POINTER-02-02-2025

Pointer Logic

```
#include<stdio.h>
int main(){
    int a=10;
    float b=2.5;
    char c='a';
    int* pint=&a;
    float* pflt =&b;
    char* pchar=&c;
    printf("Size of pint =%ld\n",sizeof(pint));
    printf("Size of pint =%ld\n",sizeof(pflt));
    printf("Size of pint =%ld\n",sizeof(pchar));

    printf("Address of a using pointer=%p\n",pint);
    printf("Address of a=%p\n",&a);
    printf("Address of pint=%p\n",&pint);

    printf("Address of b using pointer=%p\n",pflt);
    printf("Address of b=%p\n",&b);
    printf("Address of pflt=%p\n",&pflt);

    printf("Address of c using pointer=%p\n",pchar);
    printf("Address of c=%p\n",&c);
    printf("Address of pchar=%p\n",&pchar);

    *pflt=*pflt+5; //Write operation
    float n=*pflt; //read operation
    printf("n=%f\n",n);
}
```

Problem Statements

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 *ptr=&a;

        printf("a=%d\n",*ptr);

        return 0;

}
```

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=10;

    printf("Original valye of a=%f\n",a);

    float *ptr=&a;

    *ptr=*ptr+2.75;

    printf("Modified value of a=%f\n",a);

    return 0;

}
```

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>

float sum(float *a,int size);
```

```
int main(){
    int n;
    printf("Enter the size of array\n");
    scanf("%d",&n);
        float a[n];
        for(int i=0;i<n;i++){
            printf("a[%d]:",i+1);
            scanf("%f",&a[i]);
        }
        float sumElements=sum(a,n);
        printf("Sum of elemnts of array=%f\n",sumElements);
        return 0;
}
float sum(float *a,int size){
    float sum=0;
    for(int i=0;i<size;i++){
       sum+=*(a+i);
    }
    return sum;
}
```

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

```
    // Check if the pointer is null
    if (ptr == NULL) {
        printf("The pointer is null and cannot be dereferenced.\n");
    } else {
        // Dereferencing the pointer (this block will not execute because ptr is NULL)
        printf("The value at the pointer location is: %d\n", *ptr);
    }

    return 0;
}
```

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; // Declaring a pointer without initialization (wild pointer)
    printf("Dereferencing the wild pointer: %d\n", *ptr);
    return 0;
}
```

A pointer is considered "wild" when it is declared but not initialized. Its value is undefined and can point to any arbitrary memory address. This can lead to undefined behavior such as:

- Accessing memory that the program doesn't own.
- Crashes or segmentation faults.

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 *p=&a;

   int **ptr=&p;

   printf("Value of a using p=%d\n",*p);

   printf("Value of a using ptr=%d\n",**ptr);



   // printf("Address of a: %p\n", &a);

   // printf("Address stored in p: %p\n", p);

   // printf("Address of ptr: %p\n", &ptr);

   // printf("Address stored ptr: %p\n", *ptr);



   return 0;

}
```

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 *arr;

   int size,i;

   printf("Enter the number of elemnts in array:");

   scanf("%d",&size);
```

```
    // Dynamically allocate memory for the array using malloc
    arr=(int *)malloc(size*sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1; // Exit the program if memory allocation fails
    }
    printf("Enter the elements of the array:\n");
    for (i = 0; i < size; i++) {
        printf("Element %d: ", i + 1);
        scanf("%d", &arr[i]);
    }
     printf("\nThe values in the array are:\n");
    for (i = 0; i < size; i++) {
        printf("Element %d: %d\n", i + 1, *(arr + i));
    }

    // Free the allocated memory
    free(arr);

    return 0;
}
```

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,int);
```

```
int main() {
   int a,b;
   printf("Enter 2 integers\n");
   scanf("%d %d",&a,&b);
   int (*fun_ptr)(int,int);
   fun_ptr=&sum;
   int sum=fun_ptr(a,b);
   printf("%d + %d =%d\n",a,b,sum);
   return 0;
}
int sum(int a,int b){
   return a+b;
}
```

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.

Constant Pointer

```
#include <stdio.h>

int main() {

   int a=10;

   int b=5;

   printf("Original value of a=%d\n",a);

   int *const ptr=&a;

   // ptr=&b; Enabling this will cause read only pointer error

   *ptr=15;

   printf("a=%d\n",*ptr);



}
```

Constant Data pointer

```
#include <stdio.h>



int main() {
```

```
    int a=10;
    int const *ptr=&a;
    printf("Original value of a=%d\n",a);
   //  *ptr=15; This operation cannot be performed as it is a constant data pointer
    printf("a=%d\n",*ptr);

  }
```

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, b = 20;
    int *ptr1 = &a;
    int *ptr2 = &b;

    // Print the addresses stored in the pointers
    printf("Address stored in ptr1: %p\n", (void *)ptr1);
    printf("Address stored in ptr2: %p\n", (void *)ptr2);

    if (ptr1 > ptr2) {
        printf("ptr1 points to a higher memory address than ptr2.\n");
    } else if (ptr1 < ptr2) {
        printf("ptr1 points to a lower memory address than ptr2.\n");
    } else {
        printf("ptr1 and ptr2 point to the same memory address.\n");
    }
```

```
    return 0;
}
```

Problem Statements

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=5,b;

    printf("Value of a=%d(Exact value)\n",a);

    int *const ptr=&a;

    *ptr=15;

    printf("Value of a=%d(Modified value)\n",a);//value of integer can be Modified

   // ptr=&b;//Enabling this line draw assignment of read only error since ptr is constant pointer

    return 0;

}
```

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=5,b;

    printf("Value of a=%d(Exact value)\n",a);

    int const* ptr=&a;

    //*ptr=15;    //value of integer can't be Modified, since the variable is constant
```

```
    ptr=&b;

    printf("Address of b=%p\n",ptr);

    printf("Address of b=%p\n",&b);

    return 0;

}
```

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=5,b;

    printf("Value of a=%d(Exact value)\n",a);

    int const*const ptr=&a;

    //*ptr=15;   //value of both integer and pointer can't be Modified, since both are read only constants.

    //ptr=&b;

    printf("Address of b=%p\n",ptr);

    printf("Address of a=%p\n",&a);

    return 0;

}
```

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 n;
```

```
    printf("Enter the number of elements:");
    scanf("%d",&n);
    int a[n];
    for(int i=0;i<n;i++){
        printf("a[%d]=",i+1);
        scanf("%d",&a[i]);
    }
    int *const ptr=a; // By default ptr points to the first element of the array, so no need of &a.
    for(int i=0;i<n;i++){
        *(ptr+i)+=2;        // Modify the value at the current address
    }
     printf("Modified array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    return 0;
}
```

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 num1 = 10;
    int num2 = 20;
    const int *ptr1 = &num1;
    const int *ptr2 = &num2;
```

```
    // *ptr1=8; This is not possible
    // Use decision-making statements to compare the integers
    if (*ptr1 == *ptr2) {
        printf("The two integers are equal.\n");
    } else {
        printf("The two integers are not equal.\n");
    }
    return 0;
}
```

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 value1 = 10;
    int value2 = 20;
    // Declare a constant pointer to the first variable
    int *const ptr = &value2;

    // Check if the pointer points to a specific value
    if (*ptr == 10) {
        printf("The pointer is pointing to the value 10.\n");
    } else if (*ptr == 20) {
        printf("The pointer is pointing to the value 20.\n");
    } else {
        printf("The pointer is pointing to an unknown value.\n");
```

```
    }


    return 0;

}
```

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 value1 = 10;

    int value2 = 20;

    // Declare two constant pointers pointing to these variables

    int *const ptr1 = &value1;

    int *const ptr2 = &value2;

    if(ptr1>ptr2)

    printf("Value1 has highest address: %p",ptr1);

    else if(ptr1<ptr2)

    printf("Value2 has highest address: %p",ptr2);

    else

    printf("Invalid");

    return 0;

}
```

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 var1 = 10;
    float var2 = 3.14;
    char var3 = 'A';
    double var4 = 2.71828;

    // Declare constant pointers to each variable
    int * const ptr1 = &var1;
    float * const ptr2 = &var2;
    char * const ptr3 = &var3;
    double * const ptr4 = &var4;

    // Loop through the pointers
    printf("Using constant pointers:\n");
    printf("var1: %d\n", *ptr1);  // Dereference constant pointer to access value
    printf("var2: %.2f\n", *ptr2);
    printf("var3: %c\n", *ptr3);
    printf("var4: %.5f\n", *ptr4);

    return 0;
}
```

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() {
    int var1 = 10, var2 = 20, var3 = 30, var4 = 100;
    int *const ptr = &var1;  // Declare ptr as a constant pointer, pointing to var1
```

```
    printf("Values of the variables using a constant pointer and pointer arithmetic:\n");

    // Access the values using pointer arithmetic and constant pointer
    printf("Value of var1: %d\n", *ptr); // Dereference ptr to get the value of var1
    printf("Value of var2: %d\n", *(ptr + 1)); // Dereference ptr + 1 to get var2
    printf("Value of var3: %d\n", *(ptr + 2)); // Dereference ptr + 2 to get var3
    printf("Value of var4: %d\n", *(ptr + 3)); // Dereference ptr + 3 to get var4

    return 0;
}
```

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*,float*,float*);

int main() {
    float powerIn, powerOut,efficiency;
    printf("Enter the input power\n");
    scanf("%f",&powerIn);
    printf("Enter the output power\n");
    scanf("%f",&powerOut);
    calculateEfficiency(&powerIn,&powerOut,&efficiency);
    printf("Efficiency=%f\n",efficiency);
```

```
    return 0;
}
void calculateEfficiency(float *I,float *O,float *E){
    if(*I!=0){
        *E=(*O / *I)*100;
    }
    else
    *E=0;
}
```

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>
void adjustSpeed(float *, float);
int main() {
    float currentSpeed, adjustment;
    printf("Enter the current conveyor belt speed: ");
    scanf("%f", &currentSpeed);
    printf("Enter the adjustment value (positive to increase, negative to decrease): ");
    scanf("%f", &adjustment);
    adjustSpeed(&currentSpeed, adjustment);
    printf("The updated conveyor belt speed is: %.2f units\n", currentSpeed);
    return 0;
}
```

```
void adjustSpeed(float *currentSpeed, float adjustment) {
    *currentSpeed += adjustment; // Adjust the speed

    if (*currentSpeed < 0) {
        *currentSpeed = 0;  // Set speed to 0 if it goes below 0
    } else if (*currentSpeed > 100) {
        *currentSpeed = 100;  // Set speed to 100 if it exceeds 100
    }
}
```

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.

```
#include <stdio.h>

void printInventory(int *inventory, int size);

void updateInventory(int *inventory, int size, int *consumption, int *production);

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

    int inventory[n];  // Array to store inventory levels
    int consumption[n]; // Array to store consumption levels
```

```
    int production[n];  // Array to store production levels

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

    // Input consumption levels (raw material consumed)
    printf("Enter the consumption levels for each material:\n");
    for (int i = 0; i < n; i++) {
        printf("Consumption of material %d: ", i + 1);
        scanf("%d", &consumption[i]);
    }
    // Input production levels (raw material produced)
    printf("Enter the production levels for each material:\n");
    for (int i = 0; i < n; i++) {
        printf("Production of material %d: ", i + 1);
        scanf("%d", &production[i]);
    }
    // Update inventory based on consumption and production
    updateInventory(inventory, n, consumption, production);//here & is not passing, coz already array points to address

    // Output the updated inventory levels
```

```
    printf("\nUpdated inventory levels:\n");
    printInventory(inventory, n);
    return 0;
}
// Function to update inventory levels
void updateInventory(int *inventory, int size, int *consumption, int *production) {
    for (int i = 0; i < size; i++) {
        // Adjust inventory based on consumption and production
        inventory[i] = inventory[i] - consumption[i] + production[i];
        // Ensure inventory level doesn't go below zero
        if (inventory[i] < 0) {
            inventory[i] = 0;
        }
    }
}
// Function to print inventory levels
void printInventory(int *inventory, int size) {
    for (int i = 0; i < size; i++) {
        printf("Inventory level of raw material %d: %d units\n", i + 1, inventory[i]);
    }
}
```

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>

int updateCoordinates(int *x, int *y, int *z, int deltaX, int deltaY, int deltaZ,
                      int minX, int maxX, int minY, int maxY, int minZ, int maxZ) {
    // Update coordinates by adding delta values
    *x += deltaX;
    *y += deltaY;
    *z += deltaZ;
    // Check if coordinates exceed boundaries and return status
    if (*x < minX || *x > maxX || *y < minY || *y > maxY || *z < minZ || *z > maxZ) {
        return 1;  // Return 1 indicating that coordinates exceeded the boundary
    }

    return 0;  // Return 0 indicating that coordinates are within the boundary
}
int main() {
    int x = 10, y = 20, z = 30;
    int deltaX, deltaY, deltaZ;
    int minX = 0, maxX = 100, minY = 0, maxY = 100, minZ = 0, maxZ = 100;

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

```
    // Update coordinates and check if they exceed the boundary

    int result = updateCoordinates(&x, &y, &z, deltaX, deltaY, deltaZ, minX, maxX, minY,
maxY, minZ, maxZ);



    if (result == 1) {

        printf("Warning: Coordinates exceeded boundary limits!\n");

    } else {

        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.

```
 #include <stdio.h>

void adjustTemperature(float *currentTemp, float minTemp, float maxTemp) {

    // Ensure the temperature stays within the safety limits

    if (*currentTemp < minTemp) {

        *currentTemp = minTemp;  // If temperature is below the min limit, set it to min

    } else if (*currentTemp > maxTemp) {

        *currentTemp = maxTemp;  // If temperature is above the max limit, set it to max

    }

}
```

```
int main() {
    float currentTemperature;
    float minTemp, maxTemp;

    // Get the current temperature and desired temperature range from the user
    printf("Enter the current temperature: ");
    scanf("%f", &currentTemperature);

    printf("Enter the minimum safe temperature: ");
    scanf("%f", &minTemp);

    printf("Enter the maximum safe temperature: ");
    scanf("%f", &maxTemp);

    // Adjust the temperature
    adjustTemperature(&currentTemperature, minTemp, maxTemp);

    // Output the 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>

void updateToolLife(int *, int, int);

int main() {

    int currentUsage, maxLife, remainingLife;



    printf("Enter the current usage hours of the tool: ");

    scanf("%d", &currentUsage);



    printf("Enter the maximum life span of the tool (in hours): ");

    scanf("%d", &maxLife);



    updateToolLife(&remainingLife, currentUsage, maxLife);

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



    return 0;

}

void updateToolLife(int *remainingLife, int currentUsage, int maxLife) {

    // Calculate the remaining life

    *remainingLife = maxLife - currentUsage;



    // Ensure the remaining life does not go below zero

    if (*remainingLife < 0) {

        *remainingLife = 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>

float calculateTotalWeight(float*, int);

int main(){
    int n;
    printf("Enter the number of material\n");
    scanf("%d",&n);
     if (n <= 0) {
        printf("Invalid number of materials.\n");
        return 1;
    }
    float weight[n];
    printf("Enter the weight of materials:");
    for(int i=0;i<n;i++){
        printf("weight[%d]:",i+1);
        scanf("%f",&weight[i]);
    }
     float totalWeight = calculateTotalWeight(weight, n);
    printf("Total weight=%f\n",totalWeight);
```

```
    return 0;

}

float calculateTotalWeight(float* weight, int n){

    float total=0;

    for(int i=0;i<n;i++){

        total+=weight[i];

    }

    return total;

}
```

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>

void configureWeldingMachine(float*,float*,float,float,float,float);

int main(){

    float voltage,current;

    printf("Enter voltage:\n");

    scanf("%f",&voltage);

    printf("Enter current:\n");

    scanf("%f",&current);

    float minVoltage = 220.0, maxVoltage = 440.0; // Voltage operating range

    float minCurrent = 10.0, maxCurrent = 100.0; // Current operating range

    configureWeldingMachine(&voltage, &current, minVoltage, maxVoltage, minCurrent,
maxCurrent);
```

```
    printf("Updated welding machine configuration:\n");
    printf("Voltage: %.2f V\n", voltage);
    printf("Current: %.2f A\n", current);
    return 0;
}
void configureWeldingMachine(float* voltage,float* current,float minVoltage,float
maxVoltage,float minCurrent,float maxCurrent){
     // Validate and adjust voltage
    if (*voltage < minVoltage) {
        printf("Voltage too low. Adjusting to minimum voltage (%.2f).\n", minVoltage);
        *voltage = minVoltage;
    } else if (*voltage > maxVoltage) {
        printf("Voltage too high. Adjusting to maximum voltage (%.2f).\n", maxVoltage);
        *voltage = maxVoltage;
    }
    // Validate and adjust current
    if (*current < minCurrent) {
        printf("Current too low. Adjusting to minimum current (%.2f).\n", minCurrent);
        *current = minCurrent;
    } else if (*current > maxCurrent) {
        printf("Current too high. Adjusting to maximum current (%.2f).\n", maxCurrent);
        *current = maxCurrent;
    }
}
```

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>

void calcDefectiveRate(int *defective,int *total,float *defectRate);

int main()

{

  int total,sum=0;

  printf("Enter total products:");

  scanf("%d",&total);

  int defective;

  float defectRate;

  printf("\nEnter the defective products\n");

  scanf("%d",&defective);

  while(1){

     if(defective<total){

       break;

     }

     else{

        printf("Invalid entry\n");

     }



  }

  calcDefectiveRate(&defective,&total,&defectRate);
```

```
  printf("Defective rate=%0.2f \n",defectRate);

  return 0;

}


void calcDefectiveRate(int *defective,int *total,float *defectRate){

    *defectRate=((float)*defective / *total)*100;

}
```

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>

void optimizeTiming(float *timing, int size);


int main() {

    int n;

    printf("Enter the number of timing intervals: ");

    scanf("%d", &n);

    float timing[n];

    // Input the timing intervals

    printf("Enter the timing intervals (in seconds):\n");

    for (int i = 0; i < n; i++) {

        scanf("%f", &timing[i]);


        // Ensure timing intervals remain positive
```

```
        if (timing[i] <= 0) {
            printf("Invalid input. Timing intervals must be positive.\n");
            return 1;
        }
    }

    // Optimize timing intervals
    optimizeTiming(timing, n);

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

    return 0;
}

void optimizeTiming(float *timing, int size) {
    for (int i = 0; i < size; i++) {
        *(timing + i) *= 0.9;  // Example adjustment: reduce by 10%
    }
}
```

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>

void updateCoordinates(float *x, float *y, float *z, float dx, float dy, float dz,

                       float xmin, float xmax, float ymin, float ymax, float zmin, float zmax);


int main() {

    float x, y, z;       // Current coordinates

    float dx, dy, dz;    // Displacements

    float xmin = 0, xmax = 100;

    float ymin = 0, ymax = 100;

    float zmin = 0, zmax = 100;

    // Input current coordinates

    printf("Enter current x, y, z coordinates: ");

    scanf("%f %f %f", &x, &y, &z);

    // Input displacements

    printf("Enter displacement dx, dy, dz: ");

    scanf("%f %f %f", &dx, &dy, &dz);

    // Update coordinates

    updateCoordinates(&x, &y, &z, dx, dy, dz, xmin, xmax, ymin, ymax, zmin, zmax);


    // Output updated coordinates

    printf("Updated coordinates: x = %.2f, y = %.2f, z = %.2f\n", x, y, z);
```

```
    return 0;
}

void updateCoordinates(float *x, float *y, float *z, float dx, float dy, float dz,
                       float xmin, float xmax, float ymin, float ymax, float zmin, float zmax) {
    // Update x coordinate and ensure it's within limits
    *x += dx;
    if (*x < xmin) *x = xmin;
    if (*x > xmax) *x = xmax;

    // Update y coordinate and ensure it's within limits
    *y += dy;
    if (*y < ymin) *y = ymin;
    if (*y > ymax) *y = ymax;

    // Update z coordinate and ensure it's within limits
    *z += dz;
    if (*z < zmin) *z = zmin;
    if (*z > zmax) *z = zmax;
}
```

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>
void calculateTotalEnergy(float *energyArray, int size, float *totalEnergy);
int main() {
    int n;
    printf("Enter the number of machines: ");
    scanf("%d", &n);
    float energyArray[n];
    float totalEnergy = 0.0;
    printf("Enter energy usage for each machine (in kWh):\n");
    for (int i = 0; i < n; i++) {
        printf("Machine %d: ", i + 1);
        scanf("%f", &energyArray[i]);

        if (energyArray[i] < 0) {
            printf("Invalid input. Energy usage cannot be negative.\n");
            return 1;
        }
    }
    // Calculate total energy consumption
    calculateTotalEnergy(energyArray, n, &totalEnergy);
    printf("Total energy consumed: %.2f kWh\n", totalEnergy);

    return 0;
}
```

```
void calculateTotalEnergy(float *energyArray, int size, float *totalEnergy) {
    *totalEnergy = 0.0;
    // Sum up energy usage values
    for (int i = 0; i < size; i++) {
        *totalEnergy += *(energyArray + i);
    }
}
```

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>
void updateProductionRate(int *rate, float adjustmentFactor, int minLimit, int maxLimit);

int main() {
    int currentRate;
    float adjustmentFactor;
    int minLimit = 50, maxLimit = 1000;

    printf("Enter the current production rate: ");
    scanf("%d", &currentRate);

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

```
    updateProductionRate(&currentRate, adjustmentFactor, minLimit, maxLimit);

    printf("Updated production rate: %d\n", currentRate);

    return 0;
}

void updateProductionRate(int *rate, float adjustmentFactor, int minLimit, int maxLimit) {
    *rate = (int)(*rate * adjustmentFactor);

    if (*rate < minLimit) {
        *rate = minLimit;
    } else if (*rate > maxLimit) {
        *rate = maxLimit;
    }
}
```

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>
#include <stdbool.h>
bool updateMaintenanceSchedule(char *currentDate, char *nextDate);
```

```
int main() {
    char currentDate[11], nextDate[11];
    printf("Enter the current maintenance date (YYYY-MM-DD): ");
    scanf("%s", currentDate);
    printf("Enter the next maintenance date (YYYY-MM-DD): ");
    scanf("%s", nextDate);
    if (updateMaintenanceSchedule(currentDate, nextDate)) {
        printf("Maintenance schedule updated successfully.\n");
        printf("Current maintenance date: %s\n", currentDate);
        printf("Next maintenance date: %s\n", nextDate);
    } else {
        printf("Invalid input. The next maintenance date must be later than the current date.\n");
    }

    return 0;
}
bool updateMaintenanceSchedule(char *currentDate, char *nextDate) {
    if (strcmp(nextDate, currentDate) > 0) {
        // Valid schedule update
        return true;
    } else {
        // Invalid: nextDate is not later than currentDate
        return false;
    }
}
```

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>


void updateQualityMetrics(int *qualityScores, int size, int *minScore, int *maxScore, float
*averageScore);


int main() {
    int n;
    printf("Enter the number of products in the batch: ");
    scanf("%d", &n);

    int qualityScores[n];
    int minScore = 100, maxScore = 0;
    float averageScore = 0.0;

    printf("Enter quality scores for each product (0-100):\n");
    for (int i = 0; i < n; i++) {
        printf("Product %d: ", i + 1);
        scanf("%d", &qualityScores[i]);

        // Validate that the score is within 0-100
        if (qualityScores[i] < 0 || qualityScores[i] > 100) {
```

```
            printf("Invalid score. Quality scores must be between 0 and 100.\n");
            return 1;
        }
    }

    // Update quality metrics
    updateQualityMetrics(qualityScores, n, &minScore, &maxScore, &averageScore);

    printf("\nQuality Metrics:\n");
    printf("Minimum Score: %d\n", minScore);
    printf("Maximum Score: %d\n", maxScore);
    printf("Average Score: %.2f\n", averageScore);

    return 0;
}

void updateQualityMetrics(int *qualityScores, int size, int *minScore, int *maxScore, float
*averageScore) {
    int totalScore = 0;

    for (int i = 0; i < size; i++) {
        if (qualityScores[i] < *minScore) {
            *minScore = qualityScores[i];
        }
        if (qualityScores[i] > *maxScore) {
            *maxScore = qualityScores[i];
        }
```

```
        totalScore += qualityScores[i];
    }

    *averageScore = (float)totalScore / size;
}
```

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>

void adjustSpaceAllocation(int *spaces, int size, int maxCapacity);

int main() {
    int n, maxCapacity;
    printf("Enter the number of sections: ");
    scanf("%d", &n);
    printf("Enter the maximum warehouse capacity: ");
    scanf("%d", &maxCapacity);
    int spaces[n];
    printf("Enter space used for each section:\n");
    for (int i = 0; i < n; i++) {
        printf("Section %d: ", i + 1);
        scanf("%d", &spaces[i]);
        if (spaces[i] < 0) {
            printf("Invalid input. Space used cannot be negative.\n");
            return 1;
```

```
        }
    }
    adjustSpaceAllocation(spaces, n, maxCapacity);
    printf("\nUpdated Space Allocation:\n");
    for (int i = 0; i < n; i++) {
        printf("Section %d: %d\n", i + 1, spaces[i]);
    }

    return 0;
}
void adjustSpaceAllocation(int *spaces, int size, int maxCapacity) {
    int totalSpaceUsed = 0;

    for (int i = 0; i < size; i++) {
        totalSpaceUsed += spaces[i];
    }

    if (totalSpaceUsed > maxCapacity) {
        float reductionFactor = (float)maxCapacity / totalSpaceUsed;

        for (int i = 0; i < size; i++) {
            spaces[i] = (int)(spaces[i] * reductionFactor);
        }
    }
}
```

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>

void updateMachineSettings(float *speed, float *tension, float maxSpeed, float minSpeed, float maxTension, float minTension);

int main() {

    float speed, tension;

    float maxSpeed = 120.0, minSpeed = 10.0;

    float maxTension = 50.0, minTension = 5.0;

    printf("Enter current machine speed (RPM): ");

    scanf("%f", &speed);

    printf("Enter current wrap tension (N): ");

    scanf("%f", &tension);

    updateMachineSettings(&speed, &tension, maxSpeed, minSpeed, maxTension, minTension);

    printf("\nUpdated Machine Settings:\n");

    printf("Speed: %.2f RPM\n", speed);

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



    return 0;

}

void updateMachineSettings(float *speed, float *tension, float maxSpeed, float minSpeed, float maxTension, float minTension) {

    if (*speed > maxSpeed) {
```

```
        printf("Warning: Speed exceeds maximum limit. Adjusting to %.2f RPM.\n",
maxSpeed);

        *speed = maxSpeed;

    } else if (*speed < minSpeed) {

        printf("Warning: Speed is below minimum limit. Adjusting to %.2f RPM.\n",
minSpeed);

        *speed = minSpeed;

    }

    if (*tension > maxTension) {

        printf("Warning: Wrap tension exceeds maximum limit. Adjusting to %.2f N.\n",
maxTension);

        *tension = maxTension;

    } else if (*tension < minTension) {

        printf("Warning: Wrap tension is below minimum limit. Adjusting to %.2f N.\n",
minTension);

        *tension = minTension;

    }

}
```

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>



void adjustTemperature(float *currentTemperature, float minTemperature, float
maxTemperature);
```

```
int main() {

    float currentTemperature;

    float minTemperature = 15.0, maxTemperature = 75.0;  // Safe operating range for
temperature (in degrees Celsius)




    printf("Enter current process temperature (°C): ");

    scanf("%f", &currentTemperature);



    adjustTemperature(&currentTemperature, minTemperature, maxTemperature);



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



    return 0;

}

void adjustTemperature(float *currentTemperature, float minTemperature, float
maxTemperature) {

    if (*currentTemperature > maxTemperature) {

        printf("Warning: Temperature exceeds maximum limit. Adjusting to %.2f °C.\n",
maxTemperature);

        *currentTemperature = maxTemperature;

    } else if (*currentTemperature < minTemperature) {

        printf("Warning: Temperature is below minimum limit. Adjusting to %.2f °C.\n",
minTemperature);

        *currentTemperature = minTemperature;

    }

}
```

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>


void updateScrapCount(int *scrapCount, int size);


int main() {
    int n;
    printf("Enter the number of materials: ");
    scanf("%d", &n);
    int scrapCount[n];
    printf("Enter scrap count for each material:\n");
    for (int i = 0; i < n; i++) {
        printf("Material %d: ", i + 1);
        scanf("%d", &scrapCount[i]);
        if (scrapCount[i] < 0) {
            printf("Invalid input. Scrap count cannot be negative.\n");
            return 1;
        }
    }


    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;
}
void updateScrapCount(int *scrapCount, int size) {
    // Example update: Increment scrap count for each material by 1
    for (int i = 0; i < size; i++) {
        scrapCount[i] += 1;  // Modify the scrap count
    }
}
```

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>

void calculatePerformance(int *productionData, int size, float *averagePerformance, int *totalProduction);

int main() {
    int n;
    printf("Enter the number of shifts: ");
    scanf("%d", &n);

    int productionData[n];
```

```
    printf("Enter production data for each shift:\n");
    for (int i = 0; i < n; i++) {
        printf("Shift %d production: ", i + 1);
        scanf("%d", &productionData[i]);
        if (productionData[i] < 0) {
            printf("Invalid input. Production data cannot be negative.\n");
            return 1;
        }
    }
    float averagePerformance;
    int totalProduction;
    calculatePerformance(productionData, n, &averagePerformance, &totalProduction);
    printf("\nPerformance Metrics:\n");
    printf("Total Production: %d\n", totalProduction);
    printf("Average Production per Shift: %.2f\n", averagePerformance);
    return 0;
}

void calculatePerformance(int *productionData, int size, float *averagePerformance, int
*totalProduction) {
    *totalProduction = 0;

    for (int i = 0; i < size; i++) {
        *totalProduction += productionData[i];
    }
```

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