1. Write a C program that declares an integer pointer, initializes it to point to an integer variable, and prints the value of the variable using the pointer.

```
#include <stdio.h>
int main()
{
   int a = 10;
   int *p;
   p = &a;
   printf("Value of a = %d\n", *p);
   return 0;
}
```

O/P:

```
Value of a = 10
```

2. Create a program where you declare a pointer to a float variable, assign a value to the variable, and then use the pointer to change the value of the float variable. Print both the original and modified values.

```
#include <stdio.h>
int main()
{
   float a = 1.1;
   float *p;
```

```
    p = &a;
    printf("Original value: %f\n", a);
    *p = 2.5;
    printf("Modified value: %f\n", a);
    return 0;
}
```

O/P:

```
        Original value: 1.100000
        Modified value: 2.500000
```

3. Given an array of integers, write a function that takes a pointer to the array and its size as arguments. Use pointer arithmetic to calculate and return the sum of all elements in the array.

```
#include <stdio.h>

int calculate_sum(int *a, int size); // Function prototype

int main()
{
    int a[5] = {10, 20, 30, 40, 50};
    int size = sizeof(a) / sizeof(a[0]);

    //Call the function to calculate the sum of all elements in the array
    int Total_sum = calculate_sum(a, size);
    printf("Sum of all elements in the array: %d\n", Total_sum);
    return 0;
}

/*
Name: sum()
Return Type: int
Parameter:(data type of each parameter): int* and int
Short description: it is used to add all the elements in tha array
*/

// Function to calculate the sum of all the elements in an array using pointer
arithmetic
int calculate_sum(int *a, int size)
{
    int sum = 0;
    int *p;
    for (p = a; p < a + size; p++)
        sum += *p;
    return sum;
}
```

O/P:

```
        Sum of all elements in the array: 150
```

4. Write a program that demonstrates the use of a null pointer. Declare a pointer, assign it a null value, and check if it is null before attempting to dereference it.

```
#include <stdio.h>
int main()
{
   int *p = 0; // Declare a pointer and assign it a null value

   /*int a = 10;
   int *p;
   p = &a;
   p = 0;*/

   // Check if the pointer is null
   if (p == 0)
      printf("Pointer is null and cannot be dereferenced\n");
   else
      printf("Value pointed to by p: %d\n", *p);
   return 0;
}
```

O/P:

```
        Pointer is null and cannot be dereferenced
```

5. Create an example that illustrates what happens when you attempt to dereference a wild pointer (a pointer that has not been initialized). Document the output and explain why this leads to undefined behaviour.

```
#include <stdio.h>
int main()
{
    int *p; // Declare a wild pointer
   printf("Attempting to dereference a wild pointer\n");
   printf("Value pointed to by p: %d\n", *p); // Leads to undefined behaviour
   return 0;
}
```

O/P:

```
Attempting to dereference a wild pointer
|
```

```
    /* This leads to undefined behaviour because *p accesses the memory
    at an undefined loaction*/
    printf("Value pointed to by p: %d\n", *p); // Segmentation fault
```

6. Implement a C program that uses a pointer to a pointer. Initialize an integer variable, create a pointer that points to it, and then create another pointer that points to the first pointer. Print the value using both levels of indirection.

```
#include <stdio.h>
int main()
{
    int b = 10;
    int *p = &b;
    int **q = &p;

    // Print the value using the first level of indirection
    printf("Value of b using p: %d\n", *p);

    // Print the value using the second level of indirection
    printf("Value of b using q: %d\n", **q);
    return 0;
}
```

O/P:

```
        Value of b using p: 10
        Value of b using q: 10
```

7. Write a program that dynamically allocates memory for an array of integers using malloc. Populate the array with values, print them using pointers, and then free the allocated memory.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int size = 5;

    // Dynamically allocate memory for the array
    int *a = (int *)malloc(size * sizeof(int));
    if (a == NULL)
    {
        printf("Memory allocation failed\n");
        return 1;
    }

    /*printf("Enter the elements of array:\n");
    for (int i = 0; i < size; i++)
        scanf("%d", &a[i]); */

    // Populate the array with values
    for (int i = 0; i < size; i++)
        a[i] = i + 1;

    // Print the values using pointers
    printf("Elements of the array:\n");
    for (int i = 0; i < size; i++)
        printf("%d ", *(a + i));
    printf("\n");

    free(a); // Free the allocated memory
    return 0;
}
```

O/P:

```
Elements of the array:
1 2 3 4 5
```

8. Define a function that takes two integers as parameters and returns their sum. Then, create a function pointer that points to this function and use it to call the function with different integer values.

```
#include <stdio.h>

int sum(int a, int b); // Function prototype

int main()
{
   // Declare a function pointer and point it to the 'add' function
   int (*p)(int, int) = sum;

   //Call the function using function pointer with different values
   int result1 = p(15, 8);
   int result2 = p(8, 18);
   printf("Sum = %d\n", result1);
   printf("Sum = %d\n", result2);
   return 0;
}

/*
Name: sum()
Return Type: int
Parameter:(data type of each parameter): int and int
Short description: it is used to add 2 integers
*/
```

```
// Function that takes two integers and returns their sum
int sum(int a, int b)
{
   return a + b;
}
```

O/P:

```
        Sum = 23
        Sum = 26
```

9. Write a program that compares two pointers pointing to different variables of the same type. Use relational operators to determine if one pointer points to an address greater than or less than another and print the results.

```
#include <stdio.h>
int main()
{
   int a = 10, b = 20;
   int *p = &a;
   int *q = &b;

   // Compare the pointers using relational operators
   printf("Address of a: %p\n", (void *)p);
   printf("Address of b: %p\n", (void *)q);

   if (p > q)
      printf("p points to a higher address than q\n");
```

```
    else if (p < q)
        printf("p points to a lower address than q\n");
    else
        printf("p and q point to the same address\n");
    return 0;
}
```

O/P:

```
        Address of a: 00000000005FFE8C
        Address of b: 00000000005FFE88
        p points to a higher address than q
```

10. Create two examples: one demonstrating a constant pointer (where you cannot change what it points to) and another demonstrating a pointer to constant data (where you cannot change the data being pointed to). Document your findings.

```
#include <stdio.h>
int main()
{
    // Ex1: Constant pointer (pointer cannot change what it points to)
    int a = 10, b = 20;
    int *const p = &a; // p is a constant pointer

    printf("Value pointed to by p: %d\n", *p);
    *p = 15;
    printf("Modified value pointed to by p: %d\n", *p);
    // p = &b; // This is not possible. Cannot change what p points to

    // Ex2: Pointer to constant data (data being pointed to cannot be changed)
    const int c = 30;
    const int *q = &c; // q is a pointer to constant data

    printf("Before: value pointed to by q: %d\n", *q);
    // *q = 40; // This is not possible. Cannot modify the value at the address
    int d = 50;
    q = &d; // Possible and can change what q points to
    printf("After: value pointed to by q: %d\n", *q);
    return 0;
}
```

O/P:

```
        Value pointed to by p: 10
        Modified value pointed to by p: 15
        Before: value pointed to by q: 30
        After: value pointed to by q: 50
```

1. Write a program that declares a constant pointer to an integer. Initialize it with the address of an integer variable and demonstrate that you can change the value of the integer but cannot reassign the pointer to point to another variable.

```
#include <stdio.h>
int main()
{
   int a = 10;
   int b = 20;

   // Declare a constant pointer to an integer
   int *const p = &a;

   // Demonstrate modifying the value of the integer
   printf("Original value of a: %d\n", *p);
   *p = 30; // Modification of the value at the address is possible
   printf("Modified value of a: %d\n", *p);

   // Reassign the pointer
   // p = &b; // Cannot change what p points to
   return 0;
}
```

O/P:

```
        Original value of a: 10
        Modified value of a: 30
```

2. Create a program that defines a pointer to a constant integer. Attempt to modify the value pointed to by this pointer and observe the compiler's response.

```
#include <stdio.h>
int main()
{
   const int b = 10;  // constant integer
   const int *ptr = &b;  // pointer to constant integer
   printf("Value of a: %d\n", *ptr);

   // Attempting to modify the value of b through the pointer
   *ptr = 20;  // Compilation error: assignment of read-only location '*ptr'
   printf("Modified value: %d\n", *ptr);
   return 0;
}
```

O/P:

```
assignment of read-only location '*ptr'  ( Compilation error)
```

3. Implement a program that declares a constant pointer to a constant integer. Show that neither the address stored in the pointer nor the value it points to can be changed.

```
#include <stdio.h>
int main()
{
   const int c = 10;
```

```
    const int *const ptr = &c; // Constant pointer to a constant integer
    printf("Original value: %d\n", *ptr);

    // Attempting to modify the value pointed to by ptr
    // *ptr = 20;  // Compilation error: assignment of read-only location '*ptr'

    // Attempting to modify the address stored in ptr
   // ptr = &num; // Compilation error: assignment of read-only variable 'ptr'
     return 0;
}
```

O/P:

```
        Original value: 10
```

4. Develop a program that uses a constant pointer to iterate over multiple integers stored in separate variables. Show how you can modify their values through dereferencing while keeping the pointer itself constant.

```
#include <stdio.h>
int main()
{
    int a = 10, b = 20, c = 30;
    int *const ptr = &a; // Constant pointer to an integer

    printf("Original values:\n");
    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);

    // Modifying values through dereferencing the constant pointer
    *ptr = 15;  // Modifies a
    printf("\nAfter modifying 'a' through ptr: %d\n", a);

     /*Changing the pointer to point to the next variable is not possible
     because ptr is a constant pointer it cannot be reassigned so a temporary
     pointer is used to iterate */
    int *temp_ptr = ptr;

    temp_ptr = &b;
    *temp_ptr = 50; // Modify b
    printf("After modifying 'b' through temp_ptr: %d\n", b);

    temp_ptr = &c;
    *temp_ptr = 40;   // Modify c
    printf("After modifying 'c' through temp_ptr: %d\n", c);

    // The constant pointer (ptr) was not changed during iteration
    printf("\nValues after iteration:\n");
    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);

    return 0;
}
```

O/P:

```
Original values:
a: 10
b: 20
c: 30

After modifying 'a' through ptr: 15
After modifying 'b' through temp_ptr: 50
After modifying 'c' through temp_ptr: 40

Values after iteration:
a: 15
b: 50
c: 40
```

5. Implement a program that uses pointers and decision-making statements to check if two constant integers are equal or not, printing an appropriate message based on the comparison.

```c
#include <stdio.h>
int main()
{
    const int a = 10;
    const int b = 10;

    // Pointers to the constant integers
    const int *p = &a;
    const int *q = &b;

    if (*p == *q)
        printf("Two constant integers are equal\n");
    else
        printf("Two constant integers are not equal\n");
    return 0;
}
```

O/P:

```
        Two constant integers are equal
```

6. Create a program that uses conditional statements to determine if a constant pointer is pointing to a specific value, printing messages based on whether it matches or not.

```
#include <stdio.h>
int main()
{
    int a = 10;

    // Constant pointer pointing to a
    int *const ptr = &a;

    if (*ptr == 10)
        printf("Constant pointer is pointing to the value of 'a'(a = 10)\n");
    else
```

```
        printf("Constant pointer is not pointing to the value of 'a'\n");
    return 0;
}
```

O/P:

```
    Constant pointer is pointing to the value of 'a'(a = 10)
```

7. Write a program that declares two constant pointers pointing to different integer variables. Compare their addresses using relational operators and print whether one points to a higher or lower address than the other.

```
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 20;

    // Constant pointers pointing to the integer variables
    int *const p = &a;
    int *const q = &b;

    printf("Address pointed to by p: %p\n", *p);
    printf("Address pointed to by q: %p\n", *q);

    // Comparing the addresses using relational operators
    if (p > q)
        printf("p points to a higher address than q\n");
    else if (p < q)
        printf("p points to a lower address than q\n");
    else
        printf("p and q point to the same address\n");
    return 0;
}
```

O/P:

```
Address pointed to by p: 000000000000000A
Address pointed to by q: 0000000000000014
p points to a higher address than q
```

8. Implement a program that uses a constant pointer within loops to iterate through multiple variables (not stored in arrays) and print their values.

```
#include <stdio.h>
int main()
{
    // Declare multiple variables
    int a = 10, b = 20, c = 30, d = 40;

    // Create an array of pointers pointing to these variables
    int *const p[] = {&a, &b, &c, &d};

    // Iterate through the variables using a loop and a constant pointer
    int i;
    for (i = 0; i < 4; i++)
    {
        int *const q = p[i]; // Constant pointer pointing to the current variable
        printf("Value of variable %d: %d\n", i + 1, *q);
    }
    return 0;
}
```

O/P:

```
        Value of variable 1: 10
        Value of variable 2: 20
        Value of variable 3: 30
        Value of variable 4: 40
```

9. Develop a program that uses a constant pointer to iterate over several integer variables (not in an array) using pointer arithmetic while keeping the pointer itself constant.

```
#include <stdio.h>
int main()
{
    // Declare multiple variables
     int a = 10, b = 20, c = 30, d = 40;

    // Create an array of addresses for the variables
    int *p[] = {&a, &b, &c, &d};
    int **ptr = p;  // Constant pointer to an array of integer pointers
    // Use pointer arithmetic to iterate over the variables
    int i;
    for (i = 0; i < 4; i++)
    {
        printf("Value of variable %d: %d\n", i + 1, **ptr);
        ptr++;  // Move to the next pointer in the array
    }
    return 0;
}
```

O/P:

```
        Value of variable 1: 10
        Value of variable 2: 20
        Value of variable 3: 30
        Value of variable 4: 40
```

1. Machine Efficiency Calculation

Requirements:

- Input: Machine's input power and output power as floats.
- Output: Efficiency as a float.
- Function: Accepts pointers to input power and output power, calculates efficiency, and updates the result via a pointer.
- Constraints: Efficiency = (Output Power / Input Power) * 100.

```
#include <stdio.h>

// Function prototype
void calculateEfficiency(float *inputPower, float *outputPower, float
*efficiency);

int main()
{
    float inputPower, outputPower, efficiency;

    printf("Enter the machine's input power: ");
    scanf("%f", &inputPower);
    printf("Enter the machine's output power: ");
    scanf("%f", &outputPower);

    // Call the function to calculate efficiency by call by reference
    calculate_efficiency(&inputPower, &outputPower, &efficiency);

    if (inputPower != 0)
        printf("Machine Efficiency: %.2f%%\n", efficiency);
    return 0;
}

/*
Name: calculate_efficiency()
Return Type: void
Parameter:(data type of each parameter): float*, float* and float*
Short description: it is used to calculate the efficiency of machine
*/

// Function to calculate machine efficiency
void calculate_efficiency(float *inputPower, float *outputPower, float *efficiency)
{
    // Check for division by zero
    if (*inputPower == 0)
    {
        printf("Input power cannot be zero.\n");
        *efficiency = 0;
        return;
    }
  *efficiency = (*outputPower / *inputPower) * 100; // Calculate efficiency
}
```

O/P:

```
Enter the machine's input power: 120
Enter the machine's output power: 100
Machine Efficiency: 83.33%
```

2. Conveyor Belt Speed Adjustment

Requirements:

- Input: Current speed (float) and adjustment value (float).
- Output: Updated speed.
- Function: Uses pointers to adjust the speed dynamically.
- Constraints: Ensure speed remains within the allowable range (0 to 100 units).

```
#include <stdio.h>

//Function prototype
void adjust_speed(float *currentSpeed, float adjustmentValue);

int main()
{
  float currentSpeed, adjustmentValue;

  // Input the current speed and adjustment value
  do
  {
    printf("Enter the current speed of the conveyor belt: ");
    scanf("%f", &currentSpeed);
    if (currentSpeed < 0 || currentSpeed > 100)
      printf("Current speed must be between 0 and 100 units\n");
  } while (currentSpeed < 0 || currentSpeed > 100);

  printf("Enter the adjustment value: ");
  scanf("%f", &adjustmentValue);
```

```
    // Call the function to adjust speed
    adjust_speed(&currentSpeed, adjustmentValue);

    printf("Updated conveyor belt speed: %.2f units\n", currentSpeed);

    return 0;
}

/*
Name: adjust_speed()
Return Type: void
Parameter:(data type of each parameter): float*, float* and float*
Short description: it is used to calculate the efficiency of machine
*/

// Function to adjust conveyor belt speed
void adjust_speed(float *currentSpeed, float adjustmentValue)
{
    // Update the speed dynamically
    *currentSpeed += adjustmentValue;

    if (*currentSpeed > 100.0)
        *currentSpeed = 100.0;
    else if (*currentSpeed < 0.0)
        *currentSpeed = 0.0;
}
```

O/P:

```
Enter the current speed of the conveyor belt: 120
Current speed must be between 0 and 100 units
Enter the current speed of the conveyor belt: 80
Enter the adjustment value: 20
Updated conveyor belt speed: 100.00 units
```

3. Inventory Management

Requirements:

- Input: Current inventory levels of raw materials (array of integers).
- Output: Updated inventory levels.
- Function: Accepts a pointer to the inventory array and modifies values based on production or consumption.
- Constraints: No inventory level should drop below zero.

```c
#include <stdio.h>

// Function prototype
void update_inventory(int *inventory, int *modify, int size);

int main()
{
    int size;

    // Input the number of raw materials and current inventory levels
    printf("Enter the number of raw materials: ");
    scanf("%d", &size);
    int inventory[size];
    int modify[size];

    printf("\nEnter the current inventory levels\n");
    for (int i = 0; i < size; i++)
    {
        printf("Raw material %d: ", i + 1);
        scanf("%d", &inventory[i]);
    }

    // Input the modifications in inventory due to production/consumption
     printf("\nEnter the modifications in inventory (+ for production, - for
consumption)\n");
    for (int i = 0; i < size; i++)
    {
        printf("Modification for raw material %d: ", i + 1);
        scanf("%d", &modify[i]);
    }

    // Call the function to update inventory levels
    update_inventory(inventory, modify, size);

    printf("\nUpdated inventory levels\n");
    for (int i = 0; i < size; i++)
        printf("Raw material %d: %d\n", i + 1, inventory[i]);

    return 0;
```

```
}

/*
Name: update_inventory()
Return Type: void
Parameter:(data type of each parameter): int*, int* and int
Short description: it is used to update the inventory levels
*/

// Function to update inventory levels
void update_inventory(int *inventory, int *modify, int size)
{
   int i;
   for (i = 0; i < size; i++)
   {
      inventory[i] += modify[i];
      if (inventory[i] < 0)
         inventory[i] = 0;
   }
}
```

O/P:

```
Enter the number of raw materials: 5

Enter the current inventory levels
Raw material 1: 20
Raw material 2: 50
Raw material 3: 46
Raw material 4: 78
Raw material 5: 90

Enter the modifications in inventory (+ for production, - for consumption)
Modification for raw material 1: 40
Modification for raw material 2: 20
Modification for raw material 3: -10
Modification for raw material 4: -12
Modification for raw material 5: -30

Updated inventory levels
Raw material 1: 60
Raw material 2: 70
Raw material 3: 36
Raw material 4: 66
Raw material 5: 60
```

4. Robotic Arm Positioning

Requirements:

- Input: Current x, y, z coordinates (integers) and movement delta values.
- Output: Updated coordinates.
- Function: Takes pointers to x, y, z and updates them based on delta values.
- Constraints: Validate that the coordinates stay within the workspace boundaries.

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

```
#define X_MIN 0
#define X_MAX 10
#define Y_MIN 0
#define Y_MAX 10
#define Z_MIN 0
#define Z_MAX 10

//Function prototype
void update_position(int *x, int *y, int *z, int dx, int dy, int dz);

int main()
{
   int x, y, z, dx, dy, dz;

   // Input the coordinates
   printf("Enter current coordinates of x y and z: ");
   scanf("%d %d %d", &x, &y, &z);

   // Validating the entered coordinates
   if (x < X_MIN || x > X_MAX || y < Y_MIN || y > Y_MAX || z < Z_MIN
|| z > Z_MAX)
   {
      printf("Coordinates must be within the workspace boundaries\n");
      return 1;
   }

   // Input the delta values for movement
```

```
    printf("Enter movement delta values for x y and z: ");
    scanf("%d %d %d", &dx, &dy, &dz);

    // Call the function to update the position
    update_position(&x, &y, &z, dx, dy, dz);

    printf("Updated position: x = %d, y = %d, z = %d\n", x, y, z);

    return 0;
}

/*
Name: update_position()
Return Type: void
Parameter:(data type of each parameter): int*, int*, int*, int, int and int
Short description: it is used to update the position of robotic arm
*/

// Function to update robotic arm position
void update_position(int *x, int *y, int *z, int dx, int dy, int dz)
{
    // Update the coordinates based on delta values
    *x += dx;
    *y += dy;
    *z += dz;

    // Ensure the coordinates stay within the workspace boundaries
    if (*x < X_MIN)
        *x = X_MIN;
    if (*x > X_MAX)
        *x = X_MAX;
    if (*y < Y_MIN)
        *y = Y_MIN;
    if (*y > Y_MAX)
        *y = Y_MAX;
    if (*z < Z_MIN)
        *z = Z_MIN;
    if (*z > Z_MAX)
        *z = Z_MAX;
}
```

O/P:

```
Enter current coordinates of x y and z: 10 8 5
Enter movement delta values for x y and z: 4 9 4
Updated position: x = 10, y = 10, z = 9
```

5. Temperature Control in Furnace

Requirements:

- Input: Current temperature (float) and desired range.
- Output: Adjusted temperature.
- Function: Uses pointers to adjust temperature within the range.
- Constraints: Temperature adjustments must not exceed safety limits.

```
#include <stdio.h>

//Function prototype
void adjust_temperature(float *currentTemp, float minTemp, float
maxTemp, float adjustment);

int main()
{
   float currentTemp, adjustment, minTemp, maxTemp;

   // Define safety limits
   minTemp = 0.0;
   maxTemp = 1000.0;

   // Input the current temperature of the furnace and adjustment value
   printf("Enter the current temperature of the furnace: ");
   scanf("%f", &currentTemp);

   printf("Enter the temperature adjustment value: ");
   scanf("%f", &adjustment);

   // Call the function to adjust the temperature
   adjust_temperature(&currentTemp, minTemp, maxTemp, adjustment);

   printf("Adjusted temperature: %.2f degrees Celsius\n", currentTemp);

   return 0;
}
```

```
/*
Name: adjust_temperature()
Return Type: void
Parameter:(data type of each parameter): float*, float, float and float
Short description: it is used to control the temperature in furnace
*/

// Function to control the furnace temperature
void adjust_temperature(float *currentTemp, float minTemp, float maxTemp, float adjustment)
{
   // Adjust the current temperature by the given adjustment
   *currentTemp += adjustment;

   // Ensure the temperature stays within the safety limits
   if (*currentTemp < minTemp)
      *currentTemp = minTemp;
   else if (*currentTemp > maxTemp)
      *currentTemp = maxTemp;
}
```

O/P:

```
Enter the current temperature of the furnace: 100
Enter the temperature adjustment value: 20
Adjusted temperature: 120.00 degrees Celsius
```

6. Tool Life Tracker

Requirements:

- Input: Current tool usage hours (integer) and maximum life span.
- Output: Updated remaining life (integer).
- Function: Updates remaining life using pointers.
- Constraints: Remaining life cannot go below zero.

```
#include <stdio.h>

// Function prototype
void update_remaining_life(int *currentUsage, int maxLifeSpan, int
*remainingLife);

int main()
{
    int currentUsage, maxLifeSpan, remainingLife;

    // Input the current tool usage hours and maximum life span
    printf("Enter the current tool usage hours: ");
    scanf("%d", &currentUsage);
    printf("Enter the maximum life span of the tool (in hours): ");
    scanf("%d", &maxLifeSpan);

    // Call the function to update the remaining life
              update_remaining_life(&currentUsage,        maxLifeSpan,
&remainingLife);

    printf("Remaining life of the tool: %d hours\n", remainingLife);
    return 0;
}

/*
Name: update_remaining_life()
Return Type: void
Parameter:(data type of each parameter): int*, int and int
Short description: it is used to update the remaining life of the tool
*/

// Function to update the remaining life of the tool
void update_remaining_life(int *currentUsage, int maxLifeSpan, int *remainingLife)
{
    // Calculate the remaining life based on current usage
    *remainingLife = maxLifeSpan - *currentUsage;

    if (*remainingLife < 0)
        *remainingLife = 0;
}
```

O/P:

```
Enter the current tool usage hours: 45
Enter the maximum life span of the tool (in hours): 50
Remaining life of the tool: 5 hours
```

7. Material Weight Calculator

Requirements:

- Input: Weights of materials (array of floats).
- Output: Total weight (float).
- Function: Accepts a pointer to the array and calculates the sum of weights.
- Constraints: Ensure no negative weights are input.

```
#include <stdio.h>

// Function prototype
void calculate_total_weight(float *weights, int size, float *totalWeight);

int main()
{
    int size;

    // Input for the number of materials
    printf("Enter the number of materials: ");
    scanf("%d", &size);

    float weights[size];
    float totalWeight;

    // Input the weights of materials
    printf("Enter the weights of the materials\n");
    for (int i = 0; i < size; i++)
    {
        printf("Weight of material %d: ", i + 1);
        scanf("%f", &weights[i]);
        if (weights[i] < 0)
        {
            printf("Weight cannot be negative\n");
            return 1;
        }
    }

    // Call the function to calculate the total weight
    calculate_total_weight(weights, size, &totalWeight);

    printf("Total weight of the materials: %.2f\n", totalWeight);

    return 0;
}

/*
Name: calculate_total_weight()
Return Type: void
Parameter:(data type of each parameter): float*, int and float*
Short description: it is used to calculate the weight of the material
*/

// Function to calculate the total weight of materials
void calculate_total_weight(float *weights, int size, float *totalWeight)
{
    *totalWeight = 0.0;

    // Sum all the weights
    for (int i = 0; i < size; i++)
    {
        if (weights[i] < 0) {
            printf("Weights cannot be negative\n");
            return;
        }
        *totalWeight += weights[i];
    }
}
```

O/P:

```
Enter the number of materials: 3
Enter the weights of the materials
Weight of material 1: 5.6
Weight of material 2: 7.8
Weight of material 3: 15.8
Total weight of the materials: 29.20
```

8. Welding Machine Configuration

Requirements:

- Input: Voltage (float) and current (float).
- Output: Updated machine configuration.
- Function: Accepts pointers to voltage and current and modifies their values.

- Constraints: Validate that voltage and current stay within specified operating ranges.

```
#include <stdio.h>


#define MIN_V 10.0
#define MAX_V 50.0
#define MIN_C 50.0
#define MAX_C 500.0


// Function prototype
void update_machine_configuration(float *voltage, float *current);


int main()
{
  float voltage, current;


  // Input the voltage and current
  printf("Enter the voltage: ");
  scanf("%f", &voltage);


  printf("Enter the current: ");
  scanf("%f", &current);


  // Call the function to validate and update the machine configuration
  update_machine_configuration(&voltage, &current);


  printf("Updated machine configuration\n");
```

```
    printf("Voltage: %.2f V\n", voltage);
    printf("Current: %.2f A\n", current);

    return 0;
}

/*
Name: update_machine_configuration()
Return Type: void
Parameter:(data type of each parameter): float* and float*
Short description: it is used to update the configuration of welding machine
*/

// Function to update the welding machine configuration
void update_machine_configuration(float *voltage, float *current)
{
    if (*voltage < MIN_V)
    {
        *voltage = MIN_V;
        printf("Low voltage\n");
    }
    if (*voltage > MAX_V)
    {
        *voltage = MAX_V;
        printf("High voltage\n");
    }
    if (*current < MIN_C)
    {
        *current = MIN_C;
        printf("Low current\n");
    }
    if (*current > MAX_C)
    {
        *current = MAX_C;
        printf("High current\n");
    }
}
```

O/P:

```
Enter the voltage: 20
Enter the current: 10
Low current
Updated machine configuration
Voltage: 20.00 V
Current: 50.00 A
```

9. Defect Rate Analyzer

Requirements:

- Input: Total products and defective products (integers).
- Output: Defect rate (float).
- Function: Uses pointers to calculate defect rate = (Defective / Total) * 100.
- Constraints: Ensure total products > defective products.

```
#include <stdio.h>

// Function prototype
void calculate_defect_rate(int *total, int *defective, float *defectRate);

int main()
{
   int totalProducts, defectiveProducts;
   float defectRate;

   // Input the total products and defective products
   printf("Enter total number of products: ");
   scanf("%d", &totalProducts);

   printf("Enter number of defective products: ");
   scanf("%d", &defectiveProducts);

   // Call the function to calculate defect rate
   calculate_defect_rate(&totalProducts,                &defectiveProducts,
&defectRate);

   if (defectRate != -1)
      printf("Defect Rate: %.2f%%\n", defectRate);

   return 0;
}

/*
```

```
Name: calculate_defect_rate()
Return Type: void
Parameter:(data type of each parameter): int*, int* and float*
Short description: it is used to analyze the defect rate
*/

// Function to calculate defect rate using pointers
void calculate_defect_rate(int *total, int *defective, float *defectRate)
{
    if (*total > *defective)
        *defectRate = ((float)(*defective) / *total) * 100;
    else
    {
        printf("Total products must be greater than defective products\n");
        *defectRate = -1;
    }
}
```

O/P:

```
Enter total number of products: 4
Enter number of defective products: 3
Defect Rate: 75.00%
```

10. Assembly Line Optimization

Requirements:

- Input: Timing intervals between stations (array of floats).

- Output: Adjusted timing intervals.
- Function: Modifies the array values using pointers.
- Constraints: Timing intervals must remain positive.

```
#include <stdio.h>

// Function prototype
void timing_intervals(float *timings, int size);

int main()
{
   int n;

   // Input the number of timing intervals and intervals between the stations
   printf("Enter the number of timing intervals: ");
   scanf("%d", &n);

   if (n <= 0)
   {
      printf("Invalid timing interval\n");
      return 1;
   }

   float timings[n];

   printf("Enter the timing intervals (in seconds):\n");
   for (int i = 0; i < n; i++)
```

```
    {
        printf("Timing %d: ", i + 1);
        scanf("%f", &timings[i]);

        if (timings[i] <= 0)
        {
            printf("Invalid timing interval\n");
            return 1;
        }
    }

    // Call the function to optimize the timing intervals
    timing_intervals(timings, n);

    printf("Adjusted Timing Intervals\n");
    for (int i = 0; i < n; i++)
        printf("Timing %d: %.2f seconds\n", i + 1, timings[i]);

    return 0;
}

/*
Name: timing_interval()
Return Type: void
Parameter:(data type of each parameter): float* and int
Short description: it is used to optimize the timing intervals between stations
*/
```

```
// Function to modify timing intervals using pointers
void timing_intervals(float *timings, int size)
{
   int i;
   for (i = 0; i < size; i++)
   {
      if (*(timings + i) > 0)
         *(timings + i) *= 0.9;
      else
      {
         printf("Invalid timing interval\n");
         *(timings + i) = 0.1;
      }
   }
}
```

O/P:

```
Enter the number of timing intervals: 4
Enter the timing intervals (in seconds):
Timing 1: 10
Timing 2: 15
Timing 3: 18
Timing 4: 13
Adjusted Timing Intervals
Timing 1: 9.00 seconds
Timing 2: 13.50 seconds
```

Timing 3: 16.20 seconds

Timing 4: 11.70 seconds

11. CNC Machine Coordinates

Requirements:

- Input: Current x, y, z coordinates (floats).
- Output: Updated coordinates.
- Function: Accepts pointers to x, y, z values and updates them.
- Constraints: Ensure updated coordinates remain within machine limits.

```
#include <stdio.h>

#define X_LIMIT 50.0
#define Y_LIMIT 50.0
#define Z_LIMIT 50.0

// Function prototype
void update_coordinates(float *x, float *y, float *z);

int main()
{
    float x, y, z;

    // Input the current coordinates
    printf("Enter the current coordinates of x y and z: ");
    scanf("%f %f %f", &x, &y, &z);
    if (x < 0 || x > X_LIMIT || y < 0 || y > Y_LIMIT || z < 0 || z > Z_LIMIT)
    {
        printf("Coordinates must be within the limits\n");
        return 1;
    }

    // Call the function to update coordinates
    update_coordinates(&x, &y, &z);
    printf("Updated Coordinates\n");
    printf("x: %.2f\n", x);
    printf("y: %.2f\n", y);
    printf("z: %.2f\n", z);
    return 0;
}

/*
Name: update_coordinates()
Return Type: void
Parameter:(data type of each parameter): float*, float* and float*
Short description: it is used to optimize the timing intervals between stations
*/

// Function to update coordinates
void update_coordinates(float *x, float *y, float *z)
{
    float dx, dy, dz;
```

```
        printf("Enter the change in coordinates of x y and z: ");
        scanf("%f %f %f", &dx, &dy, &dz);

        *x += dx;
        *y += dy;
        *z += dz;
        if (*x < 0)
            *x = 0;
        else if (*x > X_LIMIT)
            *x = X_LIMIT;
        if (*y < 0)
            *y = 0;
        else if (*y > Y_LIMIT)
            *y = Y_LIMIT;
        if (*z < 0)
            *z = 0;
        else if (*z > Z_LIMIT)
            *z = Z_LIMIT;
    }
```

O/P:

```
Enter the current coordinates of x y and z: 3 4 5
Enter the change in coordinates of x y and z: 2 3 4
Updated Coordinates
x: 5.00
y: 7.00
z: 9.00
```

12. Energy Consumption Tracker

Requirements:

- Input: Energy usage data for machines (array of floats).
- Output: Total energy consumed (float).
- Function: Calculates and updates total energy using pointers.
- Constraints: Validate that no energy usage value is negative.

```
#include <stdio.h>


// Function prototype
float totalEnergy(float *usage, int size);


int main()
{
   int n;


   // Input the number of machines and energy usage of each machine
   printf("Enter the number of machines: ");
   scanf("%d", &n);


   if (n <= 0)
   {
      printf("Invalid number\n");
      return 1;
   }


   float energyUsage[n];
```

```
    printf("Enter the energy usage for each machine\n");
    for (int i = 0; i < n; i++)
    {
        printf("Machine %d: ", i + 1);
        scanf("%f", &energyUsage[i]);

        if (energyUsage[i] < 0)
        {
            printf("Invalid\n");
            return 1;
        }
    }

    // Call the function to calculate total energy consumed
    float total_energy = totalEnergy(energyUsage, n);

    if (total_energy != -1)
        printf("Total Energy Consumed: %.2f kWh\n", total_energy);

    return 0;
}

/*
Name: totalEnergy()
Return Type: float
Parameter:(data type of each parameter): float* and int
```

```
Short description: it is used to track the enrgy consumption of machines
*/

// Function to calculate total energy consumed
float totalEnergy(float *usage, int size)
{
   float total_energy = 0;
   int i;
   for (i = 0; i < size; i++)
   {
      if (*(usage + i) < 0)
      {
         printf("Invalid\n");
         return -1;
      }
      total_energy += *(usage + i);
   }
   return total_energy;
}
```

O/P:

```
Enter the number of machines: 3
Enter the energy usage for each machine
Machine 1: 20
Machine 2: 45
Machine 3: 30
Total Energy Consumed: 95.00 kWh
```

13. Production Rate Monitor

Requirements:

- Input: Current production rate (integer) and adjustment factor.
- Output: Updated production rate.
- Function: Modifies the production rate via a pointer.
- Constraints: Production rate must be within permissible limits.

```
#include <stdio.h>

#define MIN_RATE 50
#define MAX_RATE 100

// Function prototype
void update_production_rate(int *rate, float adjustmentFactor);

int main()
{
    int productionRate;
    float adjustmentFactor;

    // Input the production rate and adjustment factor
    printf("Enter the production rate: ");
    scanf("%d", &productionRate);

    if (productionRate < MIN_RATE || productionRate > MAX_RATE)
    {
        printf("Invalid production rate\n");
        return 1;
    }

    printf("Enter the adjustment factor: ");
    scanf("%f", &adjustmentFactor);

    // Call the function to update the production rate
    update_production_rate(&productionRate, adjustmentFactor);

    printf("Updated Production Rate: %d units per hour\n", productionRate);

    return 0;
}

/*
Name: update_production_rate()
Return Type: void
Parameter:(data type of each parameter): int* and float
Short description: it is used to monitor the production rate
*/

// Function to update the production rate
void update_production_rate(int *rate, float adjustmentFactor)
{
    *rate = (int)(*rate * adjustmentFactor);

    if (*rate < MIN_RATE)
```

```
        *rate = MIN_RATE;
    else if (*rate > MAX_RATE)
        *rate = MAX_RATE;
}
```

O/P:

```
Enter the production rate: 80
Enter the adjustment factor: 25
Updated Production Rate: 100 units per hour
```

14. Maintenance Schedule Update

Requirements:

- Input: Current and next maintenance dates (string).
- Output: Updated maintenance schedule.
- Function: Accepts pointers to the dates and modifies them.
- Constraints: Ensure next maintenance date is always later than the current date.

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

// Function prototypes
int compareDates(char *date1, char *date2);
void updateMaintenanceSchedule(char *currentDate, char *nextDate);

int main()
```

```
{
    char currentMaintenanceDate[11];
    char nextMaintenanceDate[11];

    // Input the current and next maintenance dates
    printf("Enter the current maintenance date (YYYY-MM-DD): ");
    scanf("%s", currentMaintenanceDate);
    printf("Enter the next maintenance date (YYYY-MM-DD): ");
    scanf("%s", nextMaintenanceDate);

    // Call the function to update maintenance schedule
                          updateMaintenanceSchedule(currentMaintenanceDate,
nextMaintenanceDate);

    printf("Updated Maintenance Schedule\n");
    printf("Current Maintenance Date: %s\n", currentMaintenanceDate);
    printf("Next Maintenance Date: %s\n", nextMaintenanceDate);

    return 0;
}

/*
Name: compareDates()
Return Type: int
Parameter:(data type of each parameter): char* and char*
Short description: it is used to compare dates
*/
```

```
// Function to compare dates (in the format YYYY-MM-DD)
int compareDates(char *date1, char *date2)
{
    int i;
    for (i = 0; i < 4; i++)  // Compare year
    {
        if (date1[i] < date2[i])
            return -1;
        else if (date1[i] > date2[i])
            return 1;
    }

    for (i = 5; i < 7; i++)  // Compare month
    {
        if (date1[i] < date2[i])
            return -1;
        else if (date1[i] > date2[i])
            return 1;
    }

    for (i = 8; i < 10; i++)  // Compare day
    {
        if (date1[i] < date2[i])
            return -1;
        else if (date1[i] > date2[i])
            return 1;
    }
    return 0; // Dates are equal
}

/*
Name: updateMaintenanceSchedule()
Return Type: void
Parameter:(data type of each parameter): char* and char*
Short description: it is used to update the maintenance schedule
*/

// Function to update the maintenance schedule
void updateMaintenanceSchedule(char *currentDate, char *nextDate)
{
    if (compareDates(nextDate, currentDate) <= 0)
    {
        printf("Invalid: Next maintenance date must be later than current date\n");
        return;
    }

    // Accept new next maintenance date
    printf("Enter the new next maintenance date (YYYY-MM-DD): ");
    scanf("%s", nextDate);

    while (compareDates(nextDate, currentDate) <= 0)
    {
        printf("Invalid: Next maintenance date must be later than current date\n");
```

```
            printf("Enter a valid next maintenance date (YYYY-MM-DD): ");
            scanf("%s", nextDate);
        }
    }
```

O/P:

```
Enter the current maintenance date (YYYY-MM-DD): 2025-01-02
Enter the next maintenance date (YYYY-MM-DD): 2025-02-02
Enter the new next maintenance date (YYYY-MM-DD): 2025-03-02
Updated Maintenance Schedule
Current Maintenance Date: 2025-01-02
Next Maintenance Date: 2025-03-02
```

15. Product Quality Inspection

Requirements:

- Input: Quality score (integer) for each product in a batch.
- Output: Updated quality metrics.
- Function: Updates quality metrics using pointers.
- Constraints: Ensure quality scores remain within 0-100.

```
#include <stdio.h>

// Function prototype
void updateQualityScore(int *score);

int main()
```

```
{
   int n;

   // Input the number of products in the batch
   printf("Enter the number of products in the batch: ");
   scanf("%d", &n);

   if (n <= 0)
   {
      printf("Invalid number\n");
      return 1;
   }

   int qualityScores[n], i;

   // Input the quality scores for each product
   printf("Enter the quality score for each product:\n");
   for (i = 0; i < n; i++)
   {
      printf("Product %d: ", i + 1);
      scanf("%d", &qualityScores[i]);

      // Call the function
      updateQualityScore(&qualityScores[i]);
   }

   printf("\nUpdated Quality Metrics:\n");
```

```
    for (int i = 0; i < n; i++)
        printf("Product %d: %d\n", i + 1, qualityScores[i]);

    return 0;
}

/*
Name: updateQualityScore()
Return Type: void
Parameter:(data type of each parameter): int*
Short description: it is used to update the quality score
*/

// Function to update quality score
void updateQualityScore(int *score)
{
    if (*score < 0)
        *score = 0;
    else if (*score > 100)
        *score = 100;
}
```

O/P:

```
Enter the number of products in the batch: 3
Enter the quality score for each product:
Product 1: 120
Product 2: 100
Product 3: 80

Updated Quality Metrics:
Product 1: 100
Product 2: 100
Product 3: 80
```

16. Warehouse Space Allocation

Requirements:

- Input: Space used for each section (array of integers).
- Output: Updated space allocation.
- Function: Adjusts space allocation using pointers.
- Constraints: Ensure total space used does not exceed warehouse capacity.

```
#include <stdio.h>

// Function prototype
void adjustSpaceAllocation(int *spaceUsed, int numSections, int *totalSpaceUsed, int warehouseCapacity);

int main()
{
    int numSections, totalSpaceUsed = 0;
    const int warehouseCapacity = 1000;

    printf("Enter the number of sections in the warehouse: ");
    scanf("%d", &numSections);
    if (numSections <= 0)
    {
        printf("Invalid number of sections\n");
        return 1;
    }

    int spaceUsed[numSections], i;

    printf("Enter the space used for each section:\n");
    for (i = 0; i < numSections; i++)
    {
        printf("Section %d: ", i + 1);
        scanf("%d", &spaceUsed[i]);
    }

    // Call the function
    adjustSpaceAllocation(spaceUsed, numSections, &totalSpaceUsed, warehouseCapacity);

    printf("\nUpdated Space Allocation:\n");
    for (int i = 0; i < numSections; i++)
        printf("Section %d: %d square meters\n", i + 1, spaceUsed[i]);

    printf("Total space used: %d square meters\n", totalSpaceUsed);

    return 0;
}
```

```
/*
Name: adjustSpaceAllocation()
Return Type: void
Parameter:(data type of each parameter): int*, int, int* and int
Short description: it is used to adjust the space allocation
*/

// Function to adjust space allocation
void adjustSpaceAllocation(int *spaceUsed, int numSections, int
*totalSpaceUsed, int warehouseCapacity)
{
   *totalSpaceUsed = 0;
   int i;
   for (i = 0; i < numSections; i++)
   {
      *totalSpaceUsed += *(spaceUsed + i);

      if (*totalSpaceUsed > warehouseCapacity)
      {
         printf("Total space used exceeds warehouse capacity\n");

         // Reduce space used in the last section to fit the capacity
         *(spaceUsed + i) -= (*totalSpaceUsed - warehouseCapacity);
         *totalSpaceUsed = warehouseCapacity;
         break;
      }
   }
}
```

O/P:

```
Enter the number of sections in the warehouse: 3
Enter the space used for each section:
Section 1: 450
Section 2: 700
Section 3: 1000
Total space used exceeds warehouse capacity

Updated Space Allocation:
Section 1: 450 square meters
Section 2: 550 square meters
Section 3: 1000 square meters
Total space used: 1000 square meters
```

17. Packaging Machine Settings

Requirements:

- Input: Machine settings like speed (float) and wrap tension (float).
- Output: Updated settings.
- Function: Modifies settings via pointers.
- Constraints: Validate settings remain within safe operating limits.

```
#include <stdio.h>

#define MIN_SPEED  10.0
```

```
#define MAX_SPEED  100.0
#define MIN_TENSION  10.0
#define MAX_TENSION  50.0

// Function prototype
void updateMachineSettings(float *speed, float *wrapTension);

int main()
{
   float speed, wrapTension;

   printf("Enter the machine speed: ");
   scanf("%f", &speed);

   printf("Enter the wrap tension: ");
   scanf("%f", &wrapTension);

   // Call the function
   updateMachineSettings(&speed, &wrapTension);

   printf("\nUpdated machine settings\n");
   printf("Speed: %.2f m/s\n", speed);
   printf("Wrap Tension: %.2f kg\n", wrapTension);

   return 0;
}
```

```
/*
Name: updateMachineSettings()
Return Type: void
Parameter:(data type of each parameter): float* and float*
Short description: it is used to modify machine settings
*/

// Function to modify machine settings
void updateMachineSettings(float *speed, float *wrapTension)
{
    if (*speed < MIN_SPEED)
    {
        *speed = MIN_SPEED;
        printf("Low speed\n");
    }
    else if (*speed > MAX_SPEED)
    {
        *speed = MAX_SPEED;
        printf("High speed\n");
    }

    if (*wrapTension < MIN_TENSION)
    {
        *wrapTension = MIN_TENSION;
        printf("Tension is low\n", MIN_TENSION);
    }
    else if (*wrapTension > MAX_TENSION)
```

```
        {
            *wrapTension = MAX_TENSION;
            printf("Tension is high\n", MAX_TENSION);
        }
    }
```

O/P:

```
Enter the machine speed: 60
Enter the wrap tension: 3
Tension is low

Updated machine settings
Speed: 60.00 m/s
Wrap Tension: 10.00 kg
```

18. Process Temperature Control

Requirements:

- Input: Current temperature (float).
- Output: Adjusted temperature.
- Function: Adjusts temperature using pointers.
- Constraints: Temperature must stay within a specified range.

```
#include <stdio.h>

#define MIN_TEMP  10.0
#define MAX_TEMP  80.0
```

```
// Function prototype
void adjustTemperature(float *currentTemperature);

int main()
{
    float currentTemperature;

    printf("Enter the temperature: ");
    scanf("%f", &currentTemperature);

    // Call the function
    adjustTemperature(&currentTemperature);

    printf("\nAdjusted Temperature: %.2f°C\n", currentTemperature);

    return 0;
}

/*
Name: adjustTemperature()
Return Type: void
Parameter:(data type of each parameter): float*
Short description: it is used to process the temperature control
*/

// Function to adjust the temperature
```

```
void adjustTemperature(float *currentTemperature)
{
   if (*currentTemperature < MIN_TEMP)
   {
      *currentTemperature = MIN_TEMP;
      printf("Low temperature\n", MIN_TEMP);
   }
   else if (*currentTemperature > MAX_TEMP)
   {
      *currentTemperature = MAX_TEMP;
      printf("High temperature\n", MAX_TEMP);
   }
}
```

O/P:

```
Enter the temperature: 8
Low temperature

Adjusted Temperature: 10.00°C
```

19. Scrap Material Management

Requirements:

- Input: Scrap count for different materials (array of integers).
- Output: Updated scrap count.
- Function: Modifies the scrap count via pointers.
- Constraints: Ensure scrap count remains non-negative.

```
#include <stdio.h>

// Function prototype
void updateScrapCount(int *scrapCount, int numMaterials);

int main()
{
    int n;

    printf("Enter the number of materials: ");
    scanf("%d", &n);

    if (n <= 0)
    {
        printf("Invalid number\n");
        return 1;
    }

    int scrapCount[n], i;

    printf("Enter the scrap count for each material\n");
    for (i = 0; i < n; i++)
    {
        printf("Material %d: ", i + 1);
        scanf("%d", &scrapCount[i]);
    }
    // Call the function
    updateScrapCount(scrapCount, n);

    printf("\nUpdated Scrap Count:\n");
    for (int i = 0; i < n; i++)
        printf("Material %d: %d\n", i + 1, scrapCount[i]);

    return 0;
}


/*
Name: updateScrapCount()
Return Type: void
Parameter:(data type of each parameter): int* and int
Short description: it is used to update the scrap count
*/

// Function to update scrap count
void updateScrapCount(int *scrapCount, int n)
{
    int i;
    for (i = 0; i < n; i++)
    {
        if (*(scrapCount + i) < 0)
        {
            printf("Invalid scrap count\n", i + 1);
            *(scrapCount + i) = 0;
        }
    }
}
```

O/P:

```
Enter the number of materials: 3
Enter the scrap count for each material
Material 1: 8
Material 2: -5
Material 3: 4
Invalid scrap count

Updated Scrap Count:
Material 1: 8
Material 2: 0
Material 3: 4
```

20. Shift Performance Analysis

Requirements:

- Input: Production data for each shift (array of integers).
- Output: Updated performance metrics.
- Function: Calculates and updates overall performance using pointers.
- Constraints: Validate data inputs before calculations.

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

```
// Function prototype
void updatePerformanceMetrics(int *shiftData, int numShifts, float
*overallPerformance);

int main()
{
    int numShifts;

    printf("Enter the number of shifts: ");
    scanf("%d", &numShifts);

    if (numShifts <= 0)
    {
        printf("Invalid shifts\n");
        return 1;
    }

    int shiftData[numShifts], i;
    float overallPerformance;

    printf("Enter the production data for each shift:\n");
    for (i = 0; i < numShifts; i++)
    {
        printf("Shift %d: ", i + 1);
        scanf("%d", &shiftData[i]);
        if (shiftData[i] < 0)
        {
            printf("Invalid input\n");
            shiftData[i] = 0;
        }
    }

    // Call the function
  updatePerformanceMetrics(shiftData, numShifts, &overallPerformance);
  printf("Overall Performance Metric: %.2f\n", overallPerformance);
  return 0;
}

/*
Name: updatePerformanceMetrics()
Return Type: void
Parameter:(data type of each parameter): int*, int and float*
Short description: it is used to update overall performance
*/

// Function to calculate and update overall performance
void updatePerformanceMetrics(int *shiftData, int numShifts, float
*overallPerformance)
{
    *overallPerformance = 0;
    int validShiftCount = 0, i;

    for (i = 0; i < numShifts; i++)
    {
        if (*(shiftData + i) >= 0)
```

```
            {
                *overallPerformance += *(shiftData + i);
                validShiftCount++;
            }
            else
                printf("Invalid data\n");
        }

        // Calculate average performance
        if (validShiftCount > 0)
            *overallPerformance /= validShiftCount;
        else
        {
            *overallPerformance = 0;
            printf("Warning: No valid shift data available.\n");
        }
    }
```

O/P:

```
Enter the number of shifts: 2
Enter the production data for each shift:
Shift 1: 100
Shift 2: -150
Invalid input
Overall Performance Metric: 50.00
```