Day 8 programs

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=56;

    int *pint;

    pint=&a;

    printf("The value of a=%d\n",*pint);

    return 0;

}
```

Output

The value of a=56

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=43.3;

    float *pflt;

    pflt=&a;
```

```
    printf("The original value of a=%f\n",*pflt);
    float n;
    *pflt=*pflt+10.0;
    n=*pflt;
    printf("The modified value of a=%f\n",n);
    return 0;
}
```

Output

The original value of a=43.299

The modified value of a=53.299

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 calculateSum(int *array, int size) {
    int sum = 0;
    for (int *ptr = array; ptr < array + size; ptr++) {
        sum += *ptr;
    }
    return sum;
}

int main() {
    int numbers[] = {1, 2, 3, 4};
    int size = sizeof(numbers) / sizeof(numbers[0]);
```

```
    int result = calculateSum(numbers, size);
    printf("The sum of the array elements is: %d\n", result);

    return 0;
}
```

Output

The sum of the array elements is:10

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 = NULL; // Declare a pointer and assign it a null value

    printf("Showcasing the use of a null pointer.\n");

    if (p == NULL) {
        printf("The pointer is null and cannot be dereferenced.\n");
    } else{
        printf("The value pointed to by p is: %d\n", *p);
    }

    int a = 42;
    p = &a;

    if (p == NULL) {
```

```
        printf("The pointer is still null.\n");
    } else {
        printf("The pointer now points to a valid address.\n");
        printf("The value pointed to by p is: %d\n", *p);
    }

    return 0;
}
```

Output

Showcasing the use of a null pointer.

The pointer is null and cannot be dereferenced.

The pointer now points to a valid address.

The value pointed to by p is:42

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 behavior.

```
#include <stdio.h>

int main() {
    int *ptr; // Uninitialized pointer

    printf("Dereferencing uninitialized pointer: %d\n", *ptr);

    return 0;
}
```

Output

Segmentation fault occurs

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 a=15;
    int *p1;
    p1=&a;
    int **p2;
    p2=&p1;
    printf("The value of p1 is %d\n",*p1);
    printf("The value of p2 is %d\n",**p2);

    return 0;
}
```

Output

The value of p1 is 15

The value of p2 is 15

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>
```

```c
#include <stdlib.h>

int main() {
    int size = 5;
    int *array = (int *)malloc(size * sizeof(int)); // Allocate memory

    if (array == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Populate and print array elements
    for (int i = 0; i < size; i++) {
        array[i] = i + 1; // Assign values
        printf("Element %d: %d\n", i + 1, array[i]);
    }

    return 0;
}
```

Output

Element 1:1

Element 2:2

Element 3:3

Element 4:4

Element 5:5

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>


// Function to calculate the sum of two integers
int add(int a, int b) {
    return a + b;
}


int main() {
    // Define a function pointer that matches the signature of `add`
    int (*functionPointer)(int, int) = &add;

    // Use the function pointer to call the function with different values
    int result1 = functionPointer(2, 12);
    int result2 = functionPointer(10, 10);

    // Print the results
    printf("The sum of 2 and 12 is: %d\n", result1);
    printf("The sum of 10 and 10 is: %d\n", result2);

    return 0;
}
```

Output

The sum of 2 and 12 is:14

The sum of 10 and 10 is:20

9.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() {
    int a = 10;
    int b = 20;
    int *const ptr = &a;  // Constant pointer to an integer


    // You can modify the data that ptr points to
    *ptr = 30; // Valid: Changing the value of a via the pointer
    printf("a = %d\n", a);  // Output: a = 30


    // You cannot change what ptr points to
    // ptr = &b; // Error: Cannot assign a new address to a constant pointer


    return 0;
}
```

10.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;
    int *p1;
    p1=&a;
    int b=20;
```

```
    int *p2;

    p2=&b;

    if (p1 > p2) {

        printf("p1 points to a higher memory address: %p\n",p1);

    } else if (p1 < p2) {

        printf("p1 points to a lower memory address: %p\n",p1);

    } else {

        printf("Both pointers point to the same memory address: %p\n",p1);

    }

    return 0;

}
```

Output

----

p1 points to a lower memory address:0x7ffda1551e50

Set 2 problems

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 = 20;

  int b = 45;

  int *const ptr=&a;// Declare a constant pointer to an integer and initialize it with the address of 'a'

  //change the value of using const pointer

  *ptr=30;
```

```
   // Try to change the pointer to point to 'b'
   // ptr = &b; // Uncommenting this line will cause an error

   printf("The value of a is %d\n",a);
   printf("The pointer is still pointing to a: %d\n",*ptr);
    return 0;
}
```

Output

The value of a is 30

The pointer is still pointing to 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()
{
   int a = 20;
   int b=50;
    const int *ptr=&a;
   // Try to modify the value of 'a' through the pointer (this will cause a compile-time
error)
   //*ptr=45;

   ptr=&b;//we can change the pointer to point to a different integer

   printf("The value of a is %d\n",a);
   printf("Value pointed to by ptr: %d\n", *ptr);
```

```
   return 0;

}
```

Output

The value of a is 20

Value pointed to by ptr:50

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()

{

   int a = 20;

   int b=50;

    const int *const ptr=&a;

   // Try to modify the value of 'a' through the pointer (this will cause a compile-time
error)

   //*ptr=45; uncommenting this line will cause an error


   //ptr=&b;uncommenting this line will cause an error


   printf("The value of a is %d\n",a);

   printf("Value pointed to by ptr: %d\n", *ptr);

   return 0;

}
```

Output

The value of a is 20

Value pointed to by ptr:20

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=20;
  int b=50;
  int c=80;
  int *const ptr=&a;
  *ptr=45;
  //Uncommenting both will raise compile time error
  //ptr=&b;
  //*ptr=100;

  //ptr=&c;
  //*ptr=120;

  printf("The value of a is %d\n",a);
  printf("The value of b is %d\n",b);
  printf("The value of c is %d\n",c);
  return 0;
}
```

Output

The value of a is 45

The value of b is 50

The value of c is 80

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.

```
#include <stdio.h>


int main()
{
  int a=20;
  int b=50;
  const int *ptr1=&a;
  const int *ptr2=&b;
  if(*ptr1 == *ptr2){
     printf("Both are equal");
  }
  else{
     printf("Not Equal");
  }
  return 0;
}
```

Output

Not 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;
    int b = 20;
    int c = 30;


    int* const ptr = &a;


    if (*ptr == 10) {
        printf("The pointer is pointing to a value of 10.\n");
    } else if (*ptr == 20) {
        printf("The pointer is pointing to a value of 20.\n");
    } else if (*ptr == 30) {
        printf("The pointer is pointing to a value of 30.\n");
    } else {
        printf("The pointer is pointing to a value not specified in the conditions.\n");
    }


    // Modify the value of 'a' to demonstrate different outcomes
    *ptr = 20;  // Change value of 'a' through the pointer
    if (*ptr == 10) {
        printf("The pointer is pointing to a value of 10.\n");
    } else if (*ptr == 20) {
        printf("The pointer is pointing to a value of 20.\n");
    } else if (*ptr == 30) {
        printf("The pointer is pointing to a value of 30.\n");
```

```
    } else {
        printf("The pointer is pointing to a value not specified in the conditions.\n");
    }

    return 0;
}
```

Output

The pointer is pointing to a value of 10

The pointer is pointing to a value of 20

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=20;
   int b=50;
   int *const ptr1=&a;
   int *const ptr2=&b;
   if(ptr1>ptr2){
      printf("ptr1 points to a higher address");
   }
   else if(ptr1<ptr2){
      printf("ptr2 points to a higher address");
   }
```

```
  else{
     printf("Both have same address");
  }
  return 0;
}
```

Output

----

ptr2 points to a higher address

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;
   int b = 20;
   int c = 30;

   // Declare a pointer to iterate over the variables
   int* ptr = NULL;

   // Manually iterate over variables using the pointer
   for (int i = 0; i < 3; i++) {
      if (i == 0) {
         ptr = &a;  // Point to 'a'
      } else if (i == 1) {
         ptr = &b;  // Point to 'b'
```

```
        } else if (i == 2) {
            ptr = &c;  // Point to 'c'
        }

        // Print the value pointed to by the pointer
        printf("Value of variable %d: %d\n", i + 1, *ptr);
    }

    return 0;
}
```

Output

Value of variable 1: 10

Value of variable 2: 20

Value of variable 3: 30

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 separate integer variables
    int a = 10;
    int b = 20;
    int c = 30;

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

```
    // Print the value of 'a'
    printf("Value of a: %d\n", *ptr);

    // Manually reassign the pointer indirectly to point to other variables
    *((int**)&ptr) = &b;
    printf("Value of b: %d\n", *ptr);

    *((int**)&ptr) = &c;
    printf("Value of c: %d\n", *ptr);

    return 0;
}
```

Output

Value of a: 10

Value of b: 20

Value of c: 30

set 3 problems related to call by reference

----

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>

void calculateEfficiency( float* inputPower,float* outputPower, float* efficiency) {
    if (*inputPower != 0) {
        *efficiency = (*outputPower / *inputPower) * 100;
    } else {
        *efficiency = 0;
    }
}

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

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

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

    printf("The machine's efficiency is: %.2f%%\n", efficiency);

    return 0;
}
```

Output

Enter the machine's input power (in watts):2

Enter the machine's output power (in watts):10

The machine's efficiency is:500.00%

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).

```c
#include <stdio.h>

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

    if (*currentSpeed > 100) {
        *currentSpeed = 100;
    } else if (*currentSpeed < 0) {
        *currentSpeed = 0;
    }
}

int main() {
    float currentSpeed, adjustmentValue;

    printf("Enter the current speed of the conveyor belt (0 to 100): ");
    scanf("%f", &currentSpeed);
```

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

    // Adjust the speed using the function
    adjustSpeed(&currentSpeed, adjustmentValue);

    // Output: Updated speed
    printf("The updated speed of the conveyor belt is: %.2f units\n", currentSpeed);

    return 0;
}
```

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 to update inventory levels based on production or consumption
void updateInventory(int* inventory, int size, int* adjustments) {
    for (int i = 0; i < size; i++) {
        inventory[i] += adjustments[i];
        if (inventory[i] < 0) { // Ensure inventory doesn't drop below zero
            inventory[i] = 0;
        }
```

```
    }
}

int main() {
    int size;

    // Input: Number of inventory items
    printf("Enter the number of inventory items: ");
    scanf("%d", &size);

    int inventory[size], adjustments[size];

    // Input: Current inventory levels
    printf("Enter the current inventory levels:\n");
    for (int i = 0; i < size; i++) {
        printf("Item %d: ", i + 1);
        scanf("%d", &inventory[i]);
    }

    // Input: Adjustments for each inventory item
    printf("Enter the adjustment values (positive for production, negative for
consumption):\n");
    for (int i = 0; i < size; i++) {
        printf("Item %d: ", i + 1);
        scanf("%d", &adjustments[i]);
    }

    // Update inventory levels
    updateInventory(inventory, size, adjustments);
```

```
    // Output: Updated inventory levels
    printf("Updated inventory levels:\n");
    for (int i = 0; i < size; i++) {
        printf("Item %d: %d\n", i + 1, inventory[i]);
    }

    return 0;
}
```

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>

// Function to update the coordinates with boundary check
void updatePosition(int* x, int* y, int* z, int deltaX, int deltaY, int deltaZ) {
    *x += deltaX;
    *y += deltaY;
    *z += deltaZ;

    // Ensure coordinates stay within boundaries (0 to 100)
    if (*x < 0) *x = 0;
    if (*x > 100) *x = 100;
    if (*y < 0) *y = 0;
```

```
    if (*y > 100) *y = 100;
    if (*z < 0) *z = 0;
    if (*z > 100) *z = 100;
}

int main() {
    int x, y, z;        // Current coordinates
    int deltaX, deltaY, deltaZ; // Movement deltas

    // Input current coordinates and movement deltas
    printf("Enter coordinates (x y z): ");
    scanf("%d %d %d", &x, &y, &z);
    printf("Enter deltas (deltaX deltaY deltaZ): ");
    scanf("%d %d %d", &deltaX, &deltaY, &deltaZ);

    // Update coordinates
    updatePosition(&x, &y, &z, deltaX, deltaY, deltaZ);

    // Output updated coordinates
    printf("Updated coordinates: x = %d, y = %d, z = %d\n", x, y, z);

    return 0;
}
```

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.

```c
#include <stdio.h>

// Function to adjust the temperature within the desired range
void adjustTemperature(float* currentTemperature, float minRange, float maxRange)
{
    // Ensure the temperature stays within the range
    if (*currentTemperature < minRange) {
        *currentTemperature = minRange;
    }
    if (*currentTemperature > maxRange) {
        *currentTemperature = maxRange;
    }
}

int main() {
    float currentTemperature, minRange, maxRange;

    // Input: Current temperature and desired range
    printf("Enter current temperature: ");
    scanf("%f", &currentTemperature);

    printf("Enter minimum desired temperature: ");
    scanf("%f", &minRange);

    printf("Enter maximum desired temperature: ");
    scanf("%f", &maxRange);
```

```
    // Adjust temperature using the function
    adjustTemperature(&currentTemperature, minRange, maxRange);

    // Output: Adjusted temperature
    printf("The adjusted temperature is: %.2f\n", currentTemperature);

    return 0;
}
```

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 to update remaining life of the tool
void updateToolLife(int* currentUsage, int maxLifeSpan, int* remainingLife) {
    *remainingLife = maxLifeSpan - *currentUsage;

    // Ensure remaining life doesn't go below zero
    if (*remainingLife < 0) {
        *remainingLife = 0;
    }
}
```

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

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

    printf("Enter maximum life span of the tool (in hours): ");
    scanf("%d", &maxLifeSpan);

    // Update remaining life
    updateToolLife(&currentUsage, maxLifeSpan, &remainingLife);

    // Output: Remaining life of the tool
    printf("The remaining life of the tool is: %d hours\n", remainingLife);

    return 0;
}
```

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 to calculate the total weight of materials
```

```
void calculateTotalWeight(float* weights, int size, float* totalWeight) {
    *totalWeight = 0.0;
    for (int i = 0; i < size; i++) {
        if (weights[i] < 0) weights[i] = 0; // Set negative weights to 0
        *totalWeight += weights[i];
    }
}

int main() {
    int size;
    float totalWeight;

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

    float weights[size];

    // Input: Weights of materials
    printf("Enter the weights of the materials:\n");
    for (int i = 0; i < size; i++) {
        printf("Material %d weight: ", i + 1);
        scanf("%f", &weights[i]);
    }

    // Calculate total weight
    calculateTotalWeight(weights, size, &totalWeight);

    // Output: Total weight
```

```c
    printf("Total weight: %.2f\n", totalWeight);

    return 0;
}
```

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.

```c
#include <stdio.h>

// Function to update welding machine configuration
void updateMachineConfig(float* voltage, float* current, float minVoltage, float
maxVoltage, float minCurrent, float maxCurrent) {
    // Validate and adjust voltage
    if (*voltage < minVoltage) *voltage = minVoltage;
    if (*voltage > maxVoltage) *voltage = maxVoltage;

    // Validate and adjust current
    if (*current < minCurrent) *current = minCurrent;
    if (*current > maxCurrent) *current = maxCurrent;
}

int main() {
    float voltage, current;
    float minVoltage = 10.0, maxVoltage = 50.0;  // Operating voltage range
```

```c
    float minCurrent = 5.0, maxCurrent = 200.0;  // Operating current range

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

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

    // Update machine configuration based on valid ranges
    updateMachineConfig(&voltage, &current, minVoltage, maxVoltage, minCurrent,
maxCurrent);

    // Output: Updated machine configuration
    printf("Updated machine configuration:\n");
    printf("Voltage: %.2f V (within range %.2f - %.2f V)\n", voltage, minVoltage,
maxVoltage);
    printf("Current: %.2f A (within range %.2f - %.2f A)\n", current, minCurrent,
maxCurrent);

    return 0;
}
```

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.

```c
#include <stdio.h>

// Function to calculate defect rate
void calculateDefectRate(int* totalProducts, int* defectiveProducts, float* defectRate)
{
    // Ensure total products is greater than defective products
    if (*totalProducts <= *defectiveProducts) {
        printf("Error: Total products must be greater than defective products.\n");
        *defectRate = -1;  // Error value
    } else {
        // Calculate defect rate: (Defective / Total) * 100
        *defectRate = ((float)*defectiveProducts / (float)*totalProducts) * 100;
    }
}

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

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

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

    // Calculate defect rate
    calculateDefectRate(&totalProducts, &defectiveProducts, &defectRate);
```

```c
    // Output: Defect rate
    if (defectRate != -1) {
        printf("The defect rate is: %.2f%%\n", defectRate);
    }

    return 0;
}
```

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.

```c
#include <stdio.h>

// Function to adjust timing intervals, ensuring they remain positive
void adjustTimingIntervals(float* intervals, int size) {
    for (int i = 0; i < size; i++) {
        if (intervals[i] <= 0) {
            printf("Warning: Interval %d is non-positive. Setting it to 1.0.\n", i + 1);
            intervals[i] = 1.0;  // Set non-positive interval to a default value of 1.0
        }
    }
}

int main() {
    int size;
```

```c
    // Input: Number of stations (size of the array)
    printf("Enter the number of stations: ");
    scanf("%d", &size);

    float intervals[size];

    // Input: Timing intervals between stations
    printf("Enter the timing intervals between stations:\n");
    for (int i = 0; i < size; i++) {
        printf("Station %d interval: ", i + 1);
        scanf("%f", &intervals[i]);
    }

    // Adjust timing intervals using the function
    adjustTimingIntervals(intervals, size);

    // Output: Adjusted timing intervals
    printf("Adjusted timing intervals:\n");
    for (int i = 0; i < size; i++) {
        printf("Station %d interval: %.2f\n", i + 1, intervals[i]);
    }

    return 0;
}
```

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>

// Function to update CNC machine coordinates
void updateCoordinates(float* x, float* y, float* z, float minX, float maxX, float minY,
float maxY, float minZ, float maxZ) {
    // Update x-coordinate within the limits
    if (*x < minX) *x = minX;
    if (*x > maxX) *x = maxX;

    // Update y-coordinate within the limits
    if (*y < minY) *y = minY;
    if (*y > maxY) *y = maxY;

    // Update z-coordinate within the limits
    if (*z < minZ) *z = minZ;
    if (*z > maxZ) *z = maxZ;
}

int main() {
    float x, y, z;
    // Define machine coordinate limits
    float minX = 0.0, maxX = 100.0;
    float minY = 0.0, maxY = 50.0;
    float minZ = 0.0, maxZ = 30.0;
```

```c
    // Input: Current x, y, z coordinates
    printf("Enter the current x-coordinate: ");
    scanf("%f", &x);

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

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

    // Update coordinates using the function
    updateCoordinates(&x, &y, &z, minX, maxX, minY, maxY, minZ, maxZ);

    // Output: Updated coordinates
    printf("Updated coordinates:\n");
    printf("x: %.2f (within range %.2f - %.2f)\n", x, minX, maxX);
    printf("y: %.2f (within range %.2f - %.2f)\n", y, minY, maxY);
    printf("z: %.2f (within range %.2f - %.2f)\n", z, minZ, maxZ);

    return 0;
}
```

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.

```c
#include <stdio.h>

// Function to calculate the total energy consumed
void calculateTotalEnergy(float* energyUsage, int size, float* totalEnergy) {
    *totalEnergy = 0.0;

    for (int i = 0; i < size; i++) {
        if (energyUsage[i] < 0) {
            printf("Warning: Negative energy usage detected for machine %d. Setting it to 0.\n", i + 1);
            energyUsage[i] = 0;  // Set negative values to 0
        }
        *totalEnergy += energyUsage[i];
    }
}

int main() {
    int size;
    float totalEnergy;

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

    float energyUsage[size];

    // Input: Energy usage data for each machine
    printf("Enter the energy usage for each machine:\n");
    for (int i = 0; i < size; i++) {
```

```
        printf("Machine %d energy usage: ", i + 1);
        scanf("%f", &energyUsage[i]);

        // Validate energy usage (ensure non-negative values)
        if (energyUsage[i] < 0) {
            printf("Warning: Negative energy usage entered. Setting it to 0.\n");
            energyUsage[i] = 0;  // Set to 0 if negative
        }
    }

    // Calculate total energy consumed
    calculateTotalEnergy(energyUsage, size, &totalEnergy);

    // Output: Total energy consumed
    printf("Total energy consumed: %.2f units\n", totalEnergy);

    return 0;
}
```

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>

// Function to adjust the production rate
```

```
void adjustProductionRate(int* productionRate, int adjustmentFactor, int minRate, int maxRate) {
    *productionRate += adjustmentFactor;

    // Ensure the production rate stays within the permissible limits
    if (*productionRate < minRate) {
        *productionRate = minRate;
    }
    if (*productionRate > maxRate) {
        *productionRate = maxRate;
    }
}

int main() {
    int productionRate, adjustmentFactor;
    int minRate = 50, maxRate = 500;  // Permissible production rate limits

    // Input: Current production rate
    printf("Enter current production rate: ");
    scanf("%d", &productionRate);

    // Input: Adjustment factor
    printf("Enter the adjustment factor: ");
    scanf("%d", &adjustmentFactor);

    // Adjust the production rate using the function
    adjustProductionRate(&productionRate, adjustmentFactor, minRate, maxRate);

    // Output: Updated production rate
```

```
    printf("Updated production rate: %d (within range %d - %d)\n", productionRate,
minRate, maxRate);

    return 0;
}
```

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 <string.h>

// Function to update maintenance schedule
void updateMaintenanceSchedule(char* currentDate, char* nextDate) {
    // Check if the next maintenance date is later than the current date
    if (strcmp(nextDate, currentDate) <= 0) {
        printf("Warning: Next maintenance date must be later than the current date.\n");
        printf("Please enter a valid next maintenance date: ");
        scanf("%s", nextDate);
    }
}

int main() {
    char currentDate[11], nextDate[11];
```

```c
    // Input: Current and next maintenance dates (in format YYYY-MM-DD)
    printf("Enter the current maintenance date (YYYY-MM-DD): ");
    scanf("%s", currentDate);

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

    // Update the maintenance schedule
    updateMaintenanceSchedule(currentDate, nextDate);

    // Output: Updated maintenance schedule
    printf("Updated maintenance schedule:\n");
    printf("Current date: %s\n", currentDate);
    printf("Next maintenance date: %s\n", nextDate);

    return 0;
}
```

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.

```c
#include <stdio.h>

// Function to process quality scores and calculate the average
```

```
void processScores(int* scores, int size, float* avgScore) {
    int sum = 0;

    for (int i = 0; i < size; i++) {
        if (scores[i] < 0) scores[i] = 0;       // Clamp to 0 if below range
        if (scores[i] > 100) scores[i] = 100; // Clamp to 100 if above range
        sum += scores[i];
    }

    *avgScore = (float)sum / size; // Calculate average
}

int main() {
    int n;

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

    int scores[n];
    float avgScore;

    // Input: Quality scores
    printf("Enter quality scores:\n");
    for (int i = 0; i < n; i++) {
        printf("Product %d: ", i + 1);
        scanf("%d", &scores[i]);
    }
```

```
    // Process scores
    processScores(scores, n, &avgScore);

    // Output: Updated scores and average
    printf("\nUpdated scores:\n");
    for (int i = 0; i < n; i++) {
        printf("Product %d: %d\n", i + 1, scores[i]);
    }
    printf("Average score: %.2f\n", avgScore);

    return 0;
}
```

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 to adjust space allocation
void adjustSpaceAllocation(int* sections, int size, int capacity) {
    int totalSpace = 0;

    for (int i = 0; i < size; i++) {
        totalSpace += sections[i];
        if (totalSpace > capacity) {
```

```
            printf("Warning: Total space exceeds capacity! Reducing space for section %d.\n", i + 1);
            sections[i] -= (totalSpace - capacity);
            totalSpace = capacity;
        }
    }
}

int main() {
    int n, capacity;

    // Input: Number of sections and warehouse capacity
    printf("Enter the number of sections: ");
    scanf("%d", &n);
    printf("Enter warehouse capacity: ");
    scanf("%d", &capacity);

    int sections[n];

    // Input: Space used for each section
    printf("Enter space used for each section:\n");
    for (int i = 0; i < n; i++) {
        printf("Section %d: ", i + 1);
        scanf("%d", &sections[i]);
    }

    // Adjust space allocation
    adjustSpaceAllocation(sections, n, capacity);
```

```
    // Output: Updated space allocation
    printf("\nUpdated space allocation:\n");
    for (int i = 0; i < n; i++) {
        printf("Section %d: %d\n", i + 1, sections[i]);
    }

    return 0;
}
```

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>

// Function to adjust machine settings
void adjustSettings(float* speed, float* tension, float maxSpeed, float minSpeed, float
maxTension, float minTension) {
    // Ensure speed is within limits
    if (*speed < minSpeed) {
        printf("Warning: Speed is below minimum. Setting to %.2f.\n", minSpeed);
        *speed = minSpeed;
    } else if (*speed > maxSpeed) {
        printf("Warning: Speed exceeds maximum. Setting to %.2f.\n", maxSpeed);
        *speed = maxSpeed;
    }
```

```
    // Ensure tension is within limits
    if (*tension < minTension) {
        printf("Warning: Tension is below minimum. Setting to %.2f.\n", minTension);
        *tension = minTension;
    } else if (*tension > maxTension) {
        printf("Warning: Tension exceeds maximum. Setting to %.2f.\n", maxTension);
        *tension = maxTension;
    }
}

int main() {
    float speed, tension;
    float maxSpeed = 100.0, minSpeed = 10.0;   // Speed limits
    float maxTension = 50.0, minTension = 5.0; // Tension limits

    // Input: Machine settings
    printf("Enter machine speed: ");
    scanf("%f", &speed);
    printf("Enter wrap tension: ");
    scanf("%f", &tension);

    // Adjust settings
    adjustSettings(&speed, &tension, maxSpeed, minSpeed, maxTension,
minTension);

    // Output: Updated settings
    printf("\nUpdated machine settings:\n");
    printf("Speed: %.2f\n", speed);
```

```
    printf("Wrap Tension: %.2f\n", tension);

    return 0;
}
```

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>

// Function to adjust temperature
void adjustTemperature(float* temperature, float minTemp, float maxTemp) {
    if (*temperature < minTemp) {
        printf("Warning: Temperature is below minimum. Setting to %.2f.\n", minTemp);
        *temperature = minTemp;
    } else if (*temperature > maxTemp) {
        printf("Warning: Temperature exceeds maximum. Setting to %.2f.\n",
maxTemp);
        *temperature = maxTemp;
    }
}

int main() {
    float currentTemperature;
    float minTemperature = 50.0, maxTemperature = 150.0; // Specified temperature
range
```

```
    // Input: Current temperature
    printf("Enter current temperature: ");
    scanf("%f", &currentTemperature);

    // Adjust temperature
    adjustTemperature(&currentTemperature, minTemperature, maxTemperature);

    // Output: Adjusted temperature
    printf("\nAdjusted temperature: %.2f\n", currentTemperature);

    return 0;
}
```

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 to adjust scrap counts
void adjustScrapCounts(int* scraps, int size) {
    for (int i = 0; i < size; i++) {
        if (scraps[i] < 0) {
            printf("Warning: Scrap count for material %d is negative. Setting to 0.\n", i +
1);
```

```
            scraps[i] = 0;
        }
    }
}

int main() {
    int n;

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

    int scraps[n];

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

    // Adjust scrap counts
    adjustScrapCounts(scraps, n);

    // Output: Updated scrap counts
    printf("\nUpdated scrap counts:\n");
    for (int i = 0; i < n; i++) {
        printf("Material %d: %d\n", i + 1, scraps[i]);
    }
```

```c
    return 0;
}
```

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.

```c
#include <stdio.h>

// Function to calculate overall performance
void analyzePerformance(int* production, int size, int* totalProduction, float* avgProduction) {
    *totalProduction = 0;

    for (int i = 0; i < size; i++) {
        if (production[i] < 0) {
            printf("Warning: Negative production data for shift %d. Setting to 0.\n", i + 1);
            production[i] = 0;
        }
        *totalProduction += production[i];
    }

    *avgProduction = (float)*totalProduction / size;
}
```

```c
int main() {
    int n;

    // Input: Number of shifts
    printf("Enter the number of shifts: ");
    scanf("%d", &n);

    int production[n];
    int totalProduction = 0;
    float avgProduction = 0.0;

    // Input: Production data for each shift
    printf("Enter production data for each shift:\n");
    for (int i = 0; i < n; i++) {
        printf("Shift %d: ", i + 1);
        scanf("%d", &production[i]);
    }

    // Analyze performance
    analyzePerformance(production, n, &totalProduction, &avgProduction);

    // Output: Performance metrics
    printf("\nShift Performance Analysis:\n");
    printf("Total Production: %d\n", totalProduction);
    printf("Average Production per Shift: %.2f\n", avgProduction);

    return 0;
}
```