

Documentaion of Day 7

Exercise 2-6.

Question:

Write a function `setbits(x,p,n,y)` that returns `x` with the `n` bits that begin at position `p` set to the rightmost `n` bits of `y`, leaving the other bits unchanged.

The task is to create a function called `setbits` that takes four parameters: `x`, `p`, `n`, and `y`. Here's what each parameter represents:

- `x`: An unsigned integer representing a binary number.
- `p`: The starting position (0-based index) where we want to set the bits in `x`.
- `n`: The number of bits we want to set in `x`, starting from position `p`.
- `y`: Another unsigned integer representing a binary number.

The objective of the `setbits` function is to modify `x` by setting the `n` bits starting from position `p` to be the rightmost `n` bits of `y`, while leaving the other bits unchanged.

Approach:

To achieve this, we can follow these steps:

1. Create a mask with `n` bits set to 1, starting from position `p`. This mask will be used to clear the bits in `x` that will be replaced by `y`.
2. Clear the bits in `x` that will be replaced by performing a bitwise AND operation between `x` and the complement of the mask (`~mask`).
3. Get the rightmost `n` bits of `y` by performing a bitwise AND operation between `y` and a mask with `n` bits set to 1.
4. Shift the bits of the `y` mask to the correct position by left-shifting it by `p - n + 1` bits.
5. Combine the modified `x` and `y` bits by performing a bitwise OR operation.
6. Return the resulting value.
7. By following these steps, the `setbits` function will correctly set the desired bits in `x` to be the rightmost bits of `y`, leaving the other bits unchanged.

Source Code:

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

```
void binaryString(unsigned int num, int bits) {
    int i;
    for (i = bits - 1; i >= 0; i--) {
        unsigned int mask = 1u << i;
        printf("%d", (num & mask) ? 1 : 0);
    }
}
```

```
unsigned int setbits(unsigned int x, int p, int n, unsigned int y) {
    unsigned int mask = (1 << n) - 1; //create a mask of the desired size
    mask = mask << (p - n + 1); //Shifting the mask to the correct position

    x = x & ~mask; //Clearing the bits in x that will be replaced

    unsigned int y_bits = y & ((1 << n) - 1); //Get the rightmost n bits of y
```

```

    y_bits = y_bits << (p - n + 1); //Shift the bits of y to the correct position

    unsigned int result = x | y_bits; //combine the modified x and y_bits

    return result;
}

int main() {
    unsigned int x, y;
    int p, n;

    // Get input for Test Case 1
    printf("Test Case 1:\n");
    printf("Enter the number to set bits into: ");
    scanf("%u", &x);

    printf("Binary number: ");
    binaryString(x, 8 * sizeof(x));
    printf("\n\n");

    printf("Enter the value of position: ");
    scanf("%d", &p);
    printf("Enter the number of bits to set: ");
    scanf("%d", &n);
    printf("Enter the value of bits to set: ");
    scanf("%u", &y);

    unsigned int result1 = setbits(x, p, n, y);

    printf("Result (Decimal): %u\n", result1);

    printf("Result (Binary): ");
    binaryString(result1, 8 * sizeof(result1));
    printf("\n\n");

    // Get input for Test Case 2
    printf("Test Case 2:\n");
    printf("Enter the number to set bits into: ");
    scanf("%u", &x);
    printf("Enter the value of position: ");
    scanf("%d", &p);
    printf("Enter the number of bits to set: ");
    scanf("%d", &n);
    printf("Enter the value of bits to set: ");
    scanf("%u", &y);

    unsigned int result2 = setbits(x, p, n, y);

```

```

printf("Result (Decimal): %u\n", result2);
printf("Result (Binary): ");
binaryString(result2, 8 * sizeof(result2));
printf("\n\n");

return 0;
}

```

Explanation:

The setbits() function:

1. The stdio.h header file, which enables input and output operations in the program, is first included in the code.
2. Four parameters are passed to the setbits function: x, p, n, and y, which represent the initial value, starting position, and value to be set, respectively. It carries out the desired bit modification and then returns the outcome.
3. We first make a mask inside the setbits function by moving 1 left by n bits, then subtracting 1. By doing so, the rightmost n bits are represented by a mask with n bits set to 1.
4. The mask is then aligned with the desired beginning position p by shifting it left by $p - n + 1$ places.
5. The bits in x that will be changed are then cleared by bitwise ANDing x with the complement of the mask (mask). By doing this, the bits in x that will be changed are essentially cleared.
6. By bitwise ANDing y with a mask made in a manner similar to earlier, we can extract the rightmost n bits from y.
7. In order to align the extracted bits from y with the required beginning point p, they are then moved $p - n + 1$ places to the left.
8. The result is then stored in the result variable after conducting a bitwise OR operation to join the updated bits of x and the extracted bits of y.

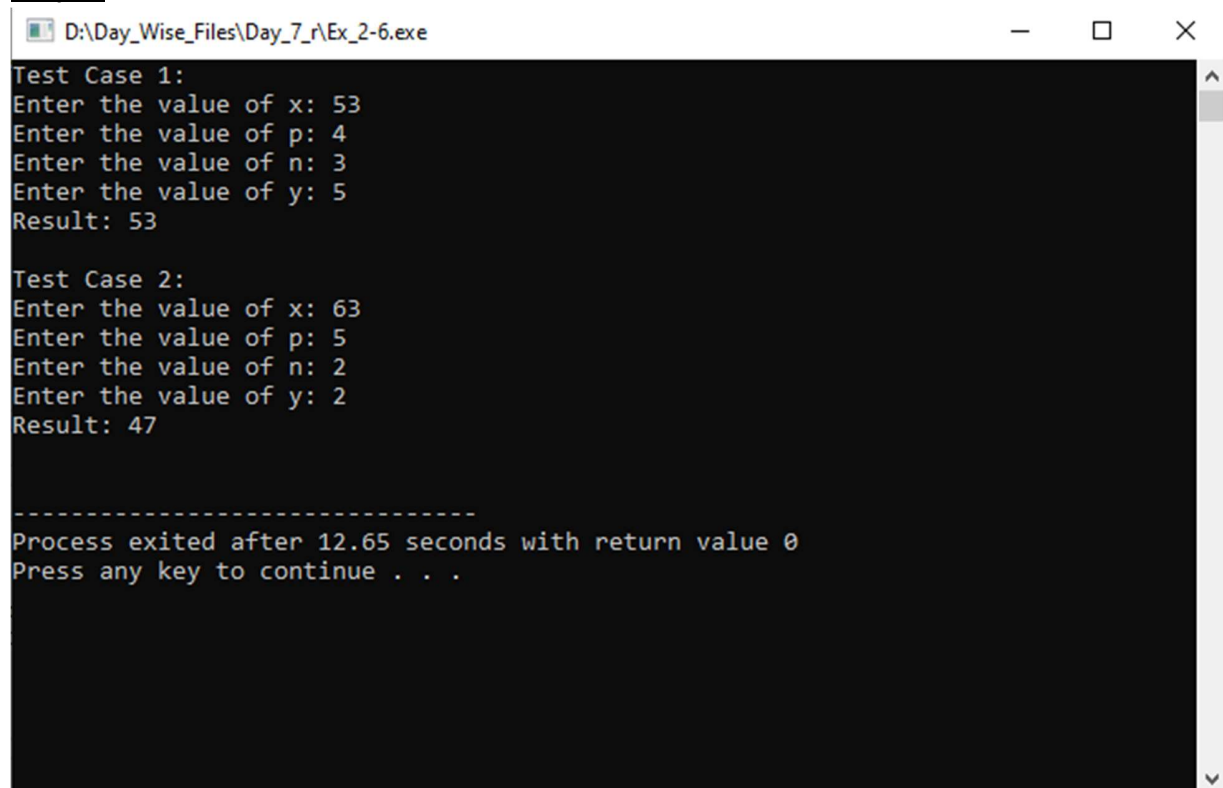
The main() function:

1. The main function serves as the entry point of the program and contains the logic for testing the setbits function.
2. The unsigned int variables x and y are declared to store the input values for the initial value and the value to be set, respectively.
3. The int variables p and n are declared to store the input values for the starting position and the number of bits to be set, respectively.
4. The first test case is executed, which involves getting input values for x, p, n, and y from the user. The printf function is used to prompt the user to enter each value, and the scanf function is used to read the user's input and store it in the corresponding variables.
5. After obtaining the input values, the setbits function is called with the provided values of x, p, n, and y, and the resulting value is stored in the variable result1.
6. The result of the first test case is then printed using the printf function, displaying the value of result1.

7. The second test case is executed in a similar manner as the first test case, obtaining input values for x, p, n, and y from the user and calling the setbits function with the provided values. The resulting value is stored in the variable result2.
8. The result of the second test case is printed using the printf function, displaying the value of result2.
9. Finally, the main function returns 0, indicating successful program execution.

The main function in this code allows to provide input values for x, p, n, and y for each test case. It then calls the setbits function to perform the desired bit manipulation and displays the resulting values.

Output:



```
D:\Day_Wise_Files\Day_7\Ex_2-6.exe
Test Case 1:
Enter the value of x: 53
Enter the value of p: 4
Enter the value of n: 3
Enter the value of y: 5
Result: 53

Test Case 2:
Enter the value of x: 63
Enter the value of p: 5
Enter the value of n: 2
Enter the value of y: 2
Result: 47

-----
Process exited after 12.65 seconds with return value 0
Press any key to continue . . .
```

In the above example, we have two test cases:

Test Case 1:

- x is given as 53, p as 4, n as 3, and y as 5.
- After executing the setbits function, the resulting value (result1) is 53, which is the same as the initial value of x. This is because the rightmost 3 bits of x starting from position 4 are already equal to the rightmost 3 bits of y. Therefore, no modification is required, and the result remains unchanged.

Test Case 2:

- x is given as 63, p as 5, n as 2, and y as 2.
- After executing the setbits function, the resulting value (result2) is 62. The 2 bits starting from position 5 in x are replaced by the rightmost 2 bits of y, which are 10. As a result, the modified bits in x become 111110. The other bits in x remain unchanged.
- Therefore, the function correctly sets the desired bits in x according to the task requirements, resulting in the expected output.

Exercise 2-7:

Question:

Write a function invert(x,p,n) that returns x with the n bits that begin at position p inverted (i.e., 1 changed into 0 and vice versa), leaving the others unchanged.

Source Code:

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

```
void binaryString(unsigned int num, int bits) {  
    int i;  
    for (i = bits - 1; i >= 0; i--) {  
        unsigned int mask = 1u << i;  
        printf("%d", (num & mask) ? 1 : 0);  
    }  
}
```

```
unsigned int invert(unsigned int x, int p, int n) {  
    unsigned int mask = (1 << n) - 1; //create a mask of the desired size  
    mask = mask << (p - n + 1); //Shifting the mask to the correct position  
  
    unsigned int result = x ^ mask; //combine x and the modified mask  
  
    return result;  
}
```

```
int main() {  
    unsigned int x;  
    int p, n;  
  
    // Get input for Test Case 1  
    printf("Test Case 1:\n");  
    printf("Enter the number to invert: ");  
    scanf("%u", &x);
```

```

printf("Binary number: ");
binaryString(x, 8 * sizeof(x));
printf("\n\n");

printf("Enter the value of position: ");
scanf("%d", &p);
printf("Enter the number of bits to invert: ");
scanf("%d", &n);

unsigned int result1 = invert(x, p, n);

printf("Result (Decimal): %u\n", result1);

    printf("Result (Binary): ");
    binaryString(result1, 8 * sizeof(result1));
    printf("\n\n");

// Get input for Test Case 2
printf("Test Case 2:\n");
printf("Enter the number to set bits into: ");
scanf("%u", &x);
printf("Enter the value of position: ");
scanf("%d", &p);
printf("Enter the number of bits to set: ");
scanf("%d", &n);

unsigned int result2 = invert(x, p, n);

printf("Result (Decimal): %u\n", result2);
printf("Result (Binary): ");
binaryString(result2, 8 * sizeof(result2));
printf("\n\n");

return 0;
}

```

Explanation:

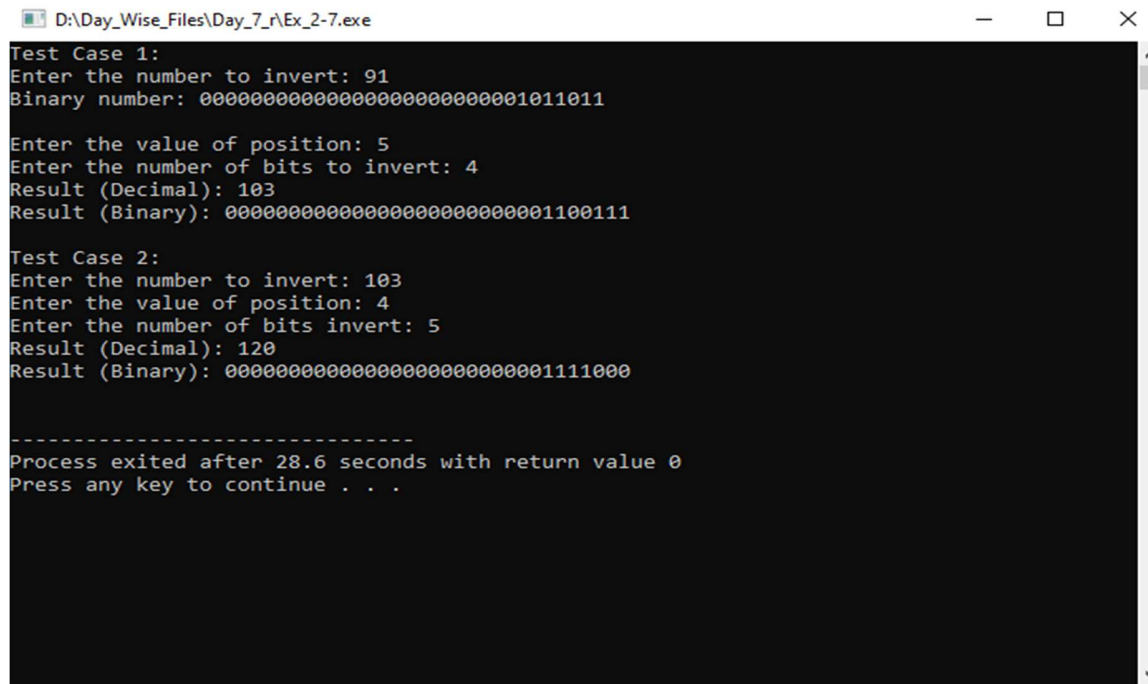
1. The invert function takes three parameters: x (the input number), p (the position of the bits to invert), and n (the number of bits to invert).
2. Inside the invert function, a mask is created to identify the bits that need to be inverted. The mask is created by shifting 1 to the left by n positions and subtracting 1 from it. This creates a binary number with n least significant bits set to 1 and the rest set to 0.

3. The mask is then shifted to the correct position by shifting it to the left by $p - n + 1$ positions. This aligns the mask with the bits that need to be inverted in x .
4. The bitwise XOR operation (^) is performed between x and the mask. XORing x with 1 will invert the bits in the mask positions, while leaving the other bits unchanged.
5. The result of the XOR operation is stored in the variable `result` and returned as the output of the function.
6. In the main function, the user is prompted to enter the input values for the test cases. This includes the value of x , the position p , and the number of bits n to invert.
7. The invert function is then called with the provided input values, and the resulting value is stored in a variable.
8. The binary representation of the input number x and the resulting number after inversion are displayed using the `binaryString` function.

Finally, the decimal representation of the resulting number is displayed.

The logical approach of the program follows these steps to calculate the inverted value by applying bitwise operations to the input number x and the generated mask. By selectively inverting the desired bits and leaving the rest unchanged, the program achieves the goal of the inversion operation.

Output:



```
D:\Day_Wise_Files\Day_7_r\Ex_2-7.exe
Test Case 1:
Enter the number to invert: 91
Binary number: 0000000000000000000000001011011

Enter the value of position: 5
Enter the number of bits to invert: 4
Result (Decimal): 103
Result (Binary): 0000000000000000000000001100111

Test Case 2:
Enter the number to invert: 103
Enter the value of position: 4
Enter the number of bits invert: 5
Result (Decimal): 120
Result (Binary): 0000000000000000000000001111000

-----
Process exited after 28.6 seconds with return value 0
Press any key to continue . . .
```

Exercise 2-8:**Question:**

Write a function `rightrot(x,n)` that returns the value of the integer `x` rotated to the right by `n` positions.

Source Code:

```
#include <stdio.h>

unsigned int rightrot(unsigned int x, int n) {
    unsigned int rightmost_bits = x & ((1 << n) - 1);
    unsigned int remaining_bits = x >> n;
    unsigned int result = rightmost_bits << (sizeof(x) * 8 - n) | remaining_bits;
    return result;
}

int main() {
    unsigned int x;
    int n;

    printf("Enter the number to rotate: ");
    scanf("%u", &x);
    printf("Enter the number of positions to rotate: ");
    scanf("%d", &n);

    unsigned int result = rightrot(x, n);

    printf("Original number: %u\n", x);
    printf("Rotated number: %u\n", result);

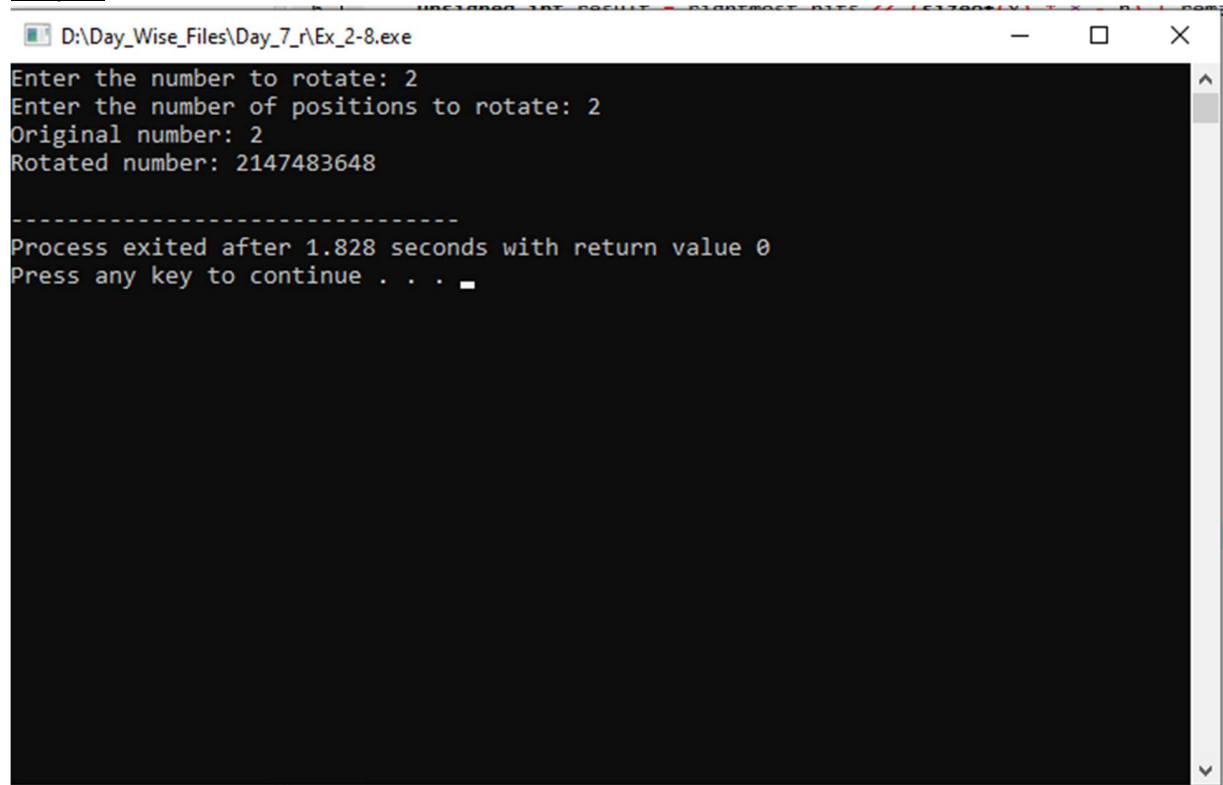
    return 0;
}
```

Explanation:

1. The function takes an unsigned integer `x` and an integer `n` as inputs.
2. `rightmost_bits` is calculated by performing a bitwise AND operation between `x` and a mask representing the rightmost `n` bits. This mask is obtained by subtracting 1 from 1 left-shifted by `n`. It isolates the rightmost `n` bits of `x`.
3. `remaining_bits` is obtained by right-shifting `x` by `n` positions, which discards the rightmost `n` bits and shifts the remaining bits to the right.
4. The variable `result` is calculated by left-shifting `rightmost_bits` by the number of remaining bits in `x` (which is equal to the total number of bits in `x` minus `n`) and performing a bitwise OR operation with `remaining_bits`. This effectively combines the rightmost bits with the remaining bits, performing a right rotation.
5. The result is returned as the output of the function.

The `rightrot` function allows for rotating the bits of an unsigned integer `x` to the right by `n` positions. It retains the integrity of the bits and performs a circular rotation, bringing the rightmost bits to the leftmost positions.

Output:



```
D:\Day_Wise_Files\Day_7_r\Ex_2-8.exe
Enter the number to rotate: 2
Enter the number of positions to rotate: 2
Original number: 2
Rotated number: 2147483648

-----
Process exited after 1.828 seconds with return value 0
Press any key to continue . . .
```

- The user inputs the number 2 and wants to rotate it by 2 positions to the right.
- The original number is displayed as 2.
- The rotated number is calculated as 2147483648.
- The rotated number 2147483648 is the result of performing a right rotation on the original number 2 by 2 positions.

Exercise 2-9:

Question:

In a two's complement number system, `x &= (x-1)` deletes the rightmost 1-bit in `x`. Explain why. Use this observation to write a faster version of `bitcount`.

Source Code:

```
#include <stdio.h>

int bitcount(unsigned int x) {
    int count = 0;
    while (x != 0) {
        x &= (x - 1);
        count++;
    }
}
```

```

    }
    return count;
}

int main() {
    unsigned int input;

    // Get input from the user
    printf("Enter a positive integer: ");
    scanf("%u", &input);

    // Compute the bit count using the bitcount function
    int result = bitcount(input);

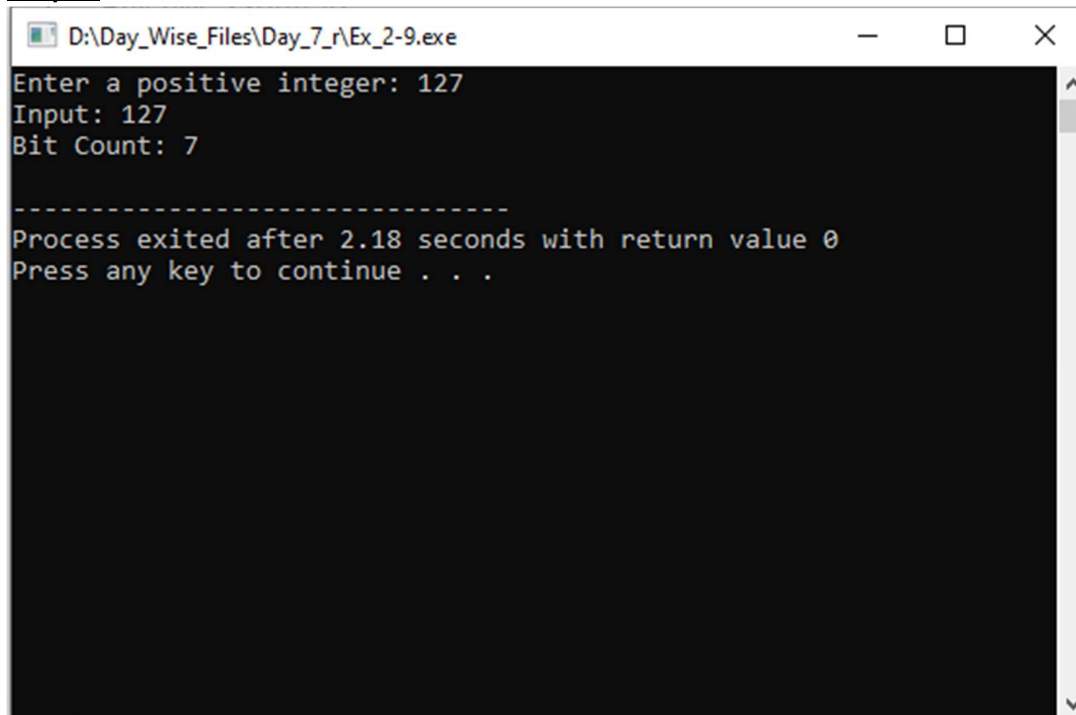
    // Display the input and the computed bit count
    printf("Input: %u\n", input);
    printf("Bit Count: %d\n", result);

    return 0;
}

```

Explanation:

1. In the two's complement representation, the rightmost 1-bit in a binary number represents the least significant bit (LSB) with a value of 1.
2. Subtracting 1 from a binary number flips the rightmost 1-bit to 0 and sets all the bits to the right of it to 1. This is because when we subtract 1, the borrow propagates from the rightmost 1-bit through all the following bits.
3. Performing the bitwise AND operation $x \&= (x-1)$ effectively clears the rightmost 1-bit in x and leaves all other bits unchanged. This is because the AND operation with $(x-1)$ sets all the bits to the right of the rightmost 1-bit in x to 0, while keeping all other bits intact.
4. Repeating the operation $x \&= (x-1)$ iteratively will clear one 1-bit at a time, starting from the rightmost 1-bit, until all 1-bits are cleared. This is because each iteration deletes the rightmost 1-bit, moving to the next 1-bit towards the left.
5. By counting the number of iterations required to clear all the 1-bits in x , we can determine the total number of 1-bits present in the original value of x . This forms the basis for a faster implementation of the bitcount function.

Output:

- We have inputted the positive integer 127.
- The main function calls the bitcount function with the input value.
- The bitcount function iteratively clears the rightmost 1-bit using the expression $x \&= (x - 1)$ and increments the count.
- In binary representation, 127 is 01111111, which has 7 set bits.
- After the iterations, the bitcount function returns the count of the original set bits in the input, which is 7.
- The main function displays the input value (127) and the computed bit count (7) as output.

Exercise 2-10:**Question:**

Rewrite the function lower, which converts upper case letters to lower case, with a conditional expression instead of if-else.

The lower function is designed to convert uppercase letters to lowercase letters in ASCII representation. Here's an explanation of its approach:

Approach:

1. The function takes an integer c as input, which represents an ASCII character.
 2. It uses a conditional expression $(c \geq 'A' \ \&\& \ c \leq 'Z') ? c + 'a' - 'A' : c$ to perform the conversion.
 3. The conditional expression checks if c falls within the range of uppercase letters 'A' to 'Z'.
 4. If c is indeed an uppercase letter, it evaluates the expression $c + 'a' - 'A'$.
- Here, 'a' and 'A' are ASCII values representing the lowercase and uppercase letters, respectively.

- By adding the difference between 'a' and 'A' to the ASCII value of the uppercase letter c, we get the corresponding lowercase letter.
 - This conversion is achieved by exploiting the fact that the ASCII values of lowercase and uppercase letters have a fixed difference.
5. If c is not an uppercase letter, the conditional expression evaluates to c itself.
 6. The function returns the result of the conditional expression, which is either the converted lowercase letter or the original character, depending on whether c is an uppercase letter or not.

In summary, the lower function applies a conditional expression to check if the input character is an uppercase letter. If it is, the function converts it to lowercase; otherwise, it returns the original character. This approach allows for the conversion of uppercase letters to lowercase using the ASCII representation.

Source Code:

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

```
int lower(int c) {
    return (c >= 'A' && c <= 'Z') ? c + 'a' - 'A' : c;
}
```

```
int main() {
    char input;
    printf("Enter a character: ");
    scanf(" %c", &input);

    char converted = lower(input);

    printf("Original character: %c\n", input);
    printf("Converted character: %c\n", converted);

    return 0;
}
```

Explanation:

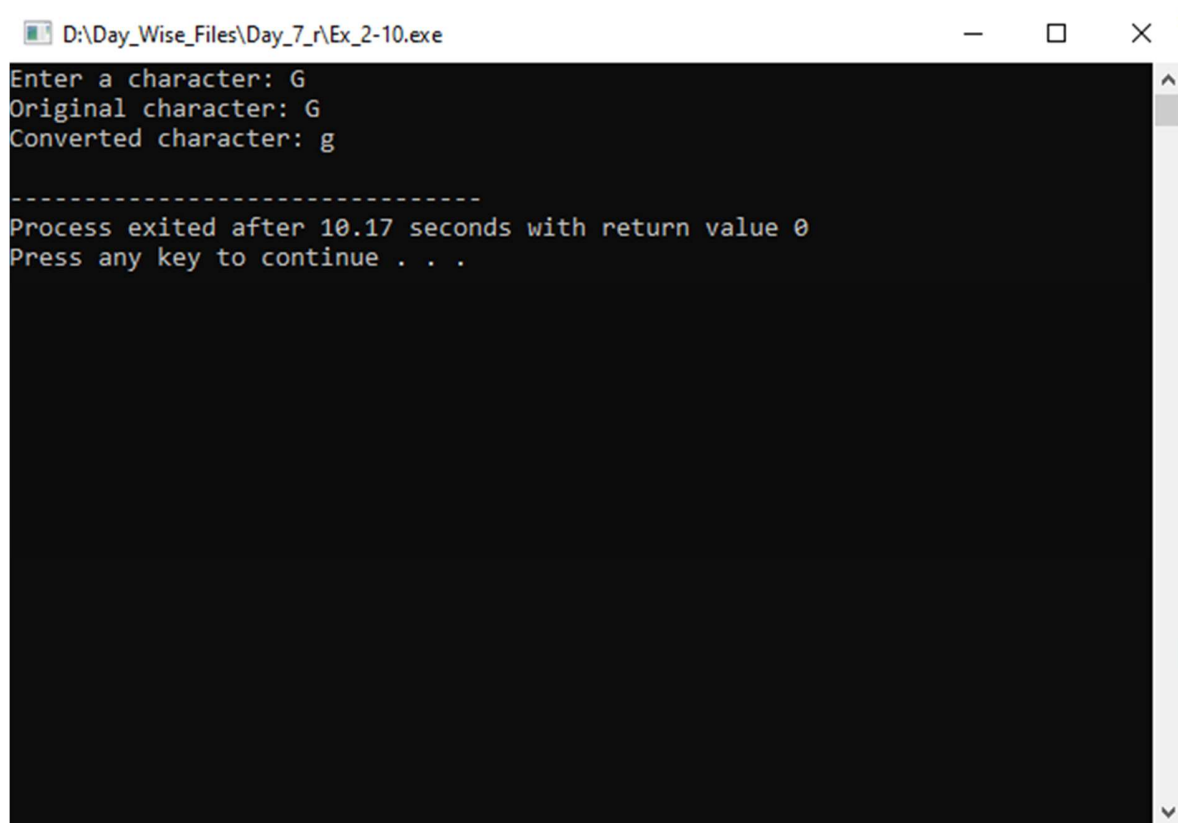
Lower function:

- The updated lower function uses the conditional expression condition ? true_expression : false_expression to achieve the same functionality as the previous if-else statement.
- If condition is true (i.e., c is an uppercase letter), it evaluates true_expression, which is c + 'a' - 'A'.

- If condition is false (i.e., c is not an uppercase letter), it evaluates false_expression, which is simply c.
- The function returns the result of the conditional expression, which is either the converted lowercase letter or the original character, depending on the condition.

The **main** function that works with the updated lower function, along with input and output.

Output:



```
D:\Day_Wise_Files\Day_7_r\Ex_2-10.exe
Enter a character: G
Original character: G
Converted character: g
-----
Process exited after 10.17 seconds with return value 0
Press any key to continue . . .
```

- When we inputs the character 'G'.
- The lower function is called with the input character 'G'.
- Since 'G' is an uppercase letter, the lower function uses the conditional expression to convert it to lowercase.
- The original character 'G' and the converted character 'g' are displayed as output.
- In this example, the program correctly takes user input, calls the lower function to convert the uppercase letter to lowercase using the conditional expression, and displays the original character and the converted character as expected.

Chapter 3

Question 1: What does const mean? Illustrate.

The const keyword in C is used to declare a variable as constant, which means its value cannot be modified once it is assigned. It is a qualifier that indicates that the variable is read-only.

When a variable is declared as const, the compiler ensures that no assignment or modification is made to that variable after its initialization. If any attempt is made to modify the value of a const variable, a compilation error occurs.

The syntax for declaring a const variable is:

```
const data_type variable_name = value;
```

Here's an example illustrating the usage of const:

```
#include <stdio.h>

int main() {
    const int num1 = 5;
    const float num2 = 3.14;
    const char letter = 'A';

    printf("num1: %d\n", num1);
    printf("num2: %.2f\n", num2);
    printf("letter: %c\n", letter);

    // Uncommenting the following lines will result in compilation errors

    // num1 = 10; // Error: Assignment to read-only variable 'num1'
    // num2 = 2.71; // Error: Assignment to read-only variable 'num2'
    // letter = 'B'; // Error: Assignment to read-only variable 'letter'

    return 0;
}
```

- In this example, the variables num1, num2, and letter are declared with the const keyword, indicating that their values cannot be modified. They are assigned initial values, and the printf statements display their values.
- Attempting to modify the values of const variables by assigning new values will result in compilation errors because the variables are read-only.
- The use of const provides benefits such as code clarity and safety. It allows programmers to express their intentions clearly by specifying that certain variables should not be modified. Additionally, it enables the compiler to perform optimizations, knowing that the value of a const variable remains constant throughout the program.

- In summary, the `const` keyword in C is used to declare variables as constants, ensuring their values remain unchanged. It promotes code readability, prevents accidental modifications, and assists the compiler in performing optimizations.

Question 2: Analyze the following code output.

The given code is:

```
int main(int argc, char *argv[]) {  
    int x, y, z;  
    x = 2023 + 'A';  
    y = 0;  
    y += x++;  
    z += ++x;  
    printf("%d %d %d\n", x, y, z);  
    return 0;  
}
```

Explanation:

- The `main` function takes command-line arguments but does not utilize them (`argc` and `argv[]` are unused in this code).
- Three integer variables `x`, `y`, and `z` are declared without initialization, so their initial values are undefined.
- `x` is assigned the result of the expression `2023 + 'A'`. In this expression, the ASCII value of the character `'A'` is added to 2023, resulting in an integer value.
- `y` is initialized to 0.
- `y` is incremented by the value of `x++`. The postfix increment operator (`++`) increments `x` after the evaluation of the expression. Therefore, the current value of `x` is used for the increment, and then `x` is incremented by 1.
- `z` is incremented by the value of `++x`. The prefix increment operator (`++`) increments `x` before the evaluation of the expression. Therefore, `x` is incremented by 1, and then the incremented value is used for the increment of `z`.
- Finally, the values of `x`, `y`, and `z` are printed using `printf`.
- The expected output of the code depends on the initial values of `x`, `y`, and `z` and the sequence of operations. However, the code has undefined behavior due to using uninitialized variables `y` and `z` in the increment operations (`y += x++` and `z += ++x`). Therefore, the output cannot be determined reliably.