

## Documentation of Day 9

### Exercise 4-1

#### Question:

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

#### Source Code:

```
#include <stdio.h>

#define MAXLINE 1000 /* maximum input line length */

int getline(char line[], int max);
int strindex(char source[], char searchfor[]);

char pattern[] = "ould"; /* pattern to search for */

int strindex(char s[], char t[])
{
    int i, j, k;
    int rightmost = -1;

    for (i = 0; s[i] != '\0'; i++)
    {
        for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++);

        if (k > 0 && t[k] == '\0')
            rightmost = i;
    }

    return rightmost;
}

int main()
{
    char line[MAXLINE];
    int position;
    printf("The string to be matched is: '%s'\n", pattern);
    while (getline(line, MAXLINE) > 0)
    {
        position = strindex(line, pattern);

        if (position >= 0)
            printf("Rightmost occurrence found at position: %d\n", position);
        else
            printf("Pattern not found.\n");
    }

    return 0;
}
```

```

}

int getline(char s[], int lim)
{
    int c, i;
    i = 0;

    while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
        s[i++] = c;

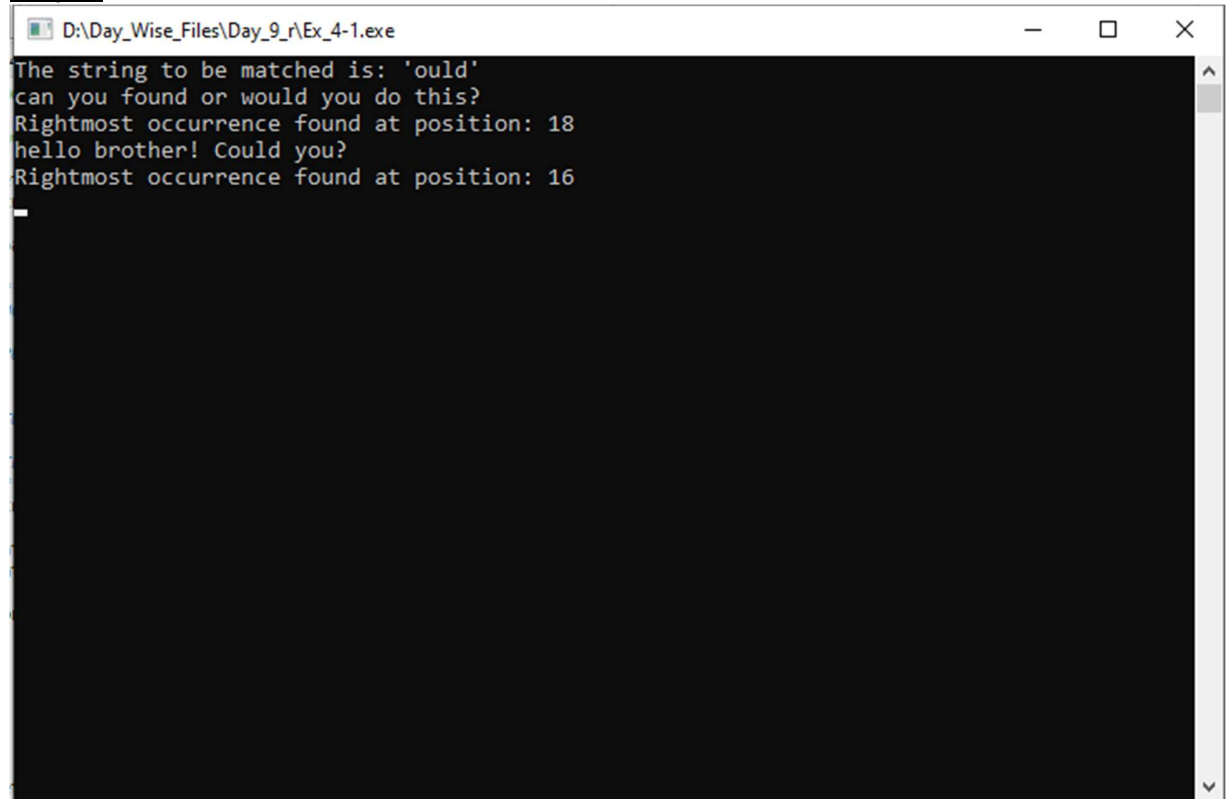
    if (c == '\n')
        s[i++] = c;

    s[i] = '\0';
    return i;
}

```

#### **Explanation:**

1. The code starts by including the necessary header file `stdio.h`, which provides functions for input/output operations.
2. The `MAXLINE` constant is defined to specify the maximum length of an input line.
3. The code declares two function prototypes: `int getline(char line[], int max);` and `int strindex(char source[], char searchfor[]);`. These prototypes specify the functions used in the program.
4. The code declares a character array called `pattern` and initializes it with the string `"ould"`. This is the pattern that will be searched for within the input lines.
5. The `strindex` function is defined, which takes two character arrays `s` and `t` as input and returns the index of the rightmost occurrence of `t` within `s`. If `t` is not found within `s`, it returns `-1`. This function uses nested loops to iterate through the characters of `s` and `t` and checks for matching characters.
6. The `main` function is defined, which is the entry point of the program. It declares a character array called `line` with a size of `MAXLINE`, which is used to store the input lines.
7. The `printf` statement outputs the pattern that will be searched for.
8. The program enters a `while` loop, which repeatedly reads input lines using the `getline` function until the end of input is reached (`getline(line, MAXLINE) > 0`).
9. Within the loop, the `strindex` function is called with the current line and the pattern as arguments. The return value is stored in the variable `position`.
10. If `position` is greater than or equal to `0`, it means the pattern was found in the line, and the program prints the position of the rightmost occurrence using `printf`.
11. If `position` is `-1`, it means the pattern was not found in the line, and the program prints a message indicating that the pattern was not found.
12. The program continues to read and process input lines until the end of input is reached.
13. Finally, the `main` function returns `0`, indicating successful execution of the program.

**Output:**

```
D:\Day_Wise_Files\Day_9_r\Ex_4-1.exe
The string to be matched is: 'ould'
can you found or would you do this?
Rightmost occurrence found at position: 18
hello brother! Could you?
Rightmost occurrence found at position: 16
_
```

**Exercise 4-2****Question:**

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.

**Source Code:**

```
#include <stdio.h>
#include <ctype.h>
double my_atof(char s[]);

int main()
{
    char input1[] = "123.45e-6";
    double result1 = my_atof(input1);
    printf("Input: %s\n", input1);
    printf("Output: %.10f\n\n", result1);

    char input2[] = "2.71828e3";
    double result2 = my_atof(input2);
    printf("Input: %s\n", input2);
    printf("Output: %.10f\n\n", result2);
}
```

```

char input3[] = "-9.8765E+2";
double result3 = my_atof(input3);
printf("Input: %s\n", input3);
printf("Output: %.10f\n\n", result3);

return 0;
}

double my_atof(char s[]) {
    double val, power;
    int i, sign, exp_sign, exponent;

    for (i = 0; isspace(s[i]); i++) /* skip white space */
        ;

    sign = (s[i] == '-') ? -1 : 1;

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

    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');

    if (s[i] == '.')
        i++;

    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10;
    }

    if (s[i] == 'e' || s[i] == 'E')
        i++;

    exp_sign = (s[i] == '-') ? -1 : 1;

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

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

    double exp_multiplier_pos = 1.0;
    double exp_multiplier_neg = 1.0;

    if (exp_sign == 1) {
        while (exponent > 0) {

```

```

        exp_multiplier_pos *= 10.0;
        exponent--;
    }
    return sign * val / power * exp_multiplier_pos;
}
if(exp_sign==1) {
    while (exponent > 0) {
        exp_multiplier_neg /= 10.0;
        exponent--;
    }
    return sign * val / power * exp_multiplier_neg;
}
}

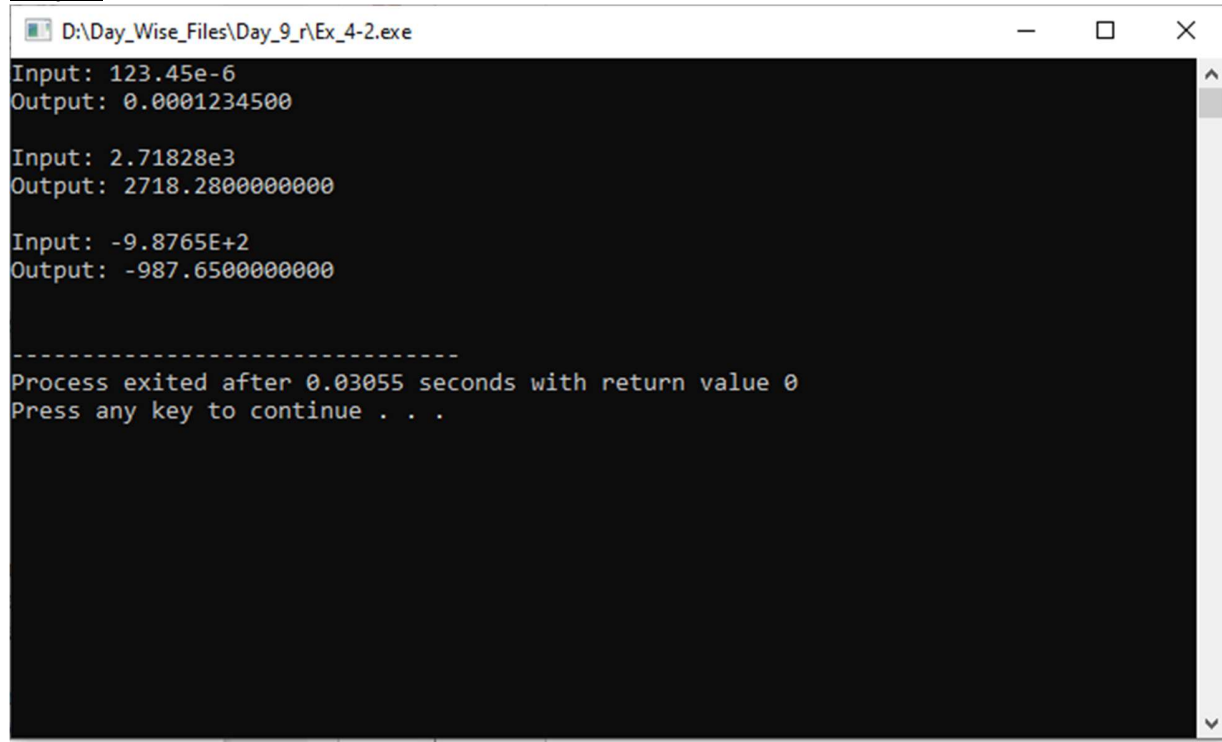
```

### **Explanation:**

1. The code includes two header files: `stdio.h` for input/output operations and `ctype.h` for character handling functions.
2. The code declares the function prototype `double my_atof(char s[]);` before the main function. This prototype specifies the function used in the program.
3. The main function is defined, which is the entry point of the program. It demonstrates the usage of the `my_atof` function by converting two input strings into their double values and printing the results.
4. Two input strings, `input1` and `input2`, are declared and initialized with the input values to be converted.
5. The `my_atof` function is called with `input1` as an argument, and the return value is stored in `result1`. The converted value is then printed using `printf`.
6. The process is repeated for `input2`, and the converted value is printed.
7. Finally, the main function returns 0, indicating successful execution of the program.
8. The `my_atof` function is defined, which takes a character array `s` as input and returns a double value. It performs the conversion from the input string to a double value following the rules of floating-point number representation.
9. The function begins by initializing variables `val` and `power` to hold the value and power parts of the floating-point number, respectively.
10. The function uses a loop to skip any leading white spaces in the input string.
11. The function checks the sign of the number (positive or negative) and assigns it to the `sign` variable.
12. If a plus or minus sign is encountered, it is skipped in the input string.
13. Another loop is used to process the digits before the decimal point. The function multiplies the existing `val` by 10 and adds the value of the current digit in the input string.
14. The function also maintains a power variable to keep track of the position of the decimal point. It is multiplied by 10 for each digit encountered.
15. If an 'e' or 'E' character is encountered, indicating the presence of an exponent, the function proceeds to parse the exponent.
16. The function checks the sign of the exponent and assigns it to the `exp_sign` variable.
17. If a plus or minus sign is encountered, it is skipped in the input string.

18. Another loop is used to process the digits representing the exponent value.
19. After parsing the exponent, the function calculates two multipliers: `exp_multiplier_pos` and `exp_multiplier_neg`. These multipliers adjust the value based on the exponent sign.
20. If the exponent is positive, the function divides `exp_multiplier_neg` by 10 repeatedly until the exponent becomes 0. This reduces the value according to the exponent.
21. The function returns the final result, which is the product of the sign, value, power, and exponent multipliers.

#### **Output:**



```
D:\Day_Wise_Files\Day_9_r\Ex_4-2.exe
Input: 123.45e-6
Output: 0.0001234500

Input: 2.71828e3
Output: 2718.2800000000

Input: -9.8765E+2
Output: -987.6500000000

-----
Process exited after 0.03055 seconds with return value 0
Press any key to continue . . .
```

#### **Exercise 4-3**

##### **Question:**

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

##### **Source Code:**

```
#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 */
#define BUFSIZE 100

int getop(char []);
void push(double);
double pop(void);
```

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

```
/* 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;
    }
}
```

```
int getch(void);
void ungetch(int);
```

```
int gettop(char s[]) {
    int i, c;

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

    s[1] = '\0'; /*ensures that s initially holds the first non-whitespace character*/

    if (!isdigit(c) && c != '.' && c != '-')
        return c; /*This covers cases where the character represents an operator or an invalid input*/

    if (c == '-') {
        int next = getch(); /*get the next character*/

        if (isdigit(next) || next == '.') { /*next char of after '-' is digit or '.' is checked*/
            s[1] = next; /*Store the numeric value in the array*/
            i = 1;

            while (isdigit(s[++i] = c = getch()))
                ;
        }
    }
}
```

```

        if (c == '.') {
            while (isdigit(s[++i] = c = getch()))
                ;
        }

        s[i] = '\0';

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

        return NUMBER;
    } else {
        if (next != EOF)
            ungetch(next); /*Push back the non-digit character*/

        return c; /*Return '-' as an operator*/
    }
}

i = 0;

if (isdigit(c)) {
    while (isdigit(s[++i] = c = getch()))
        ;

    if (c == '.') {
        while (isdigit(s[++i] = c = getch()))
            ;
    }

    s[i] = '\0';

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

    return NUMBER;
}

return c; /*Return the non-digit character as an operator*/
}

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;
}

int main() {
    int type;
    double op2;
    char s[MAXOP];
    while ((type = getop(s)) != EOF) {
        switch (type) {
            case NUMBER:
                push(atof(s));
                break;

            case '+':
                push(pop() + pop());
                break;

            case '*':
                push(pop() * pop());
                break;

            case '-':
                op2 = pop();
                push(pop() - op2);
                break;

            case '/':
                op2 = pop();
                if (op2 != 0.0)
                    push(pop() / op2);
                else
                    printf("error: zero divisor\n");
                break;

            case '%':
                op2 = pop();
                if (op2 != 0.0)
                    push((int)pop() % (int)op2);
                else
                    printf("error: zero divisor\n");
                break;

            case '\n':

```

```

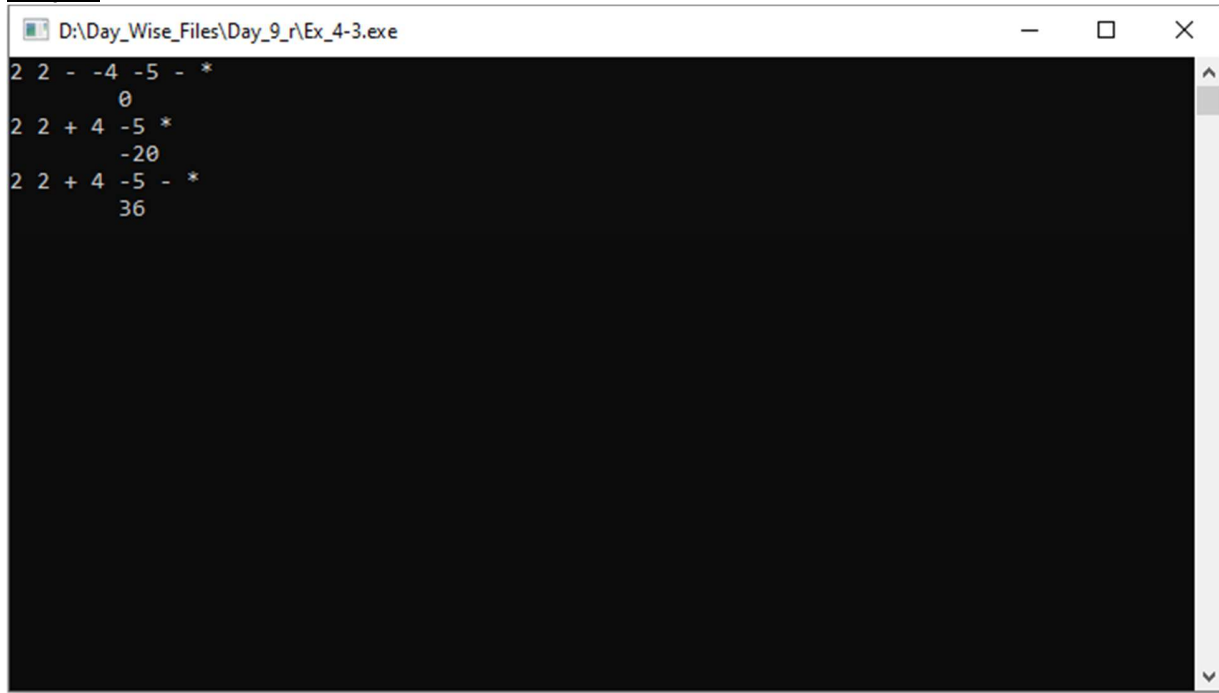
        printf("\t%.8g\n", pop());
        break;

        default:
            printf("error: unknown command %s\n", s);
            break;
    }
}
return 0;
}

```

### **Explanation:**

1. **int main(void):** This function is the entry point of the program and controls the overall execution of the calculator.
  - It declares variables type and op2 to store the type of the input token and the second operand for binary operations, respectively.
  - It declares an array s[MAXOP] to store the input token (operator or operand).
  - The program enters a while loop that continues until the end of input (EOF is encountered).
  - Inside the loop, it uses a switch statement to handle different types of input tokens.
  - If the token is a number (NUMBER), it converts it to a double using atof and pushes it onto the stack using the push function.
  - If the token is an operator (+, -, \*, /, %), it performs the corresponding operation by popping operands from the stack, and then pushes the result back onto the stack using the push function.
  - If the token is a newline character (\n), it pops the final result from the stack and prints it.
  - If the token is neither a number nor an operator, it prints an error message.
  - After processing the input, the main function returns 0 to indicate successful execution.
2. **int getop(char s[]):** This function is responsible for reading the next input token from the user.
  - It declares variables i and c.
  - It uses a while loop to skip any leading whitespace characters.
  - It assigns the first non-whitespace character to s[0] and appends a null terminator to create a string.
  - It checks if the character is not a digit, period, or minus sign. If so, it returns the character as an operator.
  - If the character is a minus sign, it checks the next character using getch().
  - If the next character is a digit or period, it stores the minus sign in s[1] and continues reading digits and the decimal point into s until it encounters a non-digit character.
  - If the next character is not a digit or period, it pushes it back using ungetch() and returns the minus sign as an operator.
  - If the character is a digit, it reads the remaining digits and decimal point into s until it encounters a non-digit character.
  - It appends a null terminator to s and checks if the non-digit character is EOF. If not, it pushes it back using ungetch().
  - Finally, it returns NUMBER to indicate that the token in s is a number.

**Output:**

```
D:\Day_Wise_Files\Day_9_r\Ex_4-3.exe
2 2 - -4 -5 - *
0
2 2 + 4 -5 *
-20
2 2 + 4 -5 - *
36
36
```

**Exercise 4-4****Question:**

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.

**Source Code:**

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

#define MAXOP 100
#define MAXVAL 100
#define BUFSIZE 100
#define NUMBER '0'

int getop(char[]);
void push(double);
double pop(void);
double printTop(void);
void duplicate(void);
void swapTop(void);
void clearStack(void);

int main(void) {
    int type;
```

```

double op2;
char s[MAXOP];

printf("Reverse Polish calculator\n");
printf("Enter numbers and operators (+, -, *, /, %, p, d, s, c).\n");
printf("Type 'p' to print the top\n");
printf("Type 'd' to duplicate the top\n");
printf("Type 's' to swap the top two elements\n");
printf("Type 'c' to clear the stack.\n");

printf("Type 'q' to quit.\n");

while ((type = gettop(s)) != 'q') {
    switch (type) {
        case NUMBER:
            push(atof(s));
            break;

        case '+':
            push(pop() + pop());
            break;

        case '-':
            op2 = pop();
            push(pop() - op2);
            break;

        case '*':
            push(pop() * pop());
            break;

        case '/':
            op2 = pop();

            if (op2 != 0.0) {
                push(pop() / op2);
            }
            else {
                printf("Error: zero divisor.\n");
            }

            break;

        case '%':
            op2 = pop();

            if (op2 != 0.0) {
                push((int)pop() % (int)op2);
            }
    }
}

```

```

    }
    else {
        printf("Error: zero divisor.\n");
    }
    break;

case 'p':
    printTop();
    break;

case 'd':
    duplicate();
    break;

case 's':
    swapTop();
    break;

case 'c':
    clearStack();
    break;

case '\n':
    printf("result: %.8g\n", printTop());
    break;

default:
    printf("Error: unknown command %s.\n", s);
    break;
}
}

printf("Exiting the Calculator.\n");

return 0;
}

int sp = 0;
double val[MAXVAL];

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

```

```

double pop(void) {
    if (sp > 0) {
        return val[--sp];
    }
    else {
        printf("Error: stack empty.\n");
        return 0.0;
    }
}

```

```

double printTop(void) {
    if (sp > 0) {
        return val[sp - 1];
    }
    else {
        return -1;
    }
}

```

```

void duplicate(void) {
    if (sp > 0 && sp < MAXVAL) {
        val[sp] = val[sp - 1];
        sp++;
    }
    else if (sp >= MAXVAL) {
        printf("Error: stack full, can't duplicate.\n");
    }
    else {
        printf("Error: stack empty, can't duplicate.\n");
    }
}

```

```

void swapTop(void) {
    if (sp > 1) {
        double temp = val[sp - 1];
        val[sp - 1] = val[sp - 2];
        val[sp - 2] = temp;
    }
    else if (sp == 1) {
        printf("Error: only one element in the stack, can't swap.\n");
    }
    else {
        printf("Error: stack empty, can't swap.\n");
    }
}

```

```

void clearStack(void) {

```

```

    sp = 0;
    printf("Stack cleared.\n");
}

int bufp = 0;
char buf[BUFSIZE];

int getch(void) {
    return (bufp > 0) ? buf[--bufp] : getchar();
}

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

int gettop(char s[]) {
    int i, c;

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

    s[1] = '\0';

    if (!isdigit(c) && c != '.' && c != '-')
        return c;

    if (c == '-') {
        int next = getch();

        if (isdigit(next) || next == '.') {
            s[1] = next; // Store the '-' in the array
            i = 1;

            while (isdigit(s[++i] = c = getch()))
                ;

            if (c == '.') {
                while (isdigit(s[++i] = c = getch()))
                    ;
            }
        }

        s[i] = '\0';
    }
}

```

```

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

    return NUMBER;
}
else {
    if (next != EOF)
        ungetch(next); // Push back the non-digit character

    return c; // Return '-' as an operator
}
}

i = 0;

if (isdigit(c)) {
    while (isdigit(s[++i] = c = getch()))
        ;

    if (c == '.') {
        while (isdigit(s[++i] = c = getch()))
            ;
    }

    s[i] = '\0';

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

    return NUMBER;
}

return c; // Return the non-digit character as an operator
}

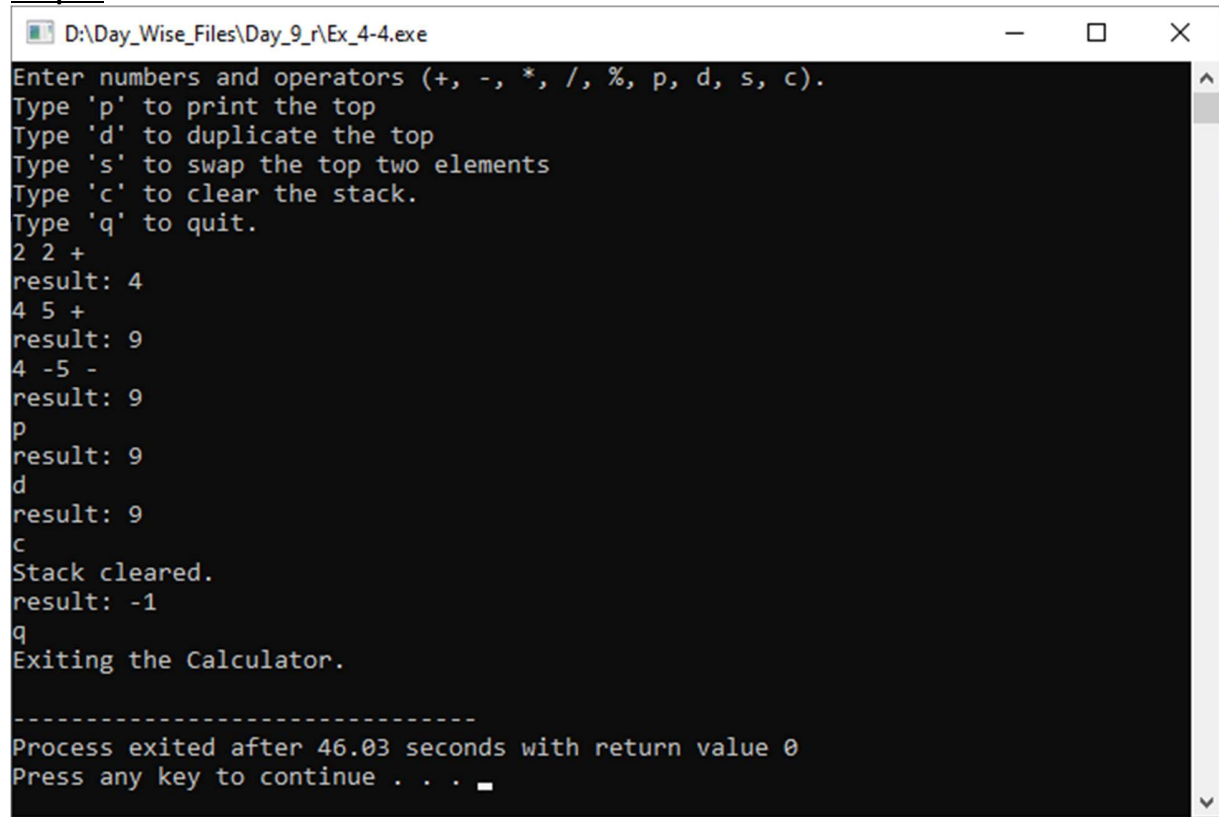
```

### **Explanation:**

1. **int main(void):** This function is the entry point of the program and controls the overall execution of the calculator.
  - It starts by printing some instructions for the user on how to interact with the calculator.
  - Inside a while loop, it reads the next input token using the getch function until the user enters 'q' to quit.
  - It uses a switch statement to handle different types of input tokens.
  - If the token is a number (NUMBER), it converts it to a double using atof and pushes it onto the stack using the push function.
  - If the token is an operator (+, -, \*, /, %), it performs the corresponding operation by popping operands from the stack and then pushing the result back onto the stack using the push function.



- It includes additional operations like printing the top element of the stack (p), duplicating the top element (d), swapping the top two elements (s), and clearing the stack (c), which are implemented using separate functions.
  - If the token is a newline character (\n), it prints the top element of the stack as the result.
  - If the token is neither a number, an operator, nor a newline character, it prints an error message.
  - After processing the input, it prints a message indicating that the calculator is exiting and returns 0 to indicate successful execution.
2. **Stack Operations:** The code includes several stack-related functions:
- `void push(double f)`: This function pushes a double value onto the stack (val) if there is enough space. If the stack is full, it prints an error message.
  - `double pop(void)`: This function pops the top element from the stack (val) and returns it. If the stack is empty, it prints an error message and returns 0.0.
  - `double printTop(void)`: This function returns the value of the top element in the stack without modifying the stack. If the stack is empty, it returns -1.
  - `void duplicate(void)`: This function duplicates the top element of the stack by pushing another copy onto the stack if there is space. It prints an error message if the stack is full or empty.
  - `void swapTop(void)`: This function swaps the positions of the top two elements on the stack. If there are fewer than two elements in the stack, it prints an error message.
  - `void clearStack(void)`: This function clears the stack by resetting the stack pointer (sp) to 0 and prints a message indicating that the stack has been cleared.
3. **int getop(char s[]):** This function is responsible for reading the next input token from the user.
- It uses a while loop to skip any leading whitespace characters.
  - It assigns the first non-whitespace character to s[0] and appends a null terminator to create a string.
  - It checks if the character is not a digit, period, or minus sign. If so, it returns the character as an operator.
  - If the character is a minus sign, it checks the next character using `getch()`.
  - If the next character is a digit or period, it stores the minus sign in s[1] and continues reading digits and the decimal
4. **printTop Function:**
- The `printTop` function prints the value of the top element of the stack instead of just returning it. It prints the value with 8 decimal places using the format specifier `%g`.

**Output:**

```
D:\Day_Wise_Files\Day_9_r\Ex_4-4.exe
Enter numbers and operators (+, -, *, /, %, p, d, s, c).
Type 'p' to print the top
Type 'd' to duplicate the top
Type 's' to swap the top two elements
Type 'c' to clear the stack.
Type 'q' to quit.
2 2 +
result: 4
4 5 +
result: 9
4 -5 -
result: 9
p
result: 9
d
result: 9
c
Stack cleared.
result: -1
q
Exiting the Calculator.

-----
Process exited after 46.03 seconds with return value 0
Press any key to continue . . . _
```

**Exercise 4-5****Question:**

Add access to library functions like sin, exp, and pow.

**Source Code:**

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

#define MAXOP 100
#define MAXVAL 100
#define BUFSIZE 100
#define NUMBER '0'

int getop(char[]);
void push(double);
double pop(void);
double printTop(void);
void duplicate(void);
void swapTop(void);
void clearStack(void);
```

```

int main(void)
{
    int type;
    double op2;
    char s[MAXOP];

    printf("Stack Calculator\n");
    printf("Enter numbers and operators (+, -, *, /, %, p, d, s, c).\n");
    printf("Type 'p' to print the top\n");
    printf("Type 'd' to duplicate the top\n");
    printf("Type 't' to swap the top two elements\n");
    printf("Type 'c' to clear the stack.\n");
    printf("Type 's' to calculate sine\n");
    printf("Type 'e' to calculate exponential\n");
    printf("Type 'w' to calculate power\n");
    printf("Type 'q' to quit.\n");

    while ((type = getop(s)) != 'q') {
        switch (type) {
            case NUMBER:
                push(atof(s));
                break;

            case '+':
                push(pop() + pop());
                break;

            case '-':
                op2 = pop();
                push(pop() - op2);
                break;

            case '*':
                push(pop() * pop());
                break;

            case '/':
                op2 = pop();

                if (op2 != 0.0)
                {
                    push(pop() / op2);
                }
                else
                {
                    printf("Error: zero divisor.\n");
                }

```

```
        break;

    case '%':
        op2 = pop();

        if (op2 != 0.0) {
            push((int)pop() % (int)op2);
        }
        else {
            printf("Error: zero divisor.\n");
        }
        break;

    case 'p':
        printTop();
        break;

    case 'd':
        duplicate();
        break;

    case 't':
        swapTop();
        break;

    case 'c':
        clearStack();
        break;

    case 's':
        push(sin(pop()));
        break;

    case 'e':
        push(exp(pop()));
        break;

    case 'w':
        op2 = pop();
        push(pow(pop(), op2));
        break;

    case '\n':
        printf("result: %.8g\n", printTop());
        break;

    default:
```

```

        printf("Error: unknown command %s.\n", s);
        break;
    }
}

printf("Exiting the Stack Calculator.\n");

return 0;
}

int sp = 0;
double val[MAXVAL];

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

double pop(void)
{
    if (sp > 0) {
        return val[--sp];
    }
    else {
        printf("Error: stack empty.\n");
        return 0.0;
    }
}

double printTop(void) {
    if (sp > 0) {
        return val[sp - 1];
    }
    else {
        return -1;
    }
}

void duplicate(void) {
    if (sp > 0 && sp < MAXVAL) {
        val[sp] = val[sp - 1];
        sp++;
    }
    else if (sp >= MAXVAL) {

```

```

    printf("Error: stack full, can't duplicate.\n");
}
else {
    printf("Error: stack empty, can't duplicate.\n");
}
}

void swapTop(void) {
    if (sp > 1) {
        double temp = val[sp - 1];
        val[sp - 1] = val[sp - 2];
        val[sp - 2] = temp;
    }
    else if (sp == 1) {
        printf("Error: only one element in the stack, can't swap.\n");
    }
    else {
        printf("Error: stack empty, can't swap.\n");
    }
}

void clearStack(void) {
    sp = 0;
    printf("Stack cleared.\n");
}

int bufp = 0;
char buf[BUFSIZE];

int getch(void) {
    return (bufp > 0) ? buf[--bufp] : getchar();
}

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

int gettop(char s[]) {
    int i, c;

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

```

```

s[1] = '\0';

if (!isdigit(c) && c != '.' && c != '-')
    return c;

if (c == '-') {
    int next = getch();

    if (isdigit(next) || next == '.') {
        s[1] = next; // Store the '-' in the array
        i = 1;

        while (isdigit(s[++i] = c = getch()))
            ;

        if (c == '.') {
            while (isdigit(s[++i] = c = getch()))
                ;
        }

        s[i] = '\0';

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

        return NUMBER;
    }
    else {
        if (next != EOF)
            ungetch(next); // Push back the non-digit character

        return c; // Return '-' as an operator
    }
}

i = 0;

if (isdigit(c)) {
    while (isdigit(s[++i] = c = getch()))
        ;

    if (c == '.') {
        while (isdigit(s[++i] = c = getch()))
            ;
    }

    s[i] = '\0';

```

```

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

    return NUMBER;
}

return c; // Return the non-digit character as an operator
}

```

### **Explanation:**

Above program is an extended version of the previous reverse polished calculator program. It includes additional mathematical functions such as 'sin', 'exp', and 'pow'. Below I have discussed the newly added functions in this code:

#### 1. Additional Mathematical Functions:

- 's': When the character 's' is encountered, the program calls the sin function from the math library to calculate the sine of the top element of the stack and pushes the result back onto the stack.
- 'e': When the character 'e' is encountered, the program calls the exp function from the math library to calculate the exponential function ( $e^x$ ) of the top element of the stack and pushes the result back onto the stack.
- 'w': When the character 'w' is encountered, the program performs the power function. It pops two elements from the stack, with the top element being the exponent (op2), and the second element being the base. It then uses the pow function from the math library to calculate the result and pushes it back onto the stack

### **Output:**



```
D:\Day_Wise_Files\Day_9_r\Ex_4-5.exe
Type 'p' to print the top
Type 'd' to duplicate the top
Type 't' to swap the top two elements
Type 'c' to clear the stack.
Type 's' to calculate sine
Type 'e' to calculate exponential
Type 'w' to calculate power
Type 'q' to quit.
2 4 +
result: 6
5 8 *
result: 40
p
result: 40
t
result: 6
s
result: -0.2794155
w
result: 0.35674641
c
Stack cleared.
result: -1
q
Exiting the Stack Calculator.

-----
Process exited after 36.45 seconds with return value 0
Press any key to continue . . .
```

## **Exercise 4-6**

### **Question:**

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.

After adding the necessary commands I have implemented a Reverse Polish Notation Calculator in C. RPN is a mathematical notation in which every operator follows all of its operands. The calculator evaluates expressions by using a stack data structure.

### **Source Code:**

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <stdbool.h>
```

```
#define MAXOP 100
#define MAXVAL 100
#define BUFSIZE 100
#define NUMBER '0'
```

```
int sp = 0;
double val[MAXVAL];
```

```

int bufp = 0;
char buf[BUFSIZE];
int gettop(char[]);
void push(double);
double pop(void);
double printTop(void);
void duplicate(void);
void swapTop(void);
void clearStack(void);

double variables[26]; // Array to store variables (a-z)
double lastPrintedValue = 0.0;
int var = 0;

int main(void)
{
    int type;
    double op1, op2;
    char s[MAXOP];

    printf("Reverse Polish Notation (RPN) Calculator\n");
    printf("Enter numbers and operators (+, -, *, /, %, ~, !, $, &, =).\n");
    printf("Type 'q' to quit.\n");
    printf("Commands:\n");
    printf("  + : Addition\n");
    printf("  - : Subtraction\n");
    printf("  * : Multiplication\n");
    printf("  / : Division\n");
    printf("  % : Modulo\n");
    printf("  ~ : Print top of stack\n");
    printf("  ! : Duplicate top of stack\n");
    printf("  $ : Swap top two elements of stack\n");
    printf("  & : Clear the stack\n");
    printf("  = : Variable assignment\n");

    while ((type = gettop(s)) != 'q')
    {
        switch (type)
        {
            case NUMBER:
                push(atof(s));
                break;

            case '+':
                op1 = pop();
                op2 = pop();
                push(op1 + op2);

```

```

    printf("Result of %g + %g is: ", op1, op2);
    printTop();
    break;

case '-':
    op1 = pop();
    op2 = pop();
    push(op2 - op1);
    printf("Result of %g - %g is: ", op2, op1);
    printTop();
    break;

case '*':
    op1 = pop();
    op2 = pop();
    push(op2 * op1);
    printf("Result of %g * %g is: ", op1, op2);
    printTop();
    break;

case '/':
    op1 = pop();
    op2 = pop();

    if (op1 != 0.0)
    {
        push(op2 / op1);
        printf("Result of %g / %g is: ", op2, op1);
        printTop();
    }
    else
    {
        printf("Error: zero divisor.\n");
    }
    break;

case '%':
    op1 = pop();
    op2 = pop();

    if (op1 != 0.0)
    {
        int modded = (int)op2 % (int)op1;
        push(modded);
        printf("Result: ", modded);
        printTop();
    }
    else

```

```

{
    printf("Error: zero divisor.\n");
}

    break;

case '~':
    printf("Top of stack is: ");
    printTop();
    break;

case '!':
    duplicate();
    printf("Top 2 element duplicated!");
    break;

case '$':
    swapTop();
    printf("Top 2 element swaped!");
    break;

case '&':
    clearStack();
    printf("Stack Cleared!\n");
    printTop();
    break;

case '?':
    push(sin(pop()));
    printf("Result: ");
        printTop();
    break;

case ';':
    push(exp(pop()));
    printf("Result: ");
        printTop();
    break;

case '^':
    op2 = pop();
    push(pow(pop(), op2));
    printf("Result: ");
        printTop();
    break;

case '=': // Variable assignment
    variables[var - 'a'] = pop();
    push(variables[var - 'a']);

```

```

    printf("var %c : %g\n",var, variables[var-'a']);
    break;

case '\n':
    break;

default:
    if (isalpha(type))
    {
        var = type;
        push(variables[var - 'a']);
    }
    else
    {
        printf("error: unknown command %s\n", s);
    }
    break;
}
}

printf("Exiting the Stack Calculator.\n");

return 0;
}

```

```

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

```

```

double pop(void)
{
    if (sp > 0)
    {
        return val[--sp];
    }
    else
    {

```

```
    printf("Error: stack empty.\n");
    return 0.0;
}
}
```

```
double printTop(void)
{
    if (sp > 0)
    {
        printf("%.8g\n", val[sp - 1]);
        return val[sp - 1];
    }
    else
    {
        printf("Stack is empty.\n");
        return -1;
    }
}
```

```
void duplicate(void)
{
    if (sp > 0 && sp < MAXVAL)
    {
        val[sp] = val[sp - 1];
        sp++;
    }
    else if (sp >= MAXVAL)
    {
        printf("Error: stack full, can't duplicate.\n");
    }
    else
    {
        printf("Error: stack empty, can't duplicate.\n");
    }
}
```

```
void swapTop(void)
{
    if (sp > 1)
    {
        double temp = val[sp - 1];
        val[sp - 1] = val[sp - 2];
        val[sp - 2] = temp;
    }
    else if (sp == 1)
    {
        printf("Error: only one element in the stack, can't swap.\n");
    }
}
```

```

else
{
    printf("Error: stack empty, can't swap.\n");
}
}

```

```

void clearStack(void)
{
    sp = 0;
    printf("Stack cleared.\n");
}

```

```

int getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

```

```

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

```

```

int gettop(char s[])
{
    int i, c;

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

    s[1] = '\0';

    if (!isdigit(c) && c != '.' && c != '-')
        return c;

    if (c == '-')
    {
        int next = getch();

        if (isdigit(next) || next == '.')
        {

```

```

s[1] = next; // Store the '-' in the array
i = 1;

while (isdigit(s[++i] = c = getch()))
    ;

if (c == '.')
{
    while (isdigit(s[++i] = c = getch()))
        ;
}

s[i] = '\0';

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

return NUMBER;
}
else
{
    if (next != EOF)
        ungetch(next); // Push back the non-digit character

    return c; // Return '-' as an operator
}
}

i = 0;

if (isdigit(c))
{
    while (isdigit(s[++i] = c = getch()))
        ;

    if (c == '.')
    {
        while (isdigit(s[++i] = c = getch()))
            ;
    }

    s[i] = '\0';

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

    return NUMBER;
}

```



```
return c; // Return the non-digit character as an operator
}
```

**Explanation:**

1. The code implements a stack calculator program that performs arithmetic operations and manipulates a stack of values.
2. It uses a value stack (val) to store operands and a stack pointer (sp) to keep track of the next free position in the stack.
3. The program reads user input and processes it until the user enters 'q' to quit.
4. It parses the input using the `gettop()` function, which identifies whether the input is a number, operator, or variable.
5. Arithmetic operations like addition, subtraction, multiplication, division, and modulus are performed using the `pop()` function to retrieve operands from the stack and the `push()` function to store the result back onto the stack.
6. Utility functions like `printTop()`, `duplicate()`, `swapTop()`, and `clearStack()` provide operations to print the top value of the stack, duplicate the top element, swap the top two elements, and clear the stack, respectively.
7. Trigonometric and exponential functions are applied to the top value of the stack, and the result is pushed back onto the stack.
8. The program allows for variable assignment using the '=' operator, storing the value in the variables array indexed by alphabets 'a' to 'z'.
9. Alphabetic characters are treated as variables, and their corresponding values from the variables array are pushed onto the stack.
10. The program provides error handling for stack overflow, stack underflow, division by zero, and unknown commands or operands.

The push function adds an element to the stack, the pop function removes and returns the top element of the stack, and the `getch` and `ungetch` functions are used to handle input characters.

Ultimately I can say that, this code implements a basic RPN calculator with support for arithmetic operations, stack manipulation, variable storage, and printing results. The calculator evaluates expressions by manipulating a stack data structure.

**Output:**

```
D:\Day_Wise_Files\Day_9_r\Ex_4-6.exe
Reverse Polish Notation (RPN) Calculator
Enter numbers and operators (+, -, *, /, , ~, !, $, &, =).
Type 'q' to quit.
Commands:
+ : Addition
- : Subtraction
* : Multiplication
/ : Division
: Modulo
~ : Print top of stack
! : Duplicate top of stack
$ : Swap top two elements of stack
& : Clear the stack
= : Variable assignment
a 5 = b 6 = c 7 =
var a : 5
var b : 6
var c : 7
a b + c *
Result of 6 + 5 is: 11
Result of 7 * 11 is: 77
a b * c %
Result of 6 * 5 is: 30
Result: 2
_
```

#### Exercise 4-7

##### Question:

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`?

Let us look at the following code for better understanding of the answer to this question:

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

```
#define BUFSIZE 100
```

```
char buf[BUFSIZE]; // Buffer to hold characters
int bufp = 0;      // Next free position in buffer
```

```
int getch(void);
void ungetch(int c);
void ungets(char s[]);
```

```

int main(void) {
    char a[] = "Hello, world!"; // Push the string "Hello, world!" back onto the input
    ungets(a);
    printf("%s", buf);
    return 0;
}

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

void ungets(char s[]) {
    int i = strlen(s)-1;

    while (i >= 0) {
        ungetch(s[i]);
        --i;
    }
}

```

#### **Oversights of the given code:**

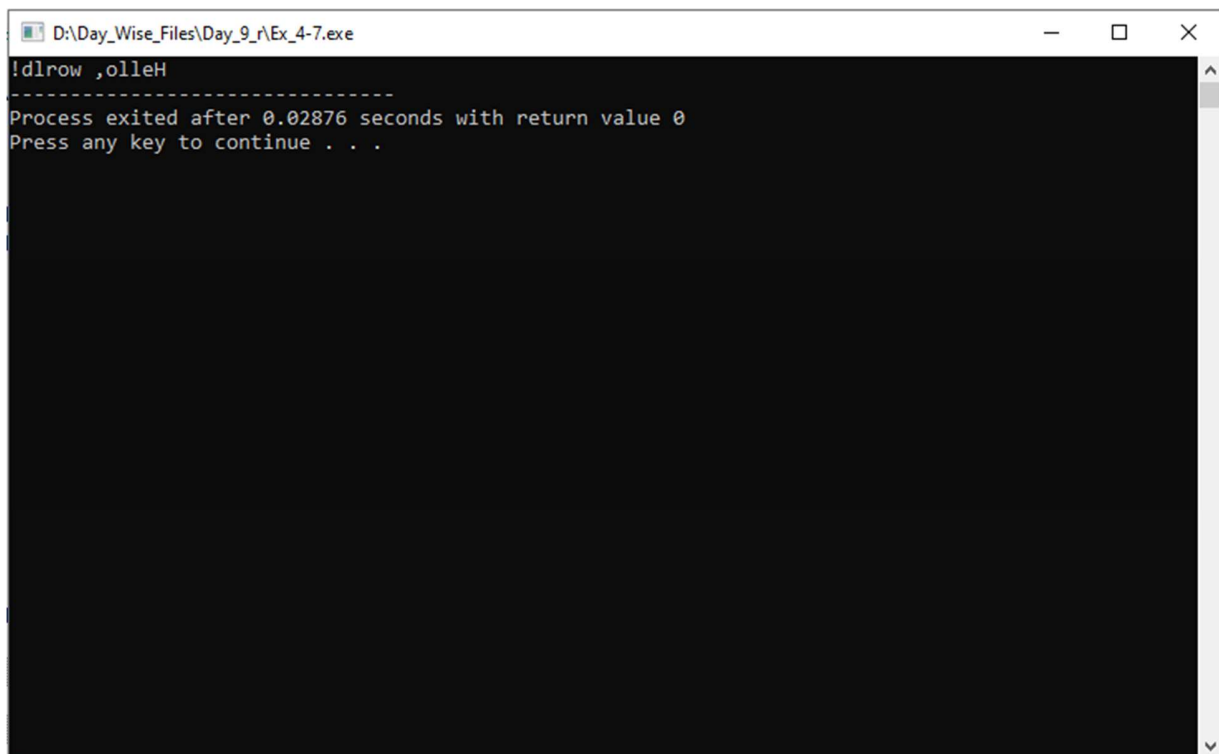
- The code includes necessary header files for standard input/output and string manipulation.
- It defines a constant BUFSIZE to specify the size of the buffer, which is used to hold characters.
- The variables buf and bufp are declared. buf is an array that serves as the buffer, and bufp keeps track of the next free position in the buffer.
- The getch() function is defined, which retrieves a character from the buffer if available (bufp > 0), or reads a character from standard input using getchar().
- The ungetch() function is defined, which pushes a character back into the buffer if there is space available (bufp < BUFSIZE).
- The ungets() function is defined, which pushes an entire string onto the input by calling ungetch() for each character in reverse order.
- In the main() function, a string a with the value "Hello, world!" is declared.
- The ungets() function is called with the string a as an argument, pushing the characters of the string back into the input buffer.
- Finally, the contents of the buffer buf are printed using printf(), which should display the string "Hello, world!".

#### **Observations:**

- The ungets() function takes a string s as input and pushes each character of the string back onto the input using ungetch(). It iterates over the characters in backward direction of the string until it reaches to the first character.

- By using `ungetch()` to push each character back onto the input, `ungets()` doesn't need to know about the specific details of the `buf` and `bufp` variables. It leverages the existing functionality of `ungetch()` to handle pushing characters back onto the input stream.
- This approach provides a more modular and encapsulated implementation, allowing `ungets()` to be used independently without relying on internal details of the `getch()` and `ungetch()` functions.

### **Output:**



```

D:\Day_Wise_Files\Day_9_r\Ex_4-7.exe
!dlrow ,olleH
-----
Process exited after 0.02876 seconds with return value 0
Press any key to continue . . .

```

- The string "Hello, world!" is assigned to the character array `a`.
- The `ungets()` function is called with the string `a` as an argument. The `ungets()` function pushes each character of the string back into the input buffer in reverse order. In this case, it pushes the characters '!', 'd', 'l', 'r', 'o', 'w', ' ', ' ', ' ', 'o', 'l', 'l', 'e', 'H'.
- After the `ungets()` function is executed, the buffer `buf` contains the characters pushed back from the string. So, the contents of `buf` are "Hello, world!".
- The `printf()` function is used to print the contents of `buf` using the `%s` format specifier. It will print the string "Hello, world!".