Day 9 Documentation

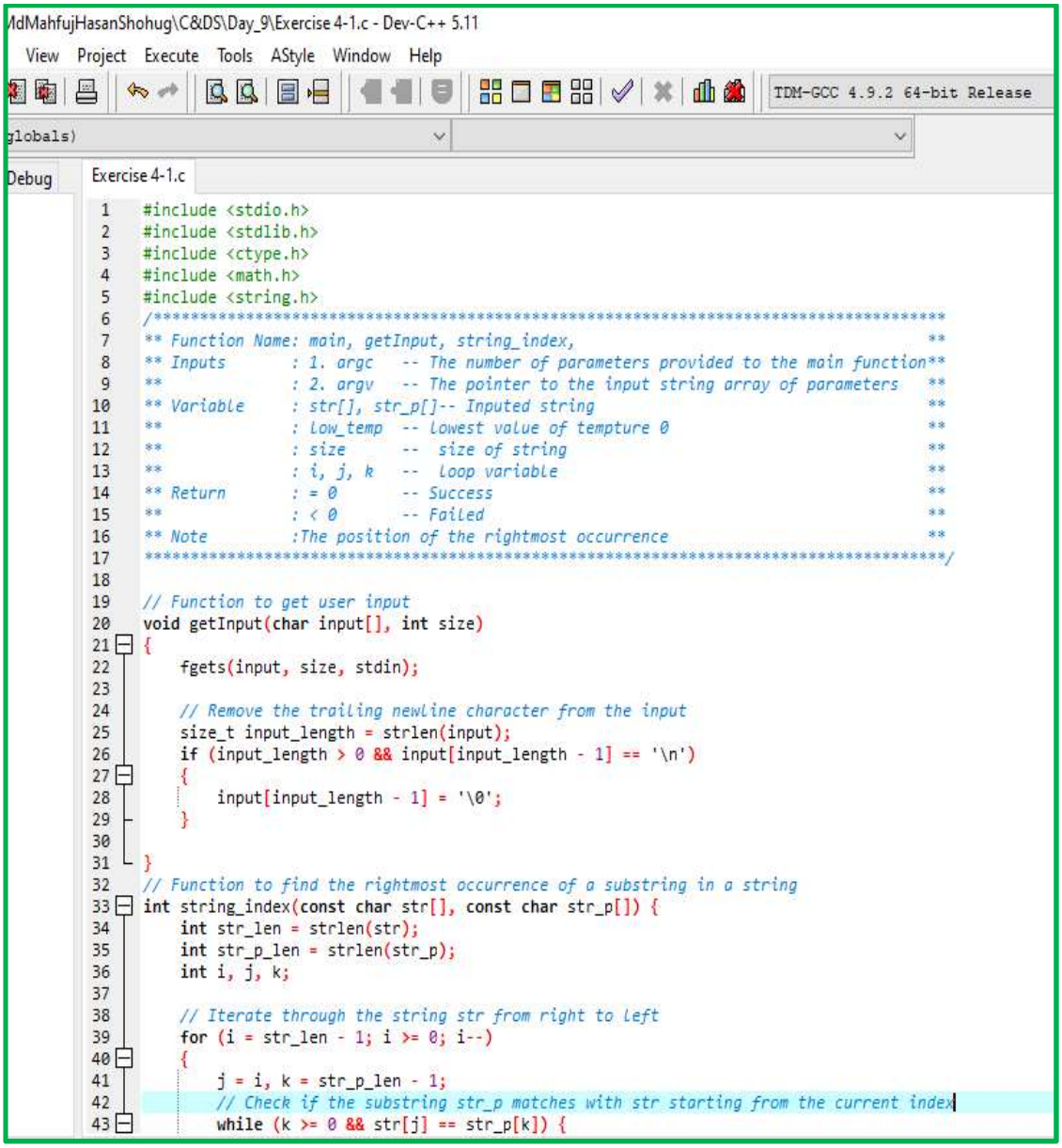
Md. Mahfuj Hasan Shohug

BDCOM0019

1. Exercise 4-1:

Problem: Write the function strindex(s,t) which returns the position of the rightmost Occurrence of t in s, or -1 if there is none.

Solution: The provided code is intended to locate a substring's rightmost occurrence within a supplied string. However, there are several possible issues and areas for improvement here is the source code:



```
            j--;
            k--;
        }

        // If the substring str_p matches completely, return the starting position of the occurrence
        if (k == -1)
            return j + str_p_len;
    }

    // If no match is found, return -1
    return -1;
}

//Main Function
int main(int argc, char *argv[])
{
    char str[100], str_p[100];

    printf("Enter First String: ");
    getInput(str, sizeof(str));

    printf("Enter Second String: ");
    getInput(str_p, sizeof(str_p));

    printf("The position of the rightmost occurrence of %s: %d\n", str_p, string_index(str, str_p));

    return 0;
}
```

Compile Log | Debug | Find Results | Close

```
Compiling single file...
--------
- Filename: D:\Repositery\MdMahfujHasanShohug\C&DS\Day_9\Exercise 4-1.c
- Compiler Name: TDM-GCC 4.9.2 64-bit Release

Processing C source file...
--------
- C Compiler: C:\Program Files (x86)\Dev-Cpp\MinGW64\bin\gcc.exe
- Command: gcc.exe "D:\Repositery\MdMahfujHasanShohug\C&DS\Day_9\Exercise 4-1.c" -o "D:\Repositery\MdMahf
```

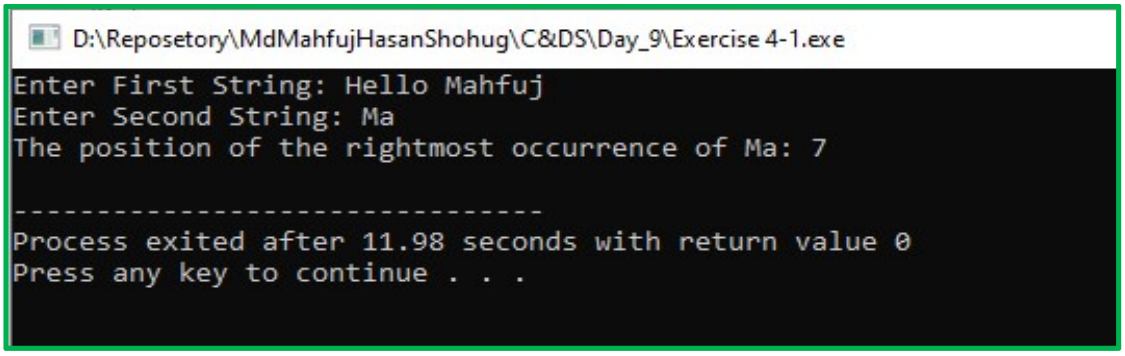
In the code: The getInput function is responsible for getting user input and removing the trailing newline character. The string_index function finds the rightmost occurrence of a substring within a string. The main function prompts the user for input, calls getInput to get the strings, and then calls string_index to find the rightmost occurrence.

Comments are added to explain the purpose and functionality of each part of the code.

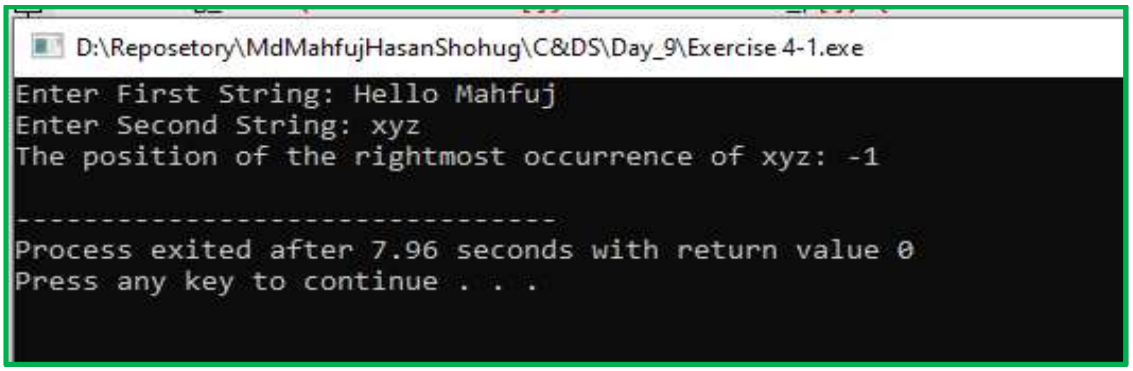
Use of fgets: The getInput method reads user input using fgets. However, the size of the input array is not properly given, which could result in buffer overflow issues. GetInput uses the sizeof operation, which only returns the size of the pointer rather than the array's size. To prevent buffer overflow flaws, it is advised to explicitly give the size to the getInput function.

The newline characters that follow: By examining the final character and replacing it with a null character if it is a newline, the function attempts to eliminate the trailing newline character from user input. The implementation, though, makes the erroneous assumption that there will always be a newline character. The function could yield unexpected results if user input doesn't terminate with a newline character. To prevent potential issues, it is advised to first verify whether a newline character is present before attempting to remove it.

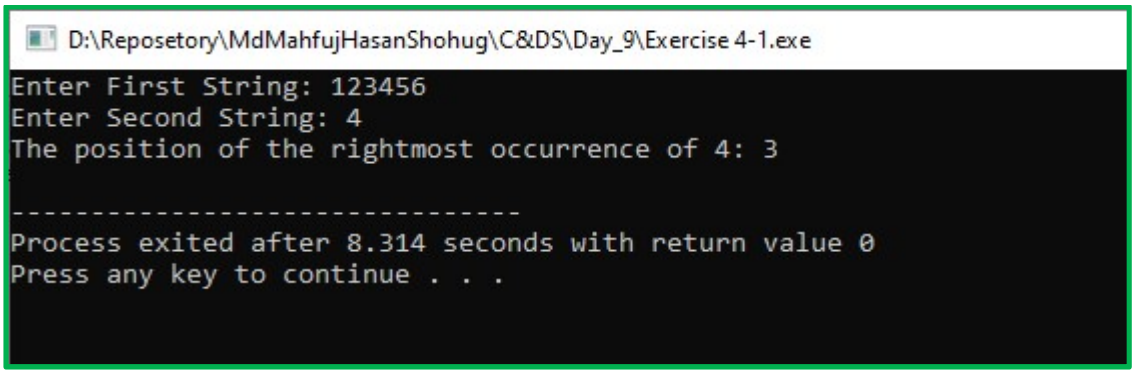
Insufficient Input Validation The provided input is not validated by the code to make sure that it does not exceed the size of the input arrays. If the user enters a string that is longer than the allotted size, this may result in buffer overflows.
No Check for Empty Strings: The code does not check for empty strings when performing the rightmost occurrence search. If either the main string or the substring is empty, the function may return incorrect results.
Here is some input output example:



Here “Ma” is found on the right most 7 position in first string.



Here the xyz is not found that’s why its return -1.

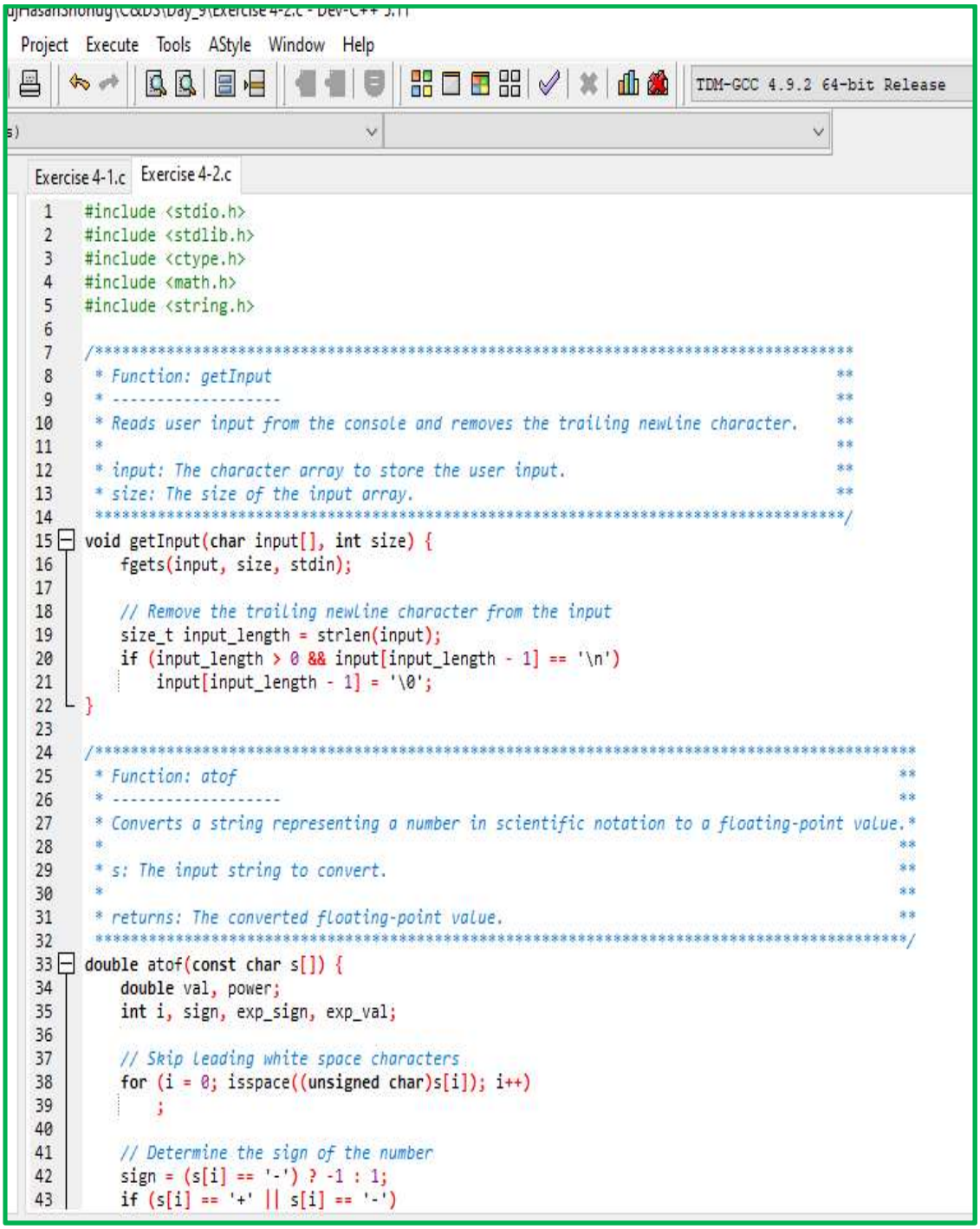


For better understand here 4 is found on the 3 number index.

2. Exercise 4-2:

Problem: Extend atof to handle scientific notation of the form 123.45e-6 where a floating-point number may be followed by e or E and an optionally signed exponent.

Solution: For solving this problem first here is the source code and after that I describe this in details:



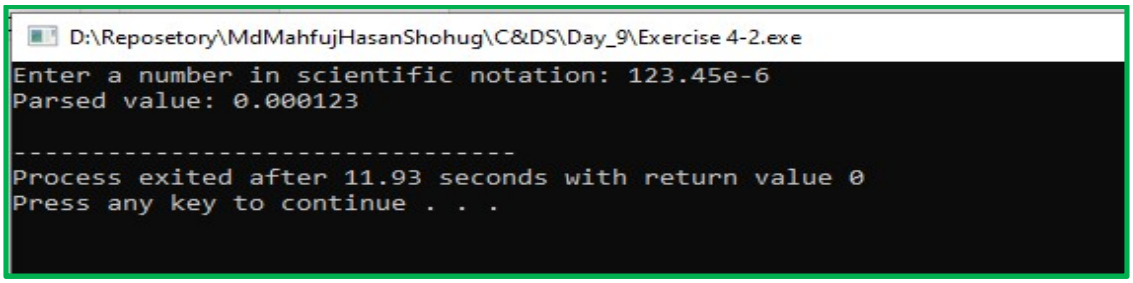

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <string.h>

/*****************************************************************************************
 * Function: getInput                                                                    **
 * ------------------                                                                    **
 * Reads user input from the console and removes the trailing newline character.         **
 *                                                                                       **
 * input: The character array to store the user input.                                   **
 * size: The size of the input array.                                                    **
 *****************************************************************************************/
void getInput(char input[], int size) {
    fgets(input, size, stdin);

    // Remove the trailing newline character from the input
    size_t input_length = strlen(input);
    if (input_length > 0 && input[input_length - 1] == '\n')
        input[input_length - 1] = '\0';
}

/*****************************************************************************************
 * Function: atof                                                                        **
 * ------------------                                                                    **
 * Converts a string representing a number in scientific notation to a floating-point value.*
 *                                                                                       **
 * s: The input string to convert.                                                       **
 *                                                                                       **
 * returns: The converted floating-point value.                                          **
 *****************************************************************************************/
double atof(const char s[]) {
    double val, power;
    int i, sign, exp_sign, exp_val;

    // Skip leading white space characters
    for (i = 0; isspace((unsigned char)s[i]); i++)
        ;

    // Determine the sign of the number
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
```

```
    if (s[i] == '+' || s[i] == '-')
        i++;

    // Process the integer part of the number
    for (val = 0.0; isdigit((unsigned char)s[i]); i++)
        val = 10.0 * val + (s[i] - '0');

    // Process the fraction part of the number
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit((unsigned char)s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10;
    }

    // Process the exponent part of the number
    if (s[i] == 'e' || s[i] == 'E') {
        i++;
        exp_sign = (s[i] == '-') ? -1 : 1;
        if (s[i] == '+' || s[i] == '-')
            i++;

        for (exp_val = 0; isdigit((unsigned char)s[i]); i++)
            exp_val = 10 * exp_val + (s[i] - '0');

        // Adjust the value based on the exponent
        if (exp_sign == 1)
            val *= pow(10, exp_val);
        else
            val /= pow(10, exp_val);
    }

    // Calculate the final value by dividing the integer part by the power of 10
    return sign * val / power;
}

/**********************************
 * Function: main                 **
 * -------------------            **
 * The entry point of the program.**
 **********************************/
int main() {
    char input[100];
    double result;

    printf("Enter a number in scientific notation: ");
    getInput(input, sizeof(input));
    result = atof(input);
    printf("Parsed value: %f\n", result);

    return 0;
}
```

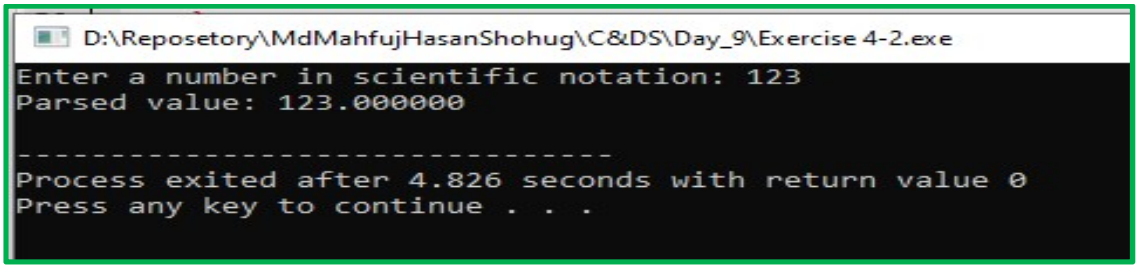
Compile Log | Debug | Find Results | Close

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: D:\Reposetory\MdMahfujHasanShohug\C&DS\Day_9\Exercise 4-2.exe
- Output Size: 152.4365234375 KiB
- Compilation Time: 0.19s
```

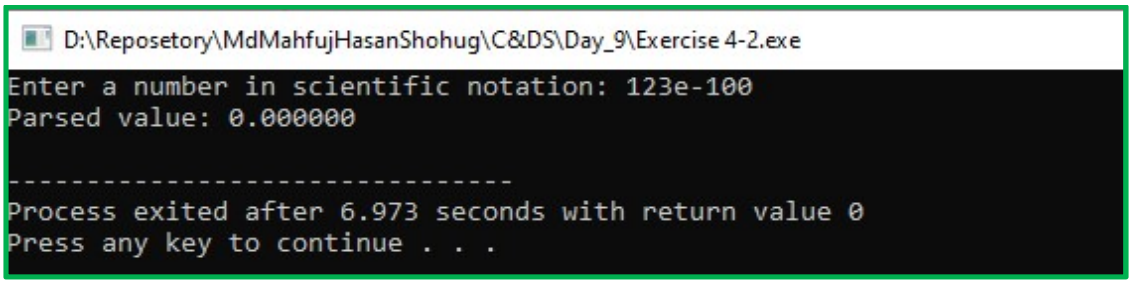
The atof function is used in this code to convert a user-inputted scientific notation number to a floating-point value. The functionality of the code is as follows:
getInput method is uses fgets to read user input from the console. Replaces the trailing newline character in the input with a null character to remove it.
atof operation accepts a scientific notation-based number as input in the form of a string, s. Extracts the integer, fractional, and exponent parts of the number from the string by processing it. Conducts the necessary calculations to produce the final floating-point value, converting the extracted components to their corresponding numeric values. Gives the result of the calculation.
The code contains the necessary header files, handles potential issues such as leading/trailing white space, handles both positive and negative numbers, and handles the index part of scientific notation. It is worth noting that this code accepts valid input from the user. Error handling and input validation are minimal, and it does not account for all possible edge cases. Thus, it is recommended to improve the code by adding proper error checking and validation mechanisms to handle unexpected input situations.
Outputs:



Here 123.45e-6 is get outputs correctly.



Here 123 has no exponential that's why result only return 123.

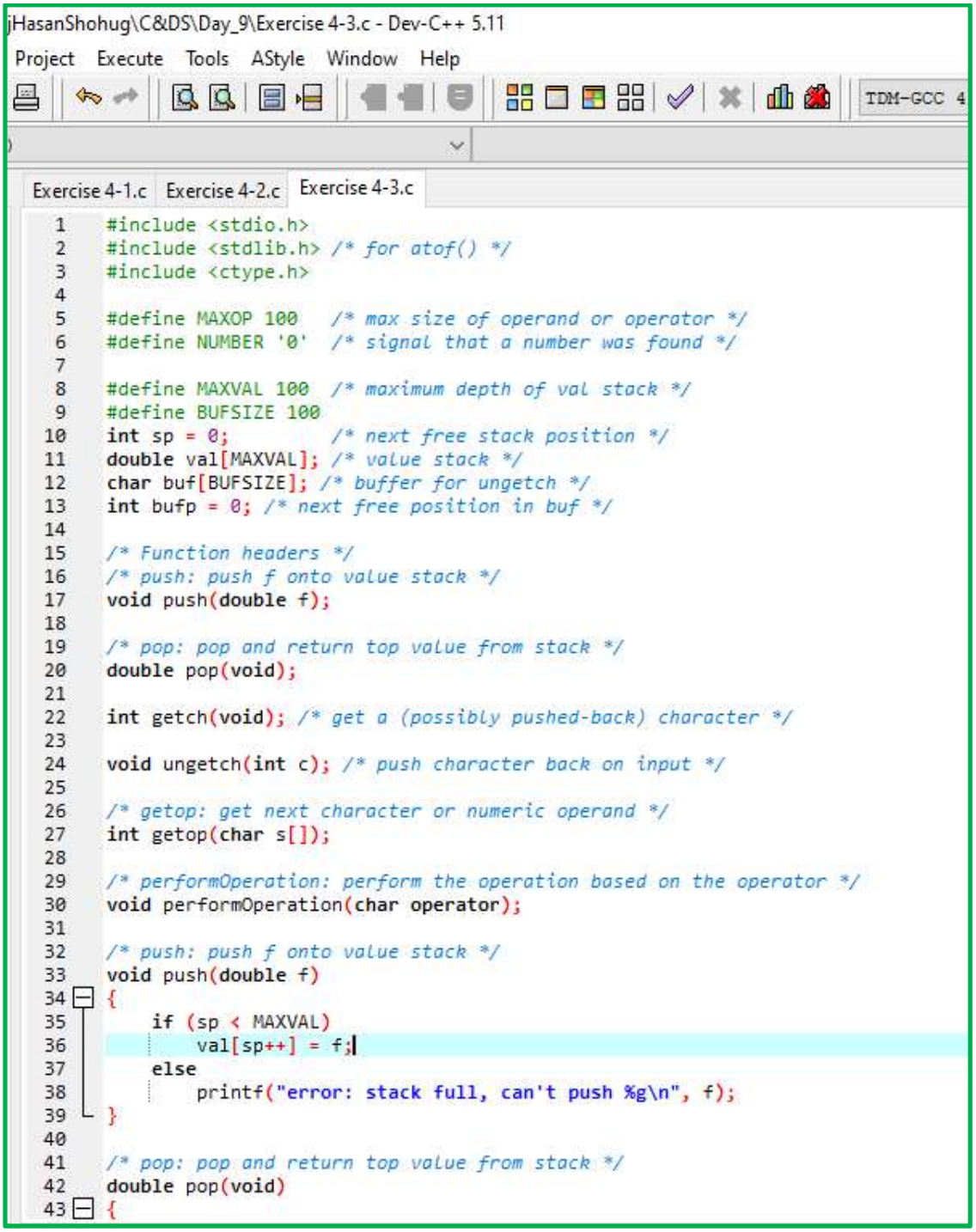


Here if this is cross the limit if I set the .100f on print function then It will show.

3. Exercise 4-3:

Problem: Given the basic framework, it's straightforward to extend the calculator. Add the modulus (%) operator and provisions for negative numbers.

Solution: Source code file:



```
43  {
44      if (sp > 0)
45          return val[--sp];
46      else
47      {
48          printf("error: stack empty\n");
49          return 0.0;
50      }
51  }
52
53  /* getch: get a (possibly pushed-back) character */
54  int getch(void)
55  {
56      return (bufp > 0) ? buf[--bufp] : getchar();
57  }
58
59  /* ungetch: push character back on input */
60  void ungetch(int c)
61  {
62      if (bufp >= BUFSIZE)
63          printf("ungetch: too many characters\n");
64      else
65          buf[bufp++] = c;
66  }
67
68  /* getop: get next character or numeric operand */
69  int getop(char s[])
70  {
71      int i, c;
72
73      while ((s[0] = c = getch()) == ' ' || c == '\t')
74          ;
75
76      s[1] = '\0';
77
78      if (!isdigit(c) && c != '.' && c != '-')
79          return c; /* not a number */
80
81      if (c == '-')
82      {
83          if (isdigit(s[1] = c = getch()))
84          {
85              i = 1;
```

```
85              i = 1;
86          }
87          else
88          {
89              ungetch(c);
90              return '-';
91          }
92      }
93      else
94      {
95          i = 0;
96      }
97
98      if (isdigit(c)) /* collect integer part */
99      {
100         while (isdigit(s[++i] = c = getch()))
101             ;
102     }
103
104     if (c == '.') /* collect fraction part */
105     {
106         while (isdigit(s[++i] = c = getch()))
107             ;
108     }
109
110     s[i] = '\0';
111
112     if (c != EOF)
113         ungetch(c);
114
115     return NUMBER;
116 }
117
118 /* performOperation: perform the operation based on the operator */
119 void performOperation(char operator)
120 {
121     switch (operator)
122     {
123         case '+':
124             push(pop() + pop());
125             break;
126         case '*':
127             push(pop() * pop());
```

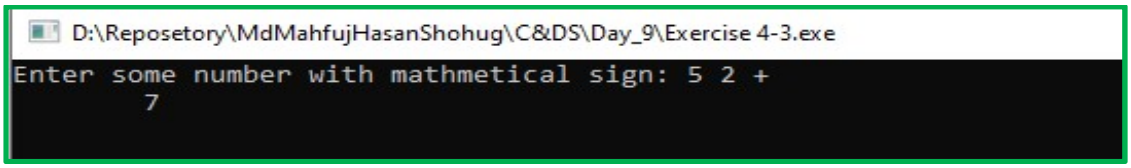
```
127                 push(pop() * pop());
128                 break;
129             case '-':
130                 {
131                     double op2 = pop();
132                     push(pop() - op2);
133                 }
134                 break;
135             case '/':
136                 {
137                     double op2 = pop();
138                     if (op2 != 0.0)
139                         push(pop() / op2);
140                     else
141                         printf("error: zero divisor\n");
142                 }
143                 break;
144             case '%':
145                 {
146                     double op2 = pop();
147                     if (op2 != 0.0)
148                         push((int)pop() % (int)op2);
149                     else
150                         printf("error: zero divisor\n");
151                 }
152                 break;
153             case '\n':
154                 printf("\t%.8g\n", pop());
155                 break;
156             default:
157                 printf("error: unknown command %c\n", operator);
158                 break;
159         }
160     }
161
162     /* Main function */
163     int main()
164     {
165         int type;
166         char s[MAXOP];
167         printf("Enter some number with mathmetical sign: ");
168         while ((type = getop(s)) != EOF)
169         {
```

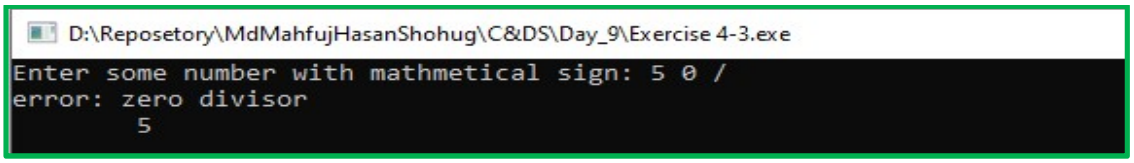
```
168         while ((type = getop(s)) != EOF)
169         {
170             if (type == NUMBER)
171                 push(atof(s));
172             else
173                 performOperation(s[0]);
174         }
175
176         return 0;
177     }
178
```

ces | Compile Log | Debug | Find Results | Close

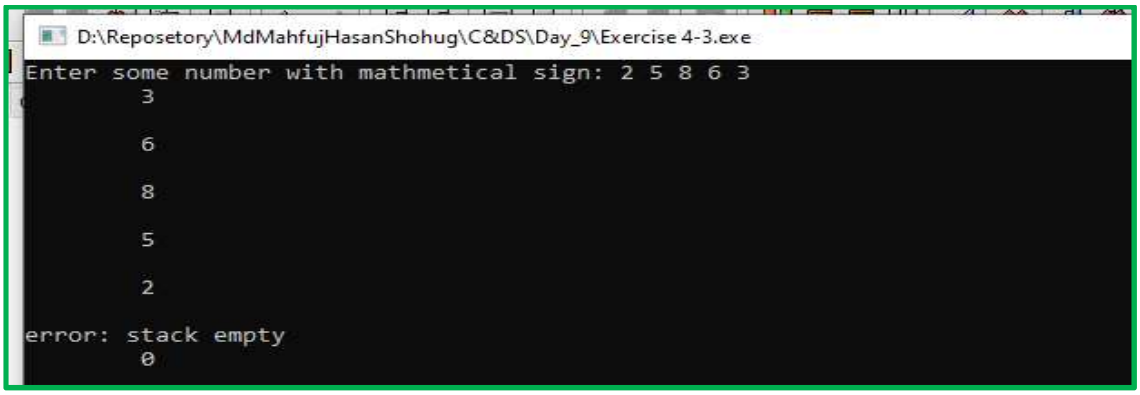
```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: D:\Reposetory\MdMahfujHasanShohug\C&DS\Day_9\Exercise 4-3.exe
- Output Size: 131.1337890625 KiB
- Compilation Time: 0.19s
```

This program is a simple Reverse Polish Notation (RPN) calculator. It uses a stack to perform arithmetic operations. Here is a brief description of the code:
The push() function pushes a value onto the stack.
The pop() function pops the top value from the stack and returns it.
The getch() function receives a character from input, possibly from a buffer.
The ungetch() function pushes one character back into the input.
The getop() function gets the next character or numeric operand. It handles numbers, negative numbers and fractions.
The PerformOperation() function performs arithmetic operations based on the operator.
The main() function is the entry point of the program. It reads input, pushes numbers onto the stack, and performs operations based on the input.
To use the program, you can enter mathematical expressions in reverse Polish notation (eg, 5 + 2 for 5 2 +). Press enter to get the result. The program supports addition (+), multiplication (*), subtraction (-), division (/), and modulo (%). Here is some outputs on this program:

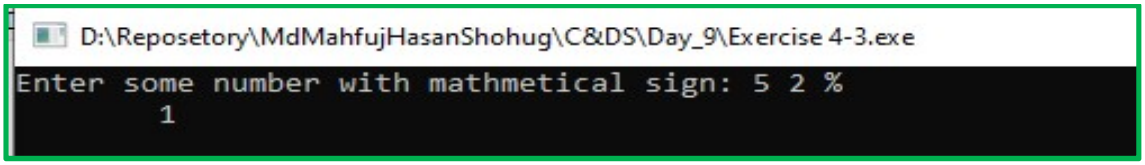




Cannot divided by 0 of any value.



For clicking the enter button '\n' the stack is automatically will be pop.



When using modulus the output is 1.

4. Exercise 4-4:

Problem: Add the commands to print the top elements of the stack without popping, to duplicate it, and to swap the top two elements. Add a command to clear the stack.

Solution: here is the details source code for this program:

```
#include <stdio.h>
#include <stdlib.h> /* for atof() */
#include <ctype.h>

#define MAXOP 100   /* max size of operand or operator */
#define NUMBER '0'  /* signal that a number was found */

#define MAXVAL 100  /* maximum depth of val stack */
/* Pushes a floating-point number onto the stack ************************************************
void push(double f);

/* Pops and returns the top value from the stack
double pop(void);

/* Prints the top element of the stack
void printTop(void);

/* Duplicates the top element of the stack
void duplicateTop(void);

/* Swaps the top two elements of the stack
void swapTopTwo(void);

/* Clears the stack
void clearStack(void);

/* Retrieves a character from input
int getch(void);

/* Pushes a character back onto the input
void ungetch(int c);

/* Gets the next character or numeric operand
 * s: character array to store the operand or operator
 * Returns the type of the token: NUMBER for a number, or the character itself for an operator
int getop(char s[]);

/* Performs the operation based on the operator
 * operator: the operator to perform the operation for
void performOperation(char operator);
*************************************************************************************************
****/

int sp = 0;       /* next free stack position */
double val[MAXVAL]; /* value stack */

/* Function prototypes */
int getch(void);
void ungetch(int c);
int getop(char s[]);
void performOperation(char operator);

/* push: push f onto value stack */
void push(double f)
{
   if (sp < MAXVAL)
      val[sp++] = f;
   else
      printf("error: stack full, can't push %g\n", f);
```

```
}

/* pop: pop and return top value from stack */
double pop(void)
{
  if (sp > 0)
    return val[--sp];
  else
  {
    printf("error: stack empty\n");
    return 0.0;
  }
}

/* Print the top element of the stack */
void printTop(void)
{
  if (sp > 0)
    printf("Top of stack: %.8g\n", val[sp - 1]);
  else
    printf("Stack is empty\n");
}

/* Duplicate the top element of the stack */
void duplicateTop(void)
{
  if (sp > 0)
  {
    double top = val[sp - 1];
    push(top);
  }
  else
    printf("Stack is empty\n");
}

/* Swap the top two elements of the stack */
void swapTopTwo(void)
{
  if (sp >= 2)
  {
    double top = pop();
    double second = pop();
    push(top);
    push(second);
  }
  else
    printf("Stack has less than two elements\n");
}

/* Clear the stack */
void clearStack(void)
{
  sp = 0;
}

#define BUFSIZE 100
char buf[BUFSIZE]; /* buffer for ungetch */
int bufp = 0; /* next free position in buf */

int getch(void) /* get a (possibly pushed-back) character */
{
  return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c) /* push character back on input */
{
```

```
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}

/* getop: get next character or numeric operand */
int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;

    s[1] = '\0';

    if (!isdigit(c) && c != '.' && c != '-')
        return c; /* not a number */

    if (c == '-')
    {
        if (isdigit(s[1] = c = getch()))
        {
            i = 1;
        }
        else
        {
            ungetch(c);
            return '-';
        }
    }
    else
    {
        i = 0;
    }

    if (isdigit(c)) /* collect integer part */
    {
        while (isdigit(s[++i] = c = getch()))
            ;
    }

    if (c == '.') /* collect fraction part */
    {
        while (isdigit(s[++i] = c = getch()))
            ;
    }

    s[i] = '\0';

    if (c != EOF)
        ungetch(c);

    return NUMBER;
}

/* performOperation: perform the operation based on the operator */
void performOperation(char operator)
{
    switch (operator)
    {
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
```

```
                    break;
                case '-':
                    {
                        double op2 = pop();
                        push(pop() - op2);
                    }
                    break;
                case '/':
                    {
                        double op2 = pop();
                        if (op2 != 0.0)
                            push(pop() / op2);
                        else
                            printf("error: zero divisor\n");
                    }
                    break;
                case '%':
                    {
                        double op2 = pop();
                        if (op2 != 0.0)
                            push((int)pop() % (int)op2);
                        else
                            printf("error: zero divisor\n");
                    }
                    break;
                case 'p':
                    printTop();
                    break;
                case 'd':
                    duplicateTop();
                    break;
                case 'w':
                    swapTopTwo();
                    break;
                case 'c':
                    clearStack();
                    break;
                case '\n':
                    printf("\t%.8g\n", pop());
                    break;
                default:
                    printf("error: unknown command %c\n", operator);
                    break;
            }
        }

        /* Main function */
        int main()
        {
            int type;
            char s[MAXOP];
            printf("Input 'p' for print top \n");
            printf("Input 'd' for duplicate \n");
            printf("Input 'w' for swap \n");
            printf("Input 'c' for clear all \n");
            while ((type = getop(s)) != EOF)
            {
                if (type == NUMBER)
                    push(atof(s));
                else
                    performOperation(s[0]);
            }

            return 0;
}
```

Here is a brief description of the code: The code includes standard library headers such as stdio.h, stdlib.h, and ctype.h. Two constants are defined: MAXOP which represents the maximum size of the operand or operator and NUMBER which is a signal indicating that a number has been found. The maximum depth of the value stack is defined as MAXVAL, and sp is a variable that keeps track of the next free stack position. val is an array used as a value stack to store numeric operands. The code defines several helper functions:

push() is used to push a value onto the value stack.

pop() is used to pop and return the top value from the value stack.

printTop() prints the top element of the stack.

duplicateTop() Duplicates the top element of the stack.

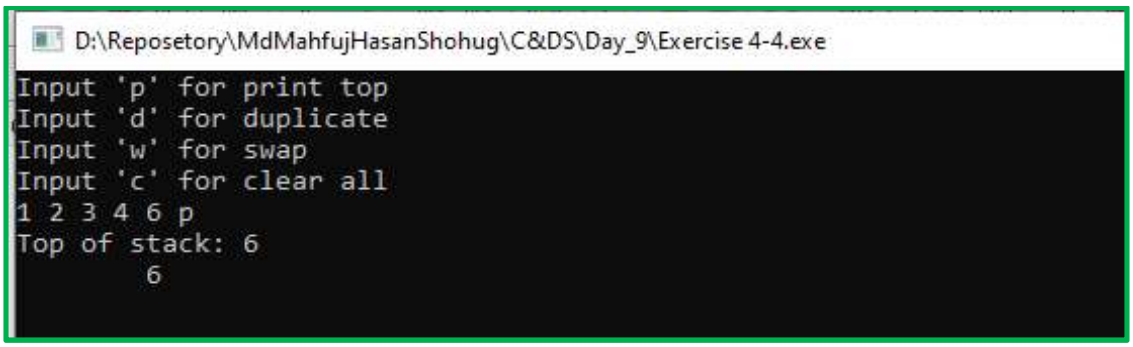
swapTopTwo() swaps the top two elements of the stack.

clearStack() clears the entire stack.

The code defines BUFSIZE as the size of the buffer for character input using the getch() and ungetch() functions. buf is an array used as a buffer to hold characters and bufp keeps track of the next free position in the buffer. The getch() and ungetch() functions are used to read characters from input, including the ability to return characters to the input stream. The getop() function is responsible for tokenizing the input and returning the next character or numeric operand. It handles numbers including integers and fractions as well as negative numbers. The PerformOperation() function performs operations based on the operator. It supports basic arithmetic operations such as addition, subtraction, multiplication, division and modulo. It also includes additional operations such as printing the top element, duplicating the top element, swapping the top two elements, and clearing the stack. Additional operations added to the code provide more flexibility and functionality for manipulating the stack during the calculation process.

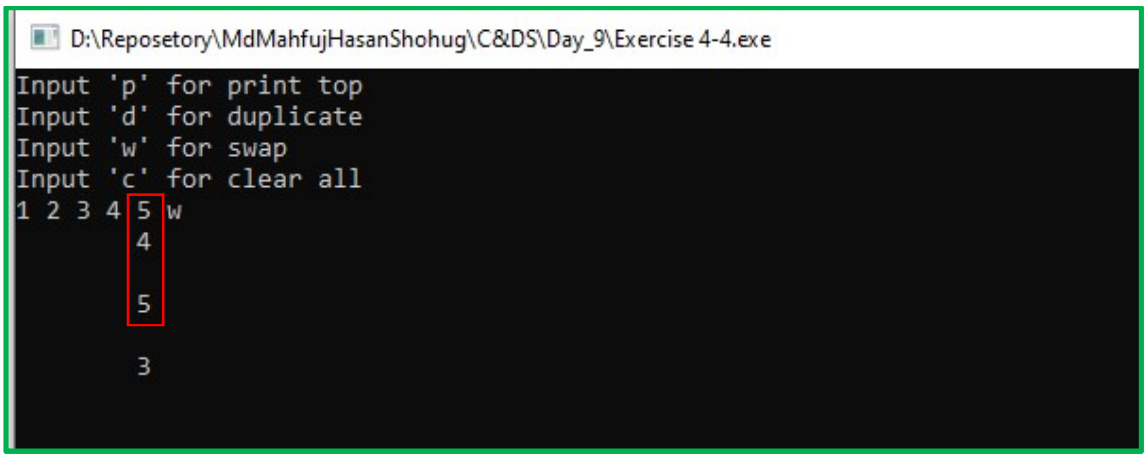
For the input and outputs:

Input 'p' for print top

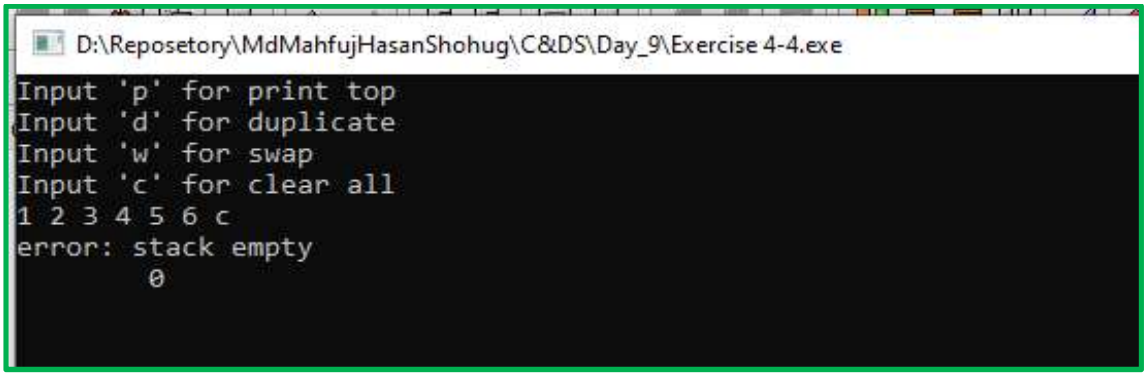


Input 'd' for duplicate

Input 'w' for swap



Input 'c' for clear all



For c clear the full stack and show empty.

5. Exercise 4-5:

Problem: Add access to library functions like sin, exp, and pow. See <math.h> in Appendix B, Section 4.

Solution: This is fully same as the previous problem just add extra case on this problem to calling the <math.h> library build in function:

Source code:

```
#include <stdio.h>
#include <stdlib.h> /* for atof() */
#include <ctype.h>
#include <math.h>

#define MAXOP 100   /* max size of operand or operator */
#define NUMBER '0'  /* signal that a number was found */

#define MAXVAL 100  /* maximum depth of val stack */
/* Pushes a floating-point number onto the stack ************************************************
void push(double f);

/* Pops and returns the top value from the stack
double pop(void);

/* Prints the top element of the stack
void printTop(void);

/* Duplicates the top element of the stack
void duplicateTop(void);

/* Swaps the top two elements of the stack
void swapTopTwo(void);

/* Clears the stack
void clearStack(void);

/* Retrieves a character from input
int getch(void);

/* Pushes a character back onto the input
void ungetch(int c);

/* Gets the next character or numeric operand
 * s: character array to store the operand or operator
 * Returns the type of the token: NUMBER for a number, or the character itself for an operator
int getop(char s[]);

/* Performs the operation based on the operator
 * operator: the operator to perform the operation for
void performOperation(char operator);
*************************************************************************************************
****/

int sp = 0;       /* next free stack position */
double val[MAXVAL]; /* value stack */

/* Function prototypes */
int getch(void);
void ungetch(int c);
int getop(char s[]);
void performOperation(char operator);

/* push: push f onto value stack */
void push(double f)
{
```

```
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n", f);
}

/* pop: pop and return top value from stack */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else
    {
        printf("error: stack empty\n");
        return 0.0;
    }
}

/* Print the top element of the stack */
void printTop(void)
{
    if (sp > 0)
        printf("Top of stack: %.8g\n", val[sp - 1]);
    else
        printf("Stack is empty\n");
}

/* Duplicate the top element of the stack */
void duplicateTop(void)
{
    if (sp > 0)
    {
        double top = val[sp - 1];
        push(top);
    }
    else
        printf("Stack is empty\n");
}

/* Swap the top two elements of the stack */
void swapTopTwo(void)
{
    if (sp >= 2)
    {
        double top = pop();
        double second = pop();
        push(top);
        push(second);
    }
    else
        printf("Stack has less than two elements\n");
}

/* Clear the stack */
void clearStack(void)
{
    sp = 0;
}

#define BUFSIZE 100
char buf[BUFSIZE]; /* buffer for ungetch */
int bufp = 0; /* next free position in buf */

int getch(void) /* get a (possibly pushed-back) character */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
```

```
}

void ungetch(int c) /* push character back on input */
{
   if (bufp >= BUFSIZE)
      printf("ungetch: too many characters\n");
   else
      buf[bufp++] = c;
}

/* getop: get next character or numeric operand */
int getop(char s[])
{
   int i, c;

   while ((s[0] = c = getch()) == ' ' || c == '\t')
      ;

   s[1] = '\0';

   if (!isdigit(c) && c != '.' && c != '-')
      return c; /* not a number */

   if (c == '-')
   {
      if (isdigit(s[1] = c = getch()))
      {
         i = 1;
      }
      else
      {
         ungetch(c);
         return '-';
      }
   }
   else
   {
      i = 0;
   }

   if (isdigit(c)) /* collect integer part */
   {
      while (isdigit(s[++i] = c = getch()))
         ;
   }

   if (c == '.') /* collect fraction part */
   {
      while (isdigit(s[++i] = c = getch()))
         ;
   }

   s[i] = '\0';

   if (c != EOF)
      ungetch(c);

   return NUMBER;
}

/* performOperation: perform the operation based on the operator */
void performOperation(char operator)
{
   switch (operator)
   {
      case '+':
```

```
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
                {
                    double op2 = pop();
                    push(pop() - op2);
                }
                break;
            case '/':
                {
                    double op2 = pop();
                    if (op2 != 0.0)
                        push(pop() / op2);
                    else
                        printf("error: zero divisor\n");
                }
                break;
            case '%':
                {
                    double op2 = pop();
                    if (op2 != 0.0)
                        push((int)pop() % (int)op2);
                    else
                        printf("error: zero divisor\n");
                }
                break;
            case 's':   /* sin function */
                push(sin(pop()));
                break;
            case 'e':   /* exp function */
                push(exp(pop()));
                break;
            case 'p':   /* pow function */
                {
                    double op2 = pop();
                    push(pow(pop(), op2));
                }
                break;
            case 'l':
                push(log(pop()));
                break;
            case 'f':
                push(floor(pop()));
                break;
            case 't':
                printTop();
                break;
            case 'd':
                duplicateTop();
                break;
            case 'w':
                swapTopTwo();
                break;
            case 'c':
                clearStack();
                break;
            case '\n':
                printf("\t%.8g\n", pop());
                break;
            default:
                printf("error: unknown command %c\n", operator);
                break;
        }
```

```
}

/* Main function */
int main()
{
   int type;
   char s[MAXOP];
   printf("Input 's' for sin(top) \n");
   printf("Input 'p' for pow(top) \n");
   printf("Input 'l' for log(top) \n");
   printf("Input 'f' for floor(top) \n");
   printf("Input 't' for print top \n");
   printf("Input 'd' for duplicate \n");
   printf("Input 'w' for swap \n");
   printf("Input 'c' for clear all \n");
   while ((type = getop(s)) != EOF)
   {
      if (type == NUMBER)
         push(atof(s));
      else
         performOperation(s[0]);
   }

   return 0;
}
```

Here is also code includes standard library headers such as stdio.h, stdlib.h, and ctype.h. and math.h Two constants are defined: MAXOP which represents the maximum size of the operand or operator and NUMBER which is a signal indicating that a number has been found. The maximum depth of the value stack is defined as MAXVAL, and sp is a variable that keeps track of the next free stack position. val is an array used as a value stack to store numeric operands. The code defines several helper functions:

push() is used to push a value onto the value stack.

pop() is used to pop and return the top value from the value stack.

printTop() prints the top element of the stack.

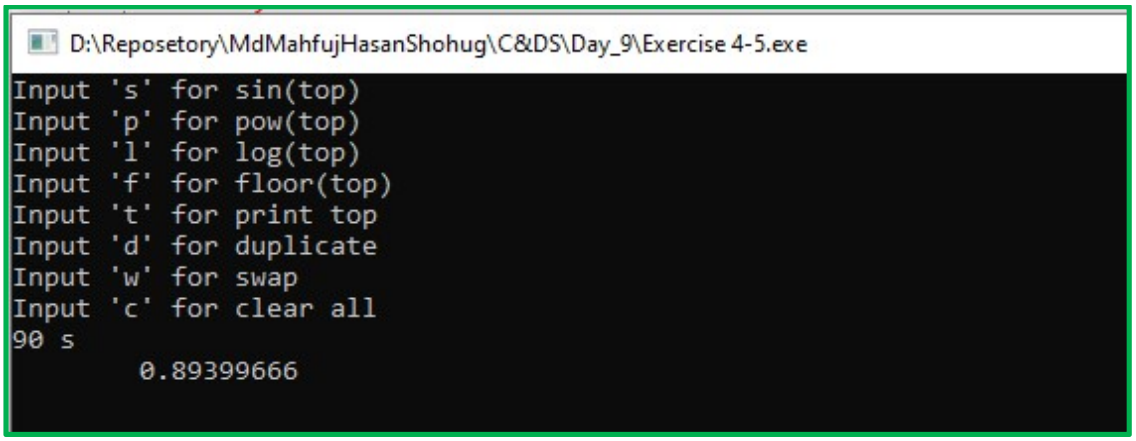
duplicateTop() Duplicates the top element of the stack.

swapTopTwo() swaps the top two elements of the stack.

clearStack() clears the entire stack.

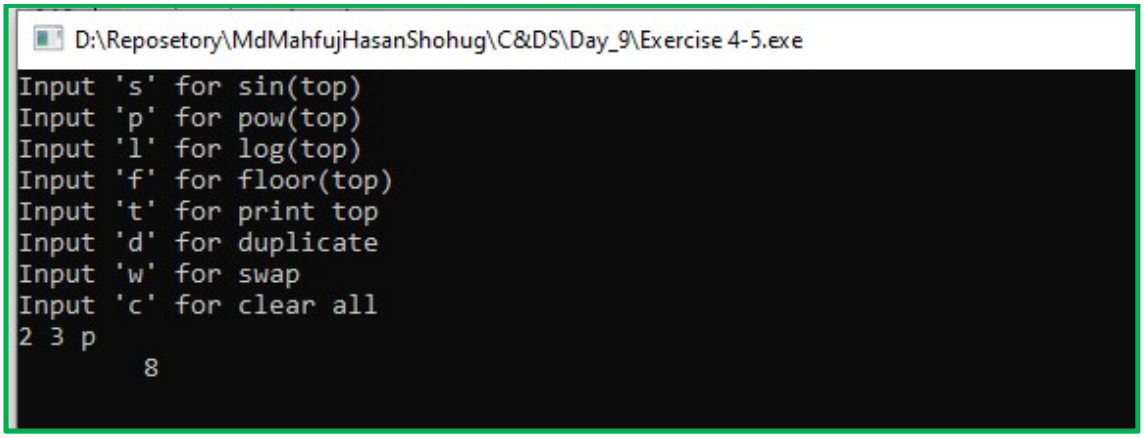
The code defines BUFSIZE as the size of the buffer for character input using the getch() and ungetch() functions. buf is an array used as a buffer to hold characters and bufp keeps track of the next free position in the buffer. The getch() and ungetch() functions are used to read characters from input, including the ability to return characters to the input stream. The getop() function is responsible for tokenizing the input and returning the next character or numeric operand. It handles numbers including integers and fractions as well as negative numbers. The PerformOperation() function performs operations based on the operator. It supports basic arithmetic operations such as addition, subtraction, multiplication, division and modulo. It also includes additional operations such as printing the top element, duplicating the top element, swapping the top two elements, and clearing the stack. Additional operations added to the code provide more flexibility and functionality for manipulating the stack during the calculation process.

Input 's' for sin(top)



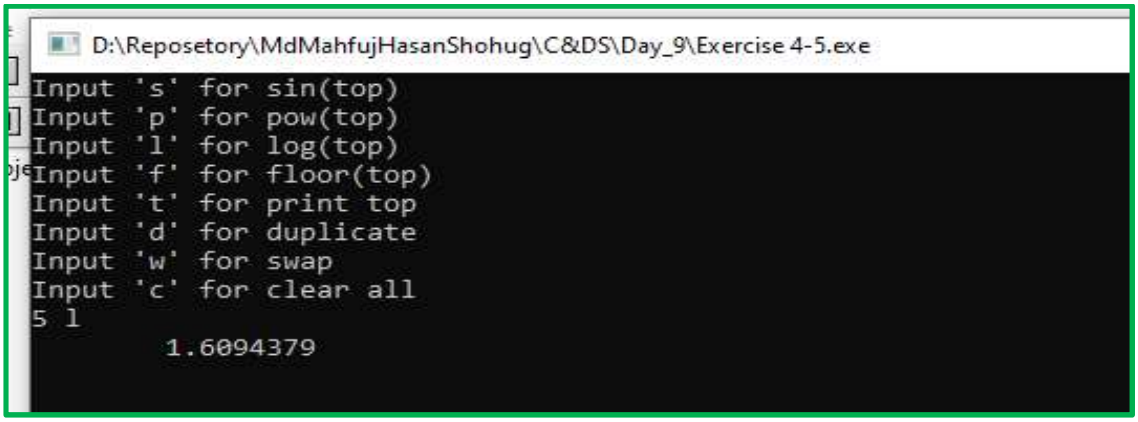
Sin(90) = 0.8939.

Input 'p' for pow(top)

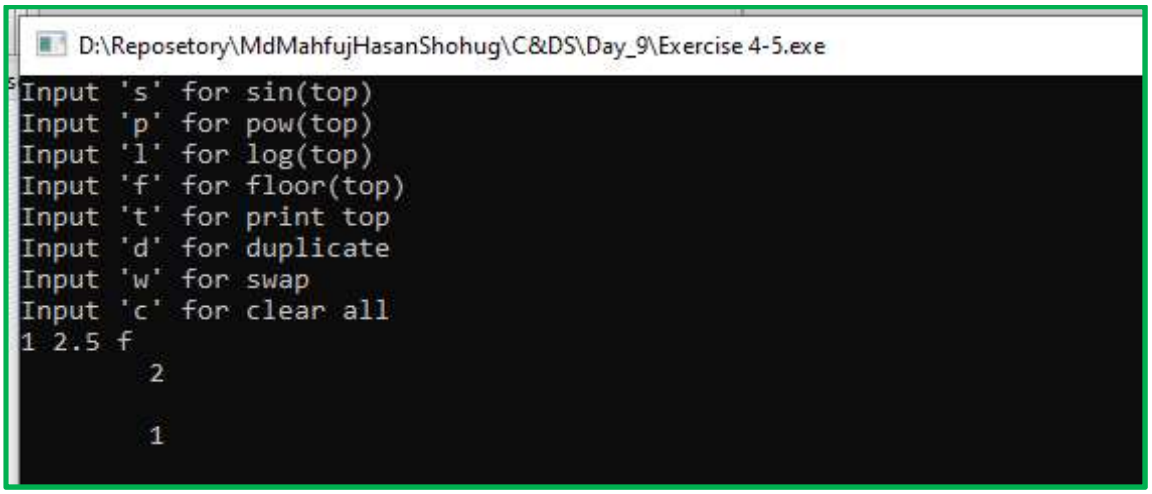


$2^3 = 8$

Input 'l' for log(top)



'f' for Floor:



6. Exercise 4-6:

Problem: Add commands for handling variables. (It's easy to provide twenty-six variables with single-letter names.) Add a variable for the most recently printed value.

Solution: Firstly, we need run code for better understand here is the source code then describe this problem and after then the solution and then with some outputs:

Here is the source code of it:

```
#include <stdio.h>
#include <stdlib.h> // for atof()
#include <ctype.h>
#include <math.h>
#include <string.h> // for strlen()

#define MAXOP 100      // max size of operand or operator
#define NUMBER '0'     // signal that a number was found
#define SETVARIABLE 's' // signal that a variable is being assigned
#define GETVARIABLE 'g' // signal that a variable is being retrieved
#define ERRORSIGNAL ' ' // signal that an error occurred in getop
#define MAXVAL 100     // maximum depth of val stack
#define BUFSIZE 100    // buffer size for getch and ungetch
#define NUMVARS 27     // number of variables supported (a-z) plus the variable L for last printed

/*****************************************************************************
*
** Function Name: main, gives on the function header section
                  **
** Inputs          : 1. argc   -- The number of parameters provided to the main function**
**                                  : 2. argv   -- The pointer to the input string array of parameters   **
** Variable                  : Describe on the comment line
                                                    **
** Return          : = 0          -- Success
                                                **
**                                  : <          0              -- Failed
                                                                    **
```

```
** Note                          :Add a variable for the most recently printed value
                                  **
*****************************************************************************
**/


enum boolean {FALSE, TRUE};

int sp = 0;                 // next free stack position
double val[MAXVAL];           // value stack
char buf[BUFSIZE];            // buffer for ungetch
int bufp = 0;                // next free position in buf
double varVals[NUMVARS] = {0.0}; // array to store the double values for variables. Supports variables a
through z (lower case). Initial value is 0

//function header
int getop(char s[]);
void push(double f);
double pop(void);
int getch(void);
void ungetch(int c);
void printTop(void);
void duplicateTop(void);
void swapTopTwo(void);
void hints(void);
// reverse Polish calculator
// note: convert ((((-1 - 2) * (4 + -5)) / -3) % 5) * (-1 - -10) to -1 2 - 4 -5 + * -3 / 5 % -1 -10 - * for reverse
Polish notation. -1 2 - 4 -5 + * -3 / 5 % -1 -10 - * == -9
int main()
{
   int type;
   double op2;
   char s[MAXOP];
   char skipNextNewline = FALSE;
   hints();
   printf("\n");
   while ((type = getop(s)) != EOF)
   {
      switch (type)
      {
         case NUMBER:
            push(atof(s)); // convert the string to type double and push it on the stack
            break;
         case SETVARIABLE:
            if (strlen(s) > 2 && s[1] == '=')
            {
               int v = s[0];         // stores variable
               int i = 1;           // start at 1 since while loop needed ++i for a few reasons
               while (s[++i] != '\0')  // this removes the variable name= part of s e.g. if s == "a=123.45" after
loop s == "123.45"
               s[i - 2] = s[i];    // shifts chars two to the left by 2
               s[i - 2] = '\0';       // since '\0' isn't copied, terminate string manually
               varVals[v - 'a'] = atof(s); // convert string to double and store it in array
            }
            else
               printf("error: set variable length too small or incorrectly formatted (%s)\n", s);

            skipNextNewline = TRUE;
```

```
            break;
        case GETVARIABLE:
            push(varVals[s[0] - 'a']);  // convert the variable name to stored value
            break;
        case '+':
            push(pop() + pop()); // pop last two digits to sum them and push the result on the stack
            break;
        case '*':
            push(pop() * pop()); // pop last two digits to multiply them and push the result on the stack
            break;
        case '-':
        /*
            Because + and * are commutative operators, the order in which the popped operands are combined is irrelevant, but for - and / the left and right operands
            must be distinguished. In push(pop() - pop());, the order in which the two calls of pop are evaluated is not defined. To guarantee the right order, it is
            necessary to pop the first value into a temporary variable. Hence op2 = pop() in - and / but not in + and *
        */
            op2 = pop();
            push(pop() - op2); // pop last two digits to subtract them in the correct order and push the result on the stack
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2); // pop last two digits to divide them in the correct order and push the result on the stack
            else
                printf("error: zero divisor\n");
            break;
        case '%':
            op2 = pop();
            if (op2 != 0.0)
                push(fmod(pop(), op2)); // pop last two digits in the correct order to find the modulus and push the result on the stack
            else
                printf("error: zero divisor\n");
            break;
        case 'p':
            printTop();
            skipNextNewline = TRUE;
            break;
        case 'd':
            duplicateTop();
            skipNextNewline = TRUE;
            break;
        case 'w':
            swapTopTwo();
            skipNextNewline = TRUE;
            break;
        case '!':
        // sets next free stack position to zero (meaning the value stack is empty).
        // all of the original values are still there, but they will no longer be accessible by the current functions and they will be overwritten when new elements are stored
            sp = 0;
            skipNextNewline = TRUE;
            break;
```

```
            case 'L':
               push(varVals[NUMVARS - 1]); // adds the last printed value to the stop of the stack
               break;
            case '\n':
               if (skipNextNewline)
                  skipNextNewline = FALSE;
               else
               {
                  varVals[NUMVARS - 1] = pop(); // updates last printed value
                  printf("\t%.8g\n", varVals[NUMVARS - 1]); // get the final result
               }
               break;
            default:
               printf("error: unknown command %s\n", s);
               break;
         }
      }
      return 0;
   }

   // push: push f onto value stack
   void push(double f)
   {
      if (sp < MAXVAL) // if value stack still has space, add f
         val[sp++] = f;
      else
         printf("error: stack full, can't push %g\n", f);
   }

   // pop: pop and return top value from stack
   double pop(void)
   {
      if (sp > 0) // if the next free stack position is greater than zero, return the highest level item from stack
         return val[--sp];
      else
      {
         printf("error: stack empty\n");
         return 0.0;
      }
   }

   // getop: get next operator or numeric operand
   int getop(char s[])
   {
      int i, c;
      char setVar = FALSE;
      i = 0;

      while ((s[0] = c = getch()) == ' ' || c == '\t') // skip white space
         ;
      s[1] = '\0'; // terminate string in case input is not a number (s is expected to be a string throughout
   program)
      if (c >= 'a' && c <= 'z')
      {
         if ((s[++i] = c = getch()) == '=') // get next char and check if it was an equal symbol. Update s in case of
   error
            setVar = TRUE;
         else if (c == ' ' || c == '\t' || c == '\n')
```

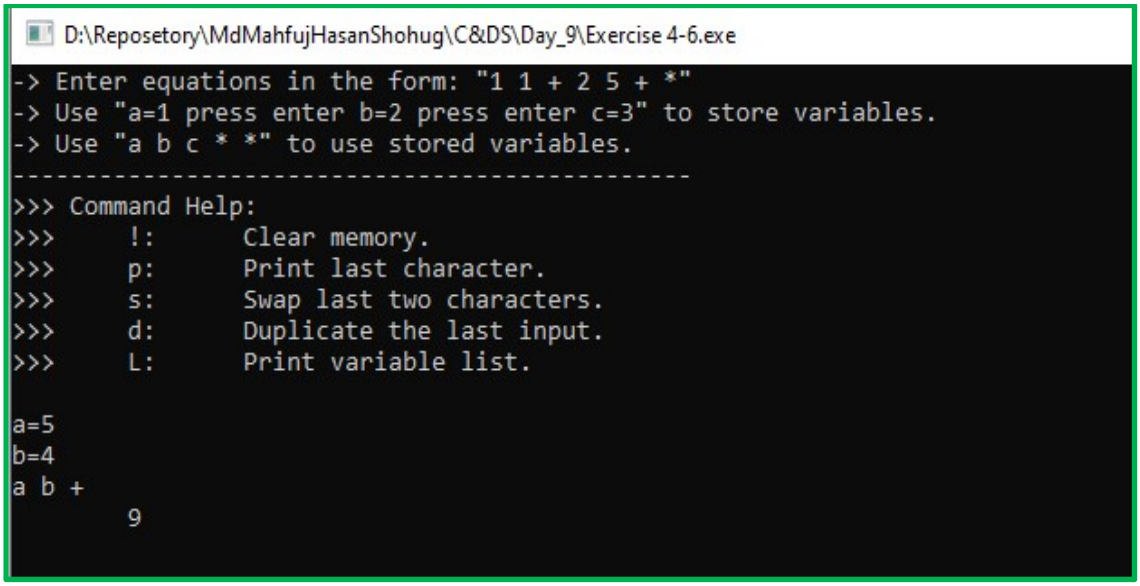
```
        {
            ungetch(c); // return the whitespace since it will be processed later
            return GETVARIABLE;
        }

        if (!(setVar && ((s[++i] = c = getch()) == '-' || isdigit(c))))
            return ERRORSIGNAL; // triggers an error and will display what is in s
    }
    if (!isdigit(c) && c != '.' && c != '-')
        return c; // not a number. Probably an operator, so return it. Minus operator is a special case and is
handled right before return NUMBER;
    if (c == '-' || isdigit(c)) // collect integer(s), if any, after first digit found or after minus symbol found
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.') // collect fraction part if period is found
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0'; // terminate string after digits were captured
    if (c != EOF)
        ungetch(c); // since we read to far, push the last read char back on the getch buffer. This buffer is read
first before getting the next char from input
    if (i == 1 && s[0] == '-') // if s[0] == '-' && s[1] == '\0', return minus operator
        return '-';
    if (setVar)
        return SETVARIABLE;
    else if (c >= 'a' && c <= 'z')
        return ERRORSIGNAL; // if last char is a variable, throw error

    return NUMBER;
}

// get a (possibly pushed back) character
// checks to see if there are any chars in buffer. If there are, get those and return it. If not, call getchar()
from stdio.h to get next char from input
int getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

// push character back on input
// if bufp is less than BUFSIZE, there is room to store more chars to be read by getch next and it stores c
and updates the index for it
void ungetch(int c)
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}

// prints the top element in the value stack
void printTop(void)
{
    if (sp > 0)
        printf("\t%.8g\n", val[sp - 1]);
    else
        printf("error: stack empty\n");
}
```

```
void duplicateTop(void)
{
   if (sp < MAXVAL) // only need to see if there is space for one more
      push(val[sp - 1]); // duplicates top item
   else
      printf("error: stack full, can't duplicate top element\n");
}

void swapTopTwo(void)
{  // if sp == 2, there are at least two elements stored
   if (sp > 1)
   {                        // <third> <second> <first>
      double first = pop();      // <third> <second>
      double second = pop();     // <third>
      push(first);               // <third> <first>
      push(second);              // <third> <first> <second>
   }
   else
      printf("error: can't swap top two, not enough elements\n");
}

void hints()
{
   printf("-> Enter equations in the form: \"1 1 + 2 5 + *\"\n");
   printf("-> Use \"a=1 press enter b=2 press enter c=3\" to store variables.\n");
   printf("-> Use \"a b c * *\" to use stored variables.\n");
   printf("-------------------------------------------------\n");
   printf(">>> Command Help:\n");
   printf(">>>    !:     Clear memory.\n");
   printf(">>>    p:     Print last character.\n");
   printf(">>>    s:     Swap last two characters.\n");
   printf(">>>    d:     Duplicate the last input.\n");
   printf(">>>    v:     Print variable list.\n");
}
```

On this code: all function works same as before now here is I understanding : The given code is an implementation of a Reverse Polish Notation (RPN) calculator in C. The RPN calculator evaluates mathematical expressions using a stack-based method. Calculator supports basic arithmetic operations, variable assignment and retrieval, as well as additional commands for manipulating the stack. The main function serves as the entry point of the program. It repeatedly calls the getup function to retrieve the next operator or operand from the input. The getop function reads characters from the input and determines their type, returning an appropriate code such as NUMBER, setvariable, obtainable, or an operator symbol. The push function is responsible for pushing numbers onto the value stack, and the pop function retrieves the top value from the stack. These functions interact with the val array, which acts as a stack to store values. The calculator supports standard arithmetic operators: addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). When an operator is encountered, the required operand is popped from the stack, the operation is performed, and the result is pushed back onto the stack.

The code also contains additional commands to manipulate the stack. The 'p' command prints the top element of the stack, 'd' duplicates the top element and 'w' swaps the top two elements. Variable assignment and retrieval is implemented using a separate array, varVals, that stores the value assigned to the variable. The code supports 'a' to 'z' (lowercase) variables and **stores the last printed value in the 'L' variable.** The program provides a basic user interface with a command prompt and supports entering equations, variable assignments, and commands. It also has a hint function that provides guidance on how to use the calculator. Briefly, the code implements a simple RPN calculator with support for basic arithmetic operations, variable assignment and retrieval, and stack manipulation commands. It demonstrates basic input/output operations as well as the use of stacks and arrays to store and manipulate data.

Outputs:



Here a = 5 and b = 4 and the a + b = 9 here is the answer. Other outputs I was show before 4 and 5 number of question.

7. Exercise 4-7:

Problem: Write a routine ungets(s) that will push back an entire string onto the input. Should ungets know about buf and bufp, or should it just use ungetch?

Solution: For the question first part the ungetch(c) function should be used by the ungets(s) procedure to push each character back onto the input. It doesn't necessarily need to be aware of buf and bufp. The ungets(s) method is implemented as follows source code:

Exercise 4-7.c

```c
#include <stdio.h>
#include <string.h>

#define BUFSIZE 5 //set buffer size
/*****************************************************************************
** Function Name: main, getInput, getch, ungetch, ungets                    **
** Inputs       : 1. argc   -- The number of parameters provided to the main function**
**              : 2. argv   -- The pointer to the input string array of parameters   **
** Variable     : buf[BUFSIZE]  -- Global variable                           **
**              : low_temp  -- lowest value of tempture 0                    **
**              : s[]       --  inputed string                           **
**              : i,        --  loop variable                            **
** Return       : = 0       -- Success                                       **
**              : < 0       -- Failed                                        **
** Note         :ungets(s) that will push back an entire string onto the input       **
*****************************************************************************/

char buf[BUFSIZE]; /* buffer for ungetch */
int bufp = 0; /* next free position in buf */

// Function to get user input and store it in a string
void getInput(char s[])
{
    int i = 0;
    printf("Enter any string: ");
    while ((s[i] = getchar()) != '\n')
    {
        i++;
    }
    s[i] = '\0'; // Null-terminate the input string
}

// Function to get a (possibly pushed-back) character
int getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

// Function to push character back on input
void ungetch(int c)
{
    if (bufp >= BUFSIZE)
    {
        printf("ungetch: too many characters\n");
    } else
    {
        buf[bufp++] = c;
        printf("%c\n", c);
    }
}

// Function to push an entire string onto the input
void ungets(const char* s)
{
```

```
55      size_t len = strlen(s);
56      printf("Before ungetch: \n");
57      while (len > 0)
58      {
59          ungetch(s[--len]);
60      }
61  }
62
63  int main(int argc, char *argv[])
64  {
65      char s[BUFSIZE];
66      getInput(s); // Get user input and store it in string s
67      ungets(s); // Push the string s onto the input
68      int c;
69      printf("After ungetch:\n");
70      while ((c = getch()) != EOF)
71      {
72          printf("%c", c); // Print characters retrieved from the input
73      }
74      return 0;
75  }
```

ces | Compile Log | Debug | Find Results | Close

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: D:\Reposetory\MdMahfujHasanShohug\C&DS\Day_9\Exercise 4-7.exe
- Output Size: 130.0498046875 KiB
- Compilation Time: 0.64s
```

For the 2nd part of question answer that by iterating through the string starting at the end and executing ungetch() for each character, the ungets(s) method uses the ungetch(c) function to push each character of the string onto the input.

The buffer management is handled by the ungetch() method, which makes sure that the characters are placed in buf and that the index bufp is updated appropriately. In this way, ungetch(c) can take care of it, and ungets(s) no longer needs to be aware of the specifics of the buffer implementation. Therefor I think for better understand used ungets know about buf and bufp, but should I also just use ungetch.

Here is two outputs of this code:

```
D:\Reposetory\MdMahfujHasanShohug\C&DS\Day_9\Exercise 4-7.exe
Enter any string: hello
Before ungetch:
o
l
l
e
h
After ungetch:
hello
```

Here that's showing correct output.

But if I input my buffersize above maximum the extra char give signal too many characters Here is the output if I get buffersize = 5.