

            if (j == slen)
```

```
            {
                found = 1;
                *position = match_start;

                for (i = *position, k = 0; k < slen; i++, k++)
                    mem_start[i] = rstr[k];
            }

            break;
        }

        match_start = i;
        i++;

        for (j = 2; j < slen; i++, j++)
        {
            if (str[j] != mem_start[i])
                break;
        }

        if (j == slen)
        {
            found = 1;
            *position = match_start;

            for (i = *position, k = 0; k < slen; i++, k++)
                mem_start[i] = rstr[k];
        }
    }
}

else
{
    while (i < mem_size && found == 0)
    {
        j = 1;
        while (str[0] != mem_start[i] && i != *position)
        {
            if (i == mem_size)
                i = -1;
            i++;
        }

        if (i == *position)
        {
            match_start = i;
            if (wild_pos != 1)
                i++;

            for (j = 1; j < slen; i++, j++)
            {
                if (j == wild_pos)
                {
                    i++;
                    j++;
                }

                if (str[j] != mem_start[i])
                    break;
            }

            if (j == slen)
            {
                found = 1;
                *position = match_start;

                for (i = *position, k = 0; k < slen; i++, k++)
                    mem_start[i] = rstr[k];
            }

            break;
        }

        match_start = i;
        i++;

        for (j = 1; j < slen; i++, j++)
        {
            if (j == wild_pos)
            {
                i++;
                j++;
            }

            if (str[j] != mem_start[i])
                break;
        }

        if (j == slen)
        {
            found = 1;
            *position = match_start;

            for (i = *position, k = 0; k < slen; i++, k++)
                mem_start[i] = rstr[k];
        }
    }

    return found;
}
/*-------------------------------------------------------------*/
/* Here is a sketch for a start to search and replace byte
 *
 * Search for a matching byte starting after position.  Wrap around
 *    to beginning of memory until found or return to starting position
 *
 * input:
 *     byte_to_find is a string with the hex characters for the byte to find
 *     byte_to_replace is a string with the hex characters to replace
 *     mem_start is starting address of the data
 * input/output:
 *     position is the location of the cursor.  If the byte is replaced
 *     then position is updated to the memory offset for the match
 *     If the byte is NOT found, then position is not changed
 * return value:
 *     true if byte replaced
 */
int replace_next_byte(int *position, char *byte_to_find, char *byte_to_replace,
        char *mem_start, int mem_size)
{
    // Converts two different ascii bytes into hex
    // Finds the first one and replaces it with the second
    int found = 0;
    int i = 0;
    char byteval, replaceval;

    for (i = 0; i < 2; i++)
    {
        if (byte_to_find[i] >= 'a' && byte_to_find [i] <= 'f')
            byte_to_find[i] -= 32;
    }

    for (i = 0; i < 2; i++)
```

```
    {
        if (byte_to_replace[i] >= 'a' && byte_to_replace [i] <= 'f')
            byte_to_replace[i] -= 32;
    }

    if (byte_to_find[0] >= '0' && byte_to_find[0] <= '9')
        byteval = (byte_to_find[0] - '0') * 16;
    else if (byte_to_find[0] >= 'A' && byte_to_find[0] <= 'F')
        byteval = ((byte_to_find[0] - 'A') + 10) * 16;

    if (byte_to_find[1] >= '0' && byte_to_find[1] <= '9')
        byteval += byte_to_find[1] - '0';
    else if (byte_to_find[1] >= 'A' && byte_to_find[1] <= 'F')
        byteval += (byte_to_find[1] - 'A') + 10;

    if (byte_to_replace[0] >= '0' && byte_to_replace[0] <= '9')
        replaceval = (byte_to_replace[0] - '0') * 16;
    else if (byte_to_replace[0] >= 'A' && byte_to_replace[0] <= 'F')
        replaceval = ((byte_to_replace[0] - 'A') + 10) * 16;

    if (byte_to_replace[1] >= '0' && byte_to_replace[1] <= '9')
        replaceval += byte_to_replace[1] - '0';
    else if (byte_to_replace[1] >= 'A' && byte_to_replace[1] <= 'F')
        replaceval += (byte_to_replace[1] - 'A') + 10;

    i = *position + 1;

    while (byteval != mem_start[i] && i != *position)
    {
        if (i == mem_size)
            i = -1;
        i++;
    }

    if (byteval == mem_start[i] && i != *position)
    {
        mem_start[i] = replaceval;
        *position = i;
        found = 1;
    }

    return found;
}
/*-------------------------------------------------------------*/
/* Here is a sketch for a start to printing a 16-byte aligned
 * line of text.
 *
 * input:
 *     position is the cursor location
 *     slen is the length of the matching string (1 if just cursor position)
 *     mem_start is starting address of the data
 *     mem_size is the number of bytes in the memory block
 *
 * output:
 *     prints a 16-byte line of text that is aligned so that starting
 *     address is a multiple of 16
 *
 *     If slen is greater than one then continues to print 16-byte lines
 *     so that all characters in the string are displayed
 */
void print_line(int position, int slen, int wild_pos, char *mem_start, int mem_size
)
{
   int i, j, k, curprint = 0;
   int linepos = position / 16;
   int curpos = position % 16;
   char *line_start = mem_start + (16*linepos);
   char printed;
   int leftover = 0;
   int repeat = 1;
   int Line1 = 1;
   j = 0;

    while (slen == 1 || repeat == 1)
    {
        printf("         ");

        // Print the place value line first
        for (i = 0, k = 0; i < 16; i++, k++)
        {
            if (i == 10)
                k = 0;

            printf(" %d ", k);
        }

        // Print the line address/first byte in the line's place value
        printf("\n[%6d] ", linepos * 16);

        for (i = 0; i < 16; i++)
        {
            if (isprint(line_start[i]))
                printed = line_start[i];
            else
                printed = ' ';

            // Print the characters
            printf(" %c ", printed);
        }

        printf("\n          ");

        // Print the hex values below the characters
        for (i = 0; i < 16; i++)
            printf("%02hhX ", line_start[i]);


        if (Line1 && (curpos + slen) > 16)
            leftover = (curpos + slen) - 16;
        else
        {
            leftover -= 16;
        }

        // The rest of this function prints out the cursor and its
        // following indicators under each necessary place
        if (Line1)
        {
            printf("\n          ");

            i = 0;

            while (i != curpos)
            {
                printf("   ");
                i++;
            }

            if (wild_pos == 0)
            {
                printf("^*");
                j++;
            }
            else
```

```
            printf("^ ");

        for (i = curpos; j < slen; i++, j++)
        {
            if (slen == 1 || curpos + j > 16)
                break;
            else if (wild_pos != -1 && j + 1 == wild_pos && j + 2 == slen)
            {
                printf("-*|");
                break;
            }
            else if (wild_pos != -1 && j == wild_pos && wild_pos != 0 && i != 1
5)
            {
                printf("-*-");
                curprint = 1;
            }
            else if (wild_pos != -1 && j == wild_pos && wild_pos != 0)
            {
                printf("-*-");
                break;
            }
            else if (j + 1 == slen && j != wild_pos)
            {
                printf("--|");
                break;
            }
            else if (j + 2 == slen && wild_pos == -1 && i < 15)
            {
                printf("--|");
                break;
            }
            else if (i == 15 && wild_pos > 0 && curprint == 1)
            {
                printf("---");
                break;
            }
            else if (i == 15)
                break;
            else if (i < 15 && j + 1 != wild_pos)
                printf("---");
        }
    }

    else
    {
        if (wild_pos == 0)
            j--;

        printf("\n         -");
        for (i = 0; j < slen; j++, i++)
        {
            if (j + 1 == slen && j == wild_pos)
            {
                printf("*|");
                break;
            }
            else if (j + 1 == slen)
            {
                printf("-|");
                break;
            }
            else if (wild_pos != -1 && j == wild_pos)
                printf("*--");
            else if (i == 15)
            {
                printf("--");
                break;
            }
            else if (i < 15)
                printf("---");
        }
    }

    printf("\n\n");

    // The following determines if the string cannot be printed on one line
    // And after determining that, it then decrements the "leftover" as necessa
ry
    if (slen == 1)
    {
        repeat = 0;
        slen = 0;
    }

    if (leftover > 0)
    {
        repeat = 1;
        Line1 = 0;
        line_start += 16;
        linepos += 1;
        j++;
    }

    else
        repeat = 0;
  }
}

/*-------------------------------------------------------------*/
/* Process the search string to remove escapes and identify the location
 * of the first wildcard symbol.
 *
 * input: inputstr is the string from the command
 * output:
 *    searchstr: a copy of the input string with the escapes removed
 *
 * the return value is
 *    -1: no wildcard symbol found
 *     X: the position in the searchstr with the first wildcard
 *
 * This function does not change the input string, and assumes that the pesky '\n'
 *      has been removed.
 *
 * Note that unlike for the s command to search and replace, it is NOT
 * possible for the input string to be invalid.  So there cannot be
 * an invalid input string for searches.
 *
 * The only possible issue is if the '\' character is the LAST character
 * in the search string. We will take the convention that if the LAST
 * character is the '\' then it is NOT an escape, and should be a literal '\'.
 *
 * Example:
 *
 *   "\"  means search for '\' since the \ is the last character
 *   "\\" also means search for '\' since the first \ is an escape
 *
 *
 * This is not true for the s command (because that makes the dividing '/'
 * poorly defined).
 */
int process_search_string(const char *inputstr, char *searchstr)
{
    int wild_position = -1;  // -1 means no wildcard found
```

```
    int i = 0;
    int j = 0;

    // Processes the escape character out if necessary
    // Also copies input into output
    while (inputstr[i] != '\0')
    {
        if (inputstr[i] == '.' && wild_position == -1)
            wild_position = j;
        else if (inputstr[i] == 92 && inputstr[i+1] == '\0')
            searchstr[j] = inputstr[i];
        else if (inputstr[i] == 92 && inputstr[i+1] != '\0')
            i++;

        searchstr[j] = inputstr[i];

        i++;
        j++;
    }

    searchstr[j] = '\0';

    return wild_position;
}


/* Simple test to verify that the replacement string has correct form.
 *
 * Input: inputstr
 * Output:
 *    searchstr: the search string with all escape '\' symbols removed
 *    replacestr: the string used to replace the search string.
 *
 *    the return value:
 *      -2 if there is any error in the strings
 *      -1 if the strings are correct and there is no wildcard
 *       X for some integer X if the input is correct and the first wildcard
 *         is found at position X in the searchstr.
 *
 * This function does not change the input string, and assumes that the pesky '\n'
 *      has been removed.
 *
 * The string must start and end with a '/', and a '/' divides the input into
 * the searchstr and the replacestr.
 * .
 * The shortest pattern has the form /a/b/
 *
 * The pattern must have the form /string1/string2/
 * The string1 many contain one wildcard '.' symbol.  In addition, multiple
 *    escape '\' symbols may be included.
 *
 *    Process string1 to create the output searchstr.  Remove the escape symbols
 *    and save the location of the first wildcard sybmol, if found.
 *
 *    The length of searchstr must match the length of replacestr.
 *
 *    Note that the rule that the replacestr must have the same length as the
 *    searchstr (after escapes have been removed) means that there is no need
 *    for escapes '\' in the replacement string.  No wildcard symbols can be
 *    included in the replacement string.
 *
 *    Examples
 *      s /Clems.n/Clemson/ -- a wildcard matches any byte but replaces it with 'o'
 *      s /Cl.ms.n/Clemson/ -- The first '.' is wildcard but the second '.' is
 *                             a literal '.' and must be matched
 *      s /.ear./Here!/ -- The first '.' is a wildcard by the second is not.
 *      s /a\.b/a/b/  -- find the literal pattern "a.b" and change to "a/b". This
 *                       will not match a*b because the '.' is not a wildcard.
 *                       Note that "a/b" does not cause confusion in finding the
 *                       replacement string because the length of the replacement
 *                       string is known once "a\.b" is processed to "a.b"
 *      s /a\/b/a+b/ -- find the literal pattern "a/b" and replace with "a+b"
 *      s /a\\b/a/b/ -- find the literal pattern "a\b" and replace with "a/b"
 *
 */
int process_replace_string(const char *inputstr, char *searchstr, char *replacestr)
{
    int wild_position = -1;  // -1 means no wildcard found
    int i, j;
    int searchlength, replacelength;
    int inputlength = strlen(inputstr);

    // Determines if the input has the correct first and last characters
    if (inputstr[0] != '/' || inputstr[inputlength - 1] != '/')
        wild_position = -2;
    else
    {
        i = 1;
        j = 0;
        while (inputstr[i] != '/')      // This loop processes the search string
        {
            if (inputstr[i] == '.' && wild_position == -1)
                wild_position = j;
            else if (inputstr[i] == 92 && inputstr[i+1] == '/' && inputstr[i+2] !=
'/')
                i++;
            else if (inputstr[i] == 92 && inputstr[i+1] != '/')
                i++;

            searchstr[j] = inputstr[i];

            i++;
            j++;
        }

        searchstr[j] = '\0';
        i++;

        // This loop processes the replace string
        for (j = 0; inputstr[i+1] != '\0'; i++, j++)
            replacestr[j] = inputstr[i];

        replacestr[j] = '\0';

        // If the two strings are not of equal length, an error will be thrown
        searchlength = strlen(searchstr);
        replacelength = strlen(replacestr);

        if (searchlength != replacelength)
            wild_position = -2;
    }

    return wild_position;
}

/*-------------------------------------------------------------*/
/* The function to open the file, find the size of the file,
 * malloc memory to hold the contents of the file.
 *
 * There are two return values
 *   1.  the return value is a pointer to the starting
 *       memory location of the data
 *   2.  the size of the memory block in bytes is also
 *       returned in file_size
```

```
 */
char *store_file(char * filename, int * file_size)
{
    FILE *filein;

    if ((filein = fopen(filename, "r")) == NULL) {
        printf("Cannot Read from File \"%s\"\n", filename);
        exit (1);
    }

    int count = 0;
    char c;

    while (fscanf(filein, "%c", &c) == 1)
        count++;

    // Dynamically allocates memory for the data from the file
    char *filemem = (char *)malloc(count * sizeof(char));

    count = 0;
    rewind(filein);

    // Copies the data from the file to memory in the space just allocated
    while (fscanf(filein,"%c",&c) == 1)
    {
        filemem[count] = c;
        count++;
    }

    fclose(filein);
    *file_size = count;
    return filemem;
}

//
// Functions from here to end of this file should NOT be changed
//
/* print the menu of choices to the user
 *
 */
void print_menu(void)
{
    printf("Simple editor commands\n\n");
    printf("f FF    : Search for next matching byte (in hex) after current location
\n");
    printf("r AB CD : Same as search, then replace byte if found\n");
    printf("/Blue Ridge  : Search for next matching string after current location\n
");
    printf("s /Blue/Red / : Same as search, then replace string of same length\n");
    printf("G num : Goto byte at position 'num' from start of memory\n");
    printf("j     : Move forward 16 bytes from current location\n");
    printf("k     : Move backward 16 bytes from current location\n");
    printf("q     : Quit\n");
    printf("?     : Print this menu\n");
}

/*-------------------------------------------------------------*/
int main(int argc, char *argv[])
{
    char *filename;        // the input file name
    char *file_in_memory;  // starting address of memory block to store file
    int file_size;
    int fn_len;            // length of the input file name
    int found = 0;         // if search was successful
    int location = 0;      // current location in memory [0, file_size)
    int items;
    char line[MAXLINE];
    char command[MAXLINE];
    char inputcs[MAXLINE];
    char replacecs[MAXLINE];
    char searchstr[MAXLINE];
    char replacestr[MAXLINE];

    if (argc != 2) {
        printf("Usage: lab4 filename\n");
        exit(1);
    }

    // prepare filename
    fn_len = strlen(argv[1]);
    // remember the null
    filename = (char *) malloc((fn_len + 1) * sizeof(char));
    strcpy(filename, argv[1]);

    // open file and store in memory starting at pointer
    file_in_memory = store_file(filename, &file_size);

    print_menu();

    printf("> ");
    while (fgets(line, MAXLINE, stdin) != NULL) {
        printf("\n");
        items = sscanf(line, "%s%s%s", command, inputcs, replacecs);
        if (items == 2 && strcmp(command, "f") == 0) {
            if (strlen(inputcs) != 2 || !isxdigit(inputcs[0]) ||
                    !isxdigit(inputcs[1])) {
                printf("f Invalid byte: %s\n", inputcs);
            } else {
                found = find_next_byte_match(&location, inputcs, file_in_memory, fi
le_size);
                if (!found) {
                    printf("Did not find byte: %s\n", inputcs);
                } else {
                    print_line(location, 1, -1, file_in_memory, file_size);
                }
            }
        } else if (items == 3 && strcmp(command, "r") == 0) {
            if (strlen(inputcs) != 2 || !isxdigit(inputcs[0]) ||
                    !isxdigit(inputcs[1]) || strlen(replacecs) != 2 ||
                    !isxdigit(replacecs[0]) || !isxdigit(replacecs[1])) {
                printf("r Invalid bytes: %s %s\n", inputcs, replacecs);
            } else {
                found = replace_next_byte(&location, inputcs, replacecs, file_in_me
mory, file_size);
                if (!found) {
                    printf("Did not replace byte: %s\n", inputcs);
                } else {
                    print_line(location, 1, -1, file_in_memory, file_size);
                }
            }
        } else if (strncmp(line, "/", 1) == 0) {
            strcpy(inputcs, line + 1);
            // chomp the pesky \n
            if (inputcs[strlen(inputcs)-1] == '\n')
                inputcs[strlen(inputcs)-1] = '\0';
            int wild_pos = process_search_string(inputcs, searchstr);
            found = find_next_string(&location, searchstr, wild_pos,
                    file_in_memory, file_size);
            if (!found) {
                if (wild_pos == -1) {
                    printf("String not found: '%s' (no wildcard)\n", searchstr);
                } else {
                    printf("String not found: '%s' wildcard at %d\n",
                            searchstr, wild_pos);
```

```
                }
            } else {
                print_line(location, strlen(searchstr), wild_pos, file_in_memory, f
ile_size);
            }
        } else if (strncmp(line, "s /", 3) == 0) {
            strcpy(inputcs, line + 2);
            // chomp the pesky \n
            if (inputcs[strlen(inputcs)-1] == '\n')
                inputcs[strlen(inputcs)-1] = '\0';
            int wild_pos = process_replace_string(inputcs, searchstr, replacestr);
            if (wild_pos == -2) {
                printf("s Invalid input: %s\n", inputcs);
            } else {
                found = replace_next_string(&location, searchstr, wild_pos,
                        replacestr, file_in_memory, file_size);
                if (!found) {
                    if (wild_pos == -1) {
                        printf("String not replaced: '%s' (no wildcard)\n", searchs
tr);
                    } else {
                        printf("String not replaced: '%s' wildcard at %d\n",
                                searchstr, wild_pos);
                    }
                } else {
                    print_line(location, strlen(searchstr), wild_pos, file_in_memor
y, file_size);
                }
            }
        } else if (items == 2 && strcmp(command, "G") == 0) {
            int new_location = -1;
            new_location = atoi(inputcs);
            if (new_location < 0 || new_location >= file_size) {
                printf("Invalid goto: %s\n", inputcs);
            } else {
                location = new_location;
                print_line(location, 1, -1, file_in_memory, file_size);
            }
        } else if (items == 1 && strcmp(command, "j") == 0) {
            if (location + 16 >= file_size) {
                printf("Invalid move down: %d\n", location);
            } else {
                location += 16;
                print_line(location, 1, -1, file_in_memory, file_size);
            }
        } else if (items == 1 && strcmp(command, "k") == 0) {
            if (location - 16 < 0) {
                printf("Invalid move up: %d\n", location);
            } else {
                location -= 16;
                print_line(location, 1, -1, file_in_memory, file_size);
            }
        } else if (items == 1 && strcmp(command, "q") == 0) {
            break;
        } else if (items == 1 && strcmp(command, "?") == 0) {
            print_menu();
        } else {
            printf("# :%s", line);
        }
        printf("> ");
    }

    // for every malloc there must be a free
    free(file_in_memory);
    free(filename);

    printf("Goodbye\n");
    return EXIT_SUCCESS;
}
```