```c
#include <stdio.h>

int main()
{
    int a;
    float b;
    char c;

    int *pint=&a;
    float *pflt=&b;
    char *pchr=&c;

    printf("Size of pint = %d \n",sizeof(pint));
    printf("Size of pflt = %d \n",sizeof(pflt));
    printf("Size of pchr = %d \n",sizeof(pchr));

    return 0;
}
```

```c
#include <stdio.h>

int main()
{
    int a;
    float b;
    char c;

    int * pint=&a;
    float * pflt=&b;
    char * pchr=&c;

    printf("Size of pint = %d \n",sizeof(pint));
    printf("Size of pflt = %d \n",sizeof(pflt));
    printf("Size of pchr = %d \n",sizeof(pchr));
    printf("Address of a=%p \n",&a);
    printf("Address of b=%p \n",&b);
    printf("Address of c=%p \n",&c);
    printf("pint = %p \n",pint);
    printf("pflt = %p \n",pflt);
    printf("pchr = %p \n",pchr);

    printf("Address of pint=%p \n",&pint);
    printf("Address of pflt=%p \n",&pflt);
    printf("Address of pchr=%p \n",&pchr);

    return 0;
}
```

```
#include <stdio.h>

int main()
   {
   int a =10;
   float b =3.14;
   char c ='c';
   float n;

   int * pint=&a;
   float * pflt=&b;
   char * pchr=&c;

   *pflt = *pflt+10.0;
   n=*pflt;
   printf("n = %f \n",n);



   return 0;
}
```

POINTER PROBLEMS

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 n = 10;          // Declare an integer variable
   int *ptr = &n;        // Declare a pointer to an integer and initialize it with the address of 'num'

   // Print the value of the variable using the pointer
   printf("The value of num is: %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() {
   // Declare a float variable float
   n = 5.5;

   // Declare a pointer to float float
   *ptr;

   // Point ptr to the address of n ptr =
   &n;
```

```
// Print the original value of n printf("Original value
of n: %.2f\n", n);

      // Use the pointer to change the value of n
      *ptr = 20.75;

      // Print the modified value of n
      printf("Modified value of n: %.2f\n", n);

      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>

int sumArray(int *arr, int size) {
    int sum = 0;
    // Iterate through the array using pointer arithmetic
    for (int i = 0; i < size; i++) {
        sum += *(arr + i); // Accessing the array element using pointer arithmetic
    }
    return sum;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5}; // Example array
    int size = sizeof(arr) / sizeof(arr[0]); // Calculate size of the array

    int result = sumArray(arr, size); // Calling the sum function
    printf("The sum of the array elements is: %d\n", result);

    return 0;
}
```

3. 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 before attempting to dereference it
    if (ptr == NULL) {
        printf("The pointer is null, cannot dereference.\n");
    } else {
        // Dereference the pointer if it is not null (this block won't be executed in this case)
        printf("Dereferencing the pointer: %d\n", *ptr);
    }

    return 0;
}
```

4. 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;  // Wild (uninitialized) pointer
    printf("Dereferencing uninitialized pointer: %d\n", *ptr);
    return 0;
}
```

5. 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 n = 10;               // Declare an integer variable and initialize it
    int *ptr = &n;            // Declare a pointer that points to n
    int **ptr2 = &ptr;        // Declare a pointer to a pointer that points to ptr

    // Print the value using both levels of indirection
    printf("Value of n using ptr: %d\n", *ptr);      // First level of indirection
    printf("Value of n using ptr2: %d\n", **ptr2);   // Second level of indirection

    return 0;
}
```

6. 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>  // For malloc and free

int main() {
    int *arr;       // Pointer to int
    int n, i;

    // Ask user for the number of elements in the array
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Dynamically allocate memory for n integers
    arr = (int *)malloc(n * sizeof(int));

    // Check if memory allocation is successful
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;  // Exit the program if malloc fails
    }

    // Populate the array with values
    for (i = 0; i < n; i++) {
```

```
        arr[i] = i * 10;  // Just an example: set values to multiples of 10
    }

    // Print the values using a pointer
    printf("Array values: ");
    for (i = 0; i < n; i++) {
        printf("%d ", *(arr + i));  // Use pointer arithmetic to print the values
    }
    printf("\n");

    // Free the dynamically allocated memory
    free(arr);

    return 0;
}
```

7. 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 that takes two integers and returns their sum
int sum(int a, int b) {
    return a + b;
}

int main() {
    // Declare a function pointer that points to a function accepting two integers and returning an integer
    int (*func_ptr)(int, int);

    // Point the function pointer to the 'sum' function
    func_ptr = sum;

    // Call the function using the function pointer with different integer values
    int result1 = func_ptr(3, 4); // Calling sum(3, 4)
    int result2 = func_ptr(10, 5); // Calling sum(10, 5)

    // Print the results
    printf("Sum of 3 and 4: %d\n", result1);
    printf("Sum of 10 and 5: %d\n", result2);

    return 0;
}
```

8. 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.

1. Constant Pointer (Pointer to a constant address)

```
#include <stdio.h>

int main() {
    int a = 10, b = 20;

    // Constant pointer to an integer
```

```
    int * const ptr = &a;

    // Cannot change the pointer itself (this would produce an error)
    // ptr = &b;  // Error: assignment of read-only variable 'ptr'

    // Can change the value at the address the pointer is pointing to
    *ptr = 30;  // This is allowed

    printf("a = %d\n", a);  // Output: a = 30
    printf("b = %d\n", b);  // Output: b = 20

    return 0;
}
```

2. Pointer to Constant Data

```
#include <stdio.h>

int main() {
    int a = 10, b = 20;

    // Pointer to constant data
    const int *ptr = &a;

    // Can change the pointer itself to point to another address
    ptr = &b;  // This is allowed

    // Cannot modify the data at the address the pointer is pointing to
    // *ptr = 30;  // Error: assignment of read-only location '*ptr'

    printf("a = %d\n", a);  // Output: a = 10
    printf("b = %d\n", b);  // Output: b = 20

    return 0;
}
```

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 = 5;
    int b = 10;

    // Declare pointers pointing to the variables
    int *ptr1 = &a;
    int *ptr2 = &b;

    // Compare the pointers using relational operators
    if (ptr1 > ptr2) {
        printf("Pointer 1 points to a higher address than Pointer 2\n");
    } else if (ptr1 < ptr2) {
```

```
        printf("Pointer 1 points to a lower address than Pointer 2\n");
    } else {
        printf("Both pointers point to the same address\n");
    }

    return 0;
}
```

WITHOUT USING &

```
#include <stdio.h>

int main()
{
    int value;
    int *ptr=&value;

    printf("enter the value =");
    scanf("%d",ptr);
    printf("\n");
    printf("value= %d",value);


    return 0;
}
```

CONSTANT POINTER

```
#include <stdio.h>

int main()
{
    int a =10;
    int const *ptr=&a;//DATA WILL BE CONSTANT
    *ptr =20;

    return 0;
}


#include <stdio.h>

int main()
{
    int a =10;
    int b=20;
    int const *ptr=&a;
     printf("001 value of ptr =%p \n",ptr);
     ptr=&b;
     printf("002 value of ptr =%p \n",ptr);
     *ptr = 30;
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int a =10;
    int b=20;
    int *const ptr=&a;//DATA CAN BE MODIFIED WITH CONTANT POINTER
     printf("001 value of ptr =%p \n",ptr);
     //ptr=&b;
     //printf("002 value of ptr =%p \n",ptr);
     *ptr = 30;
    printf("a=%d \n",a);
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int a =10;
    int b=20;
    int const *const ptr=&a;//DATA CONSTANT WITH CONTANT POINTER
     printf("001 value of ptr =%p \n",ptr);
     //ptr=&b;
     //printf("002 value of ptr =%p \n",ptr);
     *ptr = 30;
    printf("a=%d \n",a);
    return 0;
}
```

VOID POINTERS(TYPE CASTING)

```
#include <stdio.h>

int main()
{
int a = 20;
float b = 30.5;
char c = 'B';
void *ptr;
ptr = &a;
printf("Value at a = %d \n",*(int *)ptr);
ptr = &b;
printf("Value at b = %f \n",*(float *)ptr);
  ptr = &c;
printf("Value at c = %c \n",*(char *)ptr);
    return 0;
}
```

PROBLEMS

1. WAP 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.

```c
#include <stdio.h>

int main() {
    int n1 = 10;
    int n2 = 20;

    // Declare a constant pointer to an integer and initialize it with the address of n1
    int *const ptr = &n1;

    // Display the value of n1 through the pointer
    printf("Value of n1 before modification: %d\n", *ptr);

    // Modify the value of n1 through the pointer
    *ptr = 30;

    // Display the updated value of n1
    printf("Value of n1 after modification: %d\n", *ptr);

    // Try to reassign the pointer to point to n2 (this will cause a compile-time error)
    // ptr = &n2;  // Uncommenting this line will cause an error

    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.

```c
#include <stdio.h>

int main() {
    // Define a constant integer
    const int n = 10;

    // Define a pointer to the constant integer
    const int *ptr = &n;

    // Try to modify the value pointed to by the pointer
    // Uncommenting the next line will cause a compilation error
    // *ptr = 20;  // Error: cannot modify a constant value

    printf("Value of n: %d\n", n);

    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.

```c
#include <stdio.h>

int main() {
    // Declare a constant integer and initialize it
    int value = 42;

    // Declare a constant pointer to the constant integer
```

```
    const int *const ptr = &value;

    // Try to change the value through the pointer (this will cause a compilation error)
    // *ptr = 10;  // Error: assignment of read-only location

    // Try to change the address stored in the pointer (this will also cause a compilation error)
    // ptr = &value;  // Error: assignment of read-only variable 'ptr'

    // Print the value pointed to by the constant pointer
    printf("Value: %d\n", *ptr);

    // Print the address stored in the pointer
    printf("Address stored in pointer: %p\n", ptr);

    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() {
    // Define multiple integer variables
    int a = 5, b = 10, c = 15;

    // Define a constant pointer that will point to integers
    int *const ptr = &a;  // ptr is a constant pointer, its address can't change

    // Print initial values of the integers
    printf("Before modification:\n");
    printf("a = %d, b = %d, c = %d\n", a, b, c);

    // Dereference the constant pointer to modify the value it points to
    *ptr = 20;  // Modifies 'a' through the pointer

    // You can still modify other variables directly
    b = 30;
    c = 40;

    // Print the modified values
    printf("\nAfter modification:\n");
    printf("a = %d, b = %d, c = %d\n", a, b, c);

    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() {
    // Declare two constant integers
```

```
    const int n1 = 10;
    const int n2 = 10;

    // Declare pointers to hold the addresses of n1 and n2
    const int *ptr1 = &n1;
    const int *ptr2 = &n2;

    // Use decision-making statements (if-else) to compare the values pointed to by ptr1 and ptr2
    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 value = 10;
    int other_value = 20;

    // Declare a constant pointer to int
    const int *ptr = &value; // ptr points to 'value'

    // Check if the pointer is pointing 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 a different value.\n");
    }

    // Change the pointer to point to another value
    ptr = &other_value;

    // Check again after changing the pointer
    if (*ptr == 10) {
        printf("The pointer is now pointing to the value 10.\n");
    } else if (*ptr == 20) {
        printf("The pointer is now pointing to the value 20.\n");
    } else {
        printf("The pointer is now pointing to a different 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.

```c
#include <stdio.h>

int main() {
    // Declare two integer variables
    int n1 = 10;
    int n2 = 20;

    // Declare constant pointers pointing to the variables
    const int *ptr1 = &n1;
    const int *ptr2 = &n2;

    // Compare the addresses using relational operators
    if (ptr1 < ptr2) {
        printf("ptr1 points to a lower address than ptr2.\n");
    } else if (ptr1 > ptr2) {
        printf("ptr1 points to a higher address than ptr2.\n");
    } else {
        printf("ptr1 and ptr2 point to the same address.\n");
    }

    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.

```c
#include <stdio.h>

int main() {
    // Define some variables
    int a = 10, b = 20, c = 30, d = 40;

    // Declare a constant pointer to an integer
    int * const ptr = &a; // Constant pointer pointing to variable a

    // Use a loop to iterate through the variables
    // We need to manually change the pointer’s target for each iteration
    // Since we can’t reassign the constant pointer, we use an array of pointers
    int *arr[4] = {&a, &b, &c, &d};  // Array of pointers

    // Loop through the array of pointers
    for (int i = 0; i < 4; i++) {
        // Print the value the constant pointer is pointing to
        printf("Value of variable %d: %d\n", i + 1, *arr[i]);
    }

    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() {
    // Declare several integer variables
    int var1 = 10;
    int var2 = 20;
    int var3 = 30;
    int var4 = 40;

    // Declare a constant pointer to the first variable
    int *const ptr = &var1;

    // Using pointer arithmetic to iterate over the variables
    // Print values by incrementing the pointer with pointer arithmetic
    printf("Using pointer arithmetic with a constant pointer:\n");

    printf("Value of var1: %d\n", *ptr);       // Prints value of var1

    // Move to the next variable using pointer arithmetic
    ptr++;  // Increment the pointer to point to the next variable
    printf("Value of var2: %d\n", *ptr);       // Prints value of var2

    // Move to the next variable
    ptr++;  // Increment the pointer to point to the next variable
    printf("Value of var3: %d\n", *ptr);       // Prints value of var3

    // Move to the next variable
    ptr++;  // Increment the pointer to point to the next variable
    printf("Value of var4: %d\n", *ptr);       // Prints value of var4

    return 0;
}
```

CALL REFERENCE

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

     int sumvalue=0;
     sumvalue=sum(&a,&b);

     printf("sumvalue = %d \n",sumvalue);
    return 0;
}
int sum(int *p,int *q){
    *p=20;
    *q=30;
    int sum=0;
    sum=*p+*q;
    return sum;
}
```

SET OF PROGRAMS
----

1. 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 to calculate efficiency
void calculateEfficiency(float* inputPower, float* outputPower, float* efficiency) {
    // Ensure input power is not zero to avoid division by zero
    if (*inputPower != 0) {
        *efficiency = (*outputPower / *inputPower) * 100;
    } else {
        *efficiency = 0; // Efficiency is zero if input power is zero
        printf("Input power is zero, efficiency cannot be calculated.\n");
    }
}

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

    // Taking input from the user for input power and output power
    printf("Enter input power (in watts): ");
    scanf("%f", &inputPower);
    printf("Enter output power (in watts): ");
    scanf("%f", &outputPower);

    // Call the function to calculate efficiency
    calculateEfficiency(&inputPower, &outputPower, &efficiency);

    // Output the efficiency
    printf("Efficiency: %.2f%%\n", efficiency);

    return 0;
}
```

2. 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 to adjust speed dynamically
void adjustSpeed(float *speed, float adjustment) {
    // Adjust the speed
    *speed += adjustment;

    // Ensure speed stays within the range 0 to 100
    if (*speed > 100) {
        *speed = 100;
```

```
    } else if (*speed < 0) {
        *speed = 0;
    }
}

int main() {
    float currentSpeed = 50.0;  // Example current speed
    float adjustment = -10.0;  // Example adjustment value

    printf("Initial speed: %.2f\n", currentSpeed);

    // Adjust the speed
    adjustSpeed(&currentSpeed, adjustment);

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

    return 0;
}
```

3. 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 updateInventory(int *inventory, int size, int *changes) {
    for (int i = 0; i < size; i++) {
        inventory[i] += changes[i];
        if (inventory[i] < 0) {
            inventory[i] = 0;  // Prevent inventory from going below zero
        }
    }
}

int main() {
    // Sample inventory levels of raw materials (initial inventory)
    int inventory[] = {50, 100, 75, 200};
    // Sample changes (production/consumption of raw materials)
    // Positive values are for production, negative values for consumption
    int changes[] = {-30, 20, -80, -250};

    // Get the number of raw materials
    int size = sizeof(inventory) / sizeof(inventory[0]);

    // Update inventory based on changes
    updateInventory(inventory, size, changes);

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

```c
    return 0;
}
```

4. 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.

```c
#include <stdio.h>

// Define workspace boundaries
#define MIN_X 0
#define MAX_X 100
#define MIN_Y 0
#define MAX_Y 100
#define MIN_Z 0
#define MAX_Z 100

void update_coordinates(int *x, int *y, int *z, int delta_x, int delta_y, int delta_z) {
    // Calculate new coordinates
    int new_x = *x + delta_x;
    int new_y = *y + delta_y;
    int new_z = *z + delta_z;

    // Validate and update x coordinate
    if (new_x >= MIN_X && new_x <= MAX_X) {
        *x = new_x;
    } else {
        printf("X coordinate out of bounds. Staying at: %d\n", *x);
    }

    // Validate and update y coordinate
    if (new_y >= MIN_Y && new_y <= MAX_Y) {
        *y = new_y;
    } else {
        printf("Y coordinate out of bounds. Staying at: %d\n", *y);
    }

    // Validate and update z coordinate
    if (new_z >= MIN_Z && new_z <= MAX_Z) {
        *z = new_z;
    } else {
        printf("Z coordinate out of bounds. Staying at: %d\n", *z);
    }
}

int main() {
    // Initialize coordinates
    int x = 50, y = 50, z = 50;

    printf("Initial coordinates: x=%d, y=%d, z=%d\n", x, y, z);

    // Define delta values
    int delta_x = 20, delta_y = -30, delta_z = 10;
```

```
    // Call the function to update coordinates
    update_coordinates(&x, &y, &z, delta_x, delta_y, delta_z);

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

    return 0;
}
```

5. 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>

// Safety limits
#define SAFETY_MIN 0.0
#define SAFETY_MAX 100.0

// Function to adjust the temperature within the given range using pointers
void adjust_temperature(float *current_temp, float min_range, float max_range) {
    if (*current_temp < min_range) {
        // If the temperature is lower than the minimum range, set it to the minimum range
        *current_temp = min_range;
    }
    if (*current_temp > max_range) {
        // If the temperature is higher than the maximum range, set it to the maximum range
        *current_temp = max_range;
    }

    // Ensure the temperature doesn't exceed safety limits
    if (*current_temp < SAFETY_MIN) {
        *current_temp = SAFETY_MIN;
    } else if (*current_temp > SAFETY_MAX) {
        *current_temp = SAFETY_MAX;
    }
}

int main() {
    float current_temp;
    float min_range, max_range;

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

    printf("Enter the minimum desired range: ");
    scanf("%f", &min_range);

    printf("Enter the maximum desired range: ");
    scanf("%f", &max_range);

    // Call the function to adjust the temperature
    adjust_temperature(&current_temp, min_range, max_range);
```

```
    // Output the adjusted temperature
    printf("The adjusted temperature is: %.2f\n", current_temp);

    return 0;
}
```

6. 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
void updateRemainingLife(int *currentUsage, int maxLifeSpan, int *remainingLife) {
    // Calculate the remaining life
    *remainingLife = maxLifeSpan - *currentUsage;

    // Ensure that the remaining life cannot go below zero
    if (*remainingLife < 0) {
        *remainingLife = 0;
    }
}

int main() {
    int currentUsage = 100;    // Current tool usage hours
    int maxLifeSpan = 200;      // Maximum life span of the tool in hours
    int remainingLife = maxLifeSpan; // Initialize remaining life with maximum life span

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

    // Output the updated remaining life
    printf("Updated Remaining Life: %d hours\n", remainingLife);

    return 0;
}
```

7. 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 *weights, int size) {
    float totalWeight = 0.0;

    // Check each weight in the array
    for (int i = 0; i < size; i++) {
        if (weights[i] < 0) {
            printf("Error: Negative weight encountered at index %d.\n", i);
            return -1; // Return -1 to indicate an error
        }
```

```
        totalWeight += weights[i]; // Add valid weights to the total
    }

    return totalWeight;
}

int main() {
    // Example usage
    float weights[] = {10.5, 15.2, 8.3, 12.7}; // Array of material weights
    int size = sizeof(weights) / sizeof(weights[0]);

    float totalWeight = calculateTotalWeight(weights, size);

    if (totalWeight != -1) {
        printf("Total weight: %.2f\n", totalWeight);
    }

    return 0;
}
```

8. 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>

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

int main() {
    // Initial voltage and current values
    float voltage = 220.0;
    float current = 5.0;

    printf("Initial Configuration - Voltage: %.2fV, Current: %.2fA\n", voltage, current);

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

    printf("Updated Configuration - Voltage: %.2fV, Current: %.2fA\n", voltage, current);

    return 0;
}

void update_machine_configuration(float* voltage, float* current) {
    // Specify the valid operating ranges
    float min_voltage = 200.0;  // Minimum allowable voltage
    float max_voltage = 240.0;  // Maximum allowable voltage
    float min_current = 3.0;    // Minimum allowable current
    float max_current = 10.0;   // Maximum allowable current

    // Update voltage within the valid range
    if (*voltage < min_voltage) {
        *voltage = min_voltage;
```

```c
        printf("Voltage too low, adjusted to %.2fV\n", *voltage);
    } else if (*voltage > max_voltage) {
        *voltage = max_voltage;
        printf("Voltage too high, adjusted to %.2fV\n", *voltage);
    }

    // Update current within the valid range
    if (*current < min_current) {
        *current = min_current;
        printf("Current too low, adjusted to %.2fA\n", *current);
    } else if (*current > max_current) {
        *current = max_current;
        printf("Current too high, adjusted to %.2fA\n", *current);
    }
}
```

9. 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 using pointers
void calculate_defect_rate(int *total_products, int *defective_products) {
    if (*total_products > *defective_products) {
        // Calculate defect rate
        float defect_rate = ((float)(*defective_products) / *total_products) * 100;
        // Print the defect rate
        printf("Defect rate: %.2f%%\n", defect_rate);
    } else {
        printf("Error: Total products should be greater than defective products.\n");
    }
}

int main() {
    int total, defective;

    // Take input from the user
    printf("Enter total number of products: ");
    scanf("%d", &total);
    printf("Enter number of defective products: ");
    scanf("%d", &defective);

    // Ensure the total products is greater than defective products
    if (total > defective) {
        // Call the function to calculate defect rate
        calculate_defect_rate(&total,  &defective);
    } else {
        printf("Invalid input. Total products must be greater than defective products.\n");
    }

    return 0;
}
```

10. 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 adjustTimingIntervals(float *intervals, int size) {
    // Ensure the timing intervals are positive
    for (int i = 0; i < size; i++) {
        if (*(intervals + i) <= 0) {
            printf("Invalid timing interval at index %d: %.2f. Adjusting to a small positive value.\n", i, *(intervals
+ i));
            *(intervals + i) = 0.1; // Adjust invalid (non-positive) intervals to a small positive value
        }
    }

    // Example adjustment logic: Increase intervals by 10%
    for (int i = 0; i < size; i++) {
        *(intervals + i) *= 1.1; // Increase by 10%
    }
}

void printIntervals(float *intervals, int size) {
    for (int i = 0; i < size; i++) {
        printf("Interval %d: %.2f\n", i, *(intervals + i));
    }
}

int main() {
    // Example timing intervals
    float intervals[] = { 2.5, 3.0, -1.0, 4.5, 0.0, 6.0 };
    int size = sizeof(intervals) / sizeof(intervals[0]);

    printf("Original timing intervals:\n");
    printIntervals(intervals, size);

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

    printf("\nAdjusted timing intervals:\n");
    printIntervals(intervals, size);

    return 0;
}
```

11. 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>
#include <float.h>  // For FLT_MAX and FLT_MIN

// Function to update coordinates with bounds check
```

```c
void update_coordinates(float *x, float *y, float *z) {
    // Define the bounds for machine limits (could be customized)
    float lower_limit = -FLT_MAX;
    float upper_limit = FLT_MAX;

    // Ensure the updated coordinates are within the machine's limits
    if (*x < lower_limit) *x = lower_limit;
    else if (*x > upper_limit) *x = upper_limit;

    if (*y < lower_limit) *y = lower_limit;
    else if (*y > upper_limit) *y = upper_limit;

    if (*z < lower_limit) *z = lower_limit;
    else if (*z > upper_limit) *z = upper_limit;
}

int main() {
    // Example usage
    float x = 10.5, y = -3000.5, z = 0.0;

    printf("Before update: x = %.2f, y = %.2f, z = %.2f\n", x, y, z);

    // Update the coordinates
    update_coordinates(&x, &y, &z);

    printf("After update: x = %.2f, y = %.2f, z = %.2f\n", x, y, z);

    return 0;
}
```

12. 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>

float calculate_total_energy(float *energy_data, int size) {
    float total_energy = 0.0;

    // Validate and calculate total energy usage
    for (int i = 0; i < size; i++) {
        if (energy_data[i] < 0) {
            printf("Error: Negative energy usage detected at index %d\n", i);
            return -1.0; // Return an error value
        }
        total_energy += energy_data[i];
    }
    return total_energy;
}

int main() {
    float energy_usage[] = {10.5, 20.0, 15.3, 5.0, 7.2};
    int size = sizeof(energy_usage) / sizeof(energy_usage[0]);
```

```c
    float total_energy = calculate_total_energy(energy_usage, size);
    if (total_energy != -1.0) {
        printf("Total energy consumed: %.2f\n", total_energy);
    } else {
        printf("Energy usage contains invalid values.\n");
    }

    return 0;
}
```

13. 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

```c
#include <stdio.h>

// Function to adjust the production rate
void adjustProductionRate(int *currentRate, float adjustmentFactor, int minLimit, int maxLimit) {
    // Adjust the production rate by applying the adjustment factor
    *currentRate = (int)(*currentRate * adjustmentFactor);

    // Ensure the production rate stays within permissible limits
    if (*currentRate < minLimit) {
        *currentRate = minLimit;  // Set to minimum limit if it goes below
    } else if (*currentRate > maxLimit) {
        *currentRate = maxLimit;  // Set to maximum limit if it exceeds
    }
}

int main() {
    int productionRate = 500;  // Example initial production rate
    float adjustmentFactor = 1.2;  // Example adjustment factor (20% increase)
    int minLimit = 0;  // Minimum allowable production rate
    int maxLimit = 1000;  // Maximum allowable production rate

    printf("Current production rate: %d\n", productionRate);

    // Call function to adjust the production rate
    adjustProductionRate(&productionRate, adjustmentFactor, minLimit, maxLimit);

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

    return 0;
}
```

14. 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.

```c
#include <stdio.h>

// Function to compare two dates
int compareDates(int year1, int month1, int day1, int year2, int month2, int day2) {
```

```
    if (year1 < year2) {
        return -1;  // first date is earlier
    } else if (year1 > year2) {
        return 1;  // first date is later
    }
    if (month1 < month2) {
        return -1;  // first date is earlier
    } else if (month1 > month2) {
        return 1;  // first date is later
    }
    if (day1 < day2) {
        return -1;  // first date is earlier
    } else if (day1 > day2) {
        return 1;   // first date is later
    }
    return 0;   // dates are equal
}

// Function to update the maintenance schedule
void updateMaintenanceSchedule(int *currentYear, int *currentMonth, int *currentDay,
                               int *nextYear, int *nextMonth, int *nextDay) {
    // Compare current and next maintenance dates
    if (compareDates(*currentYear, *currentMonth, *currentDay, *nextYear, *nextMonth, *nextDay) >= 0) {
        printf("Next maintenance date must be later than the current maintenance date.\n");
        printf("Please enter a new next maintenance date (year, month, day):\n");

        // Get a new valid next date from the user
        do {
            printf("Year: ");
            scanf("%d", nextYear);

            printf("Month: ");
            scanf("%d", nextMonth);

            printf("Day: ");
            scanf("%d", nextDay);

            // Check if the new date is valid
            if (compareDates(*currentYear, *currentMonth, *currentDay, *nextYear, *nextMonth, *nextDay) >=
0) {
                printf("The next maintenance date cannot be earlier than or equal to the current date.\n");
            }
        } while (compareDates(*currentYear, *currentMonth, *currentDay, *nextYear, *nextMonth, *nextDay)
>= 0);
    } else {
        printf("Maintenance schedule updated successfully.\n");
    }
}

// Function to print the date
void printDate(int year, int month, int day) {
    printf("%d-%02d-%02d\n", year, month, day);
}

int main() {
```

```
    int currentYear, currentMonth, currentDay;
    int nextYear, nextMonth, nextDay;

    // Input the current maintenance date
    printf("Enter current maintenance date (year, month, day):\n");
    printf("Year: ");
    scanf("%d", &currentYear);
    printf("Month: ");
    scanf("%d", &currentMonth);
    printf("Day: ");
    scanf("%d", &currentDay);

    // Input the next maintenance date
    printf("Enter next maintenance date (year, month, day):\n");
    printf("Year: ");
    scanf("%d", &nextYear);
    printf("Month: ");
    scanf("%d", &nextMonth);
    printf("Day: ");
    scanf("%d", &nextDay);

    // Update the maintenance schedule
    updateMaintenanceSchedule(&currentYear, &currentMonth, &currentDay,
                              &nextYear, &nextMonth, &nextDay);

    // Display the updated maintenance schedule
    printf("Current Maintenance Date: ");
    printDate(currentYear, currentMonth, currentDay);

    printf("Next Maintenance Date: ");
    printDate(nextYear, nextMonth, nextDay);

    return 0;
}
```

15. 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 update_quality_metrics(int *quality_scores, int batch_size) {
    for (int i = 0; i < batch_size; i++) {
        // Ensure that the quality score stays within the range [0, 100]
        if (quality_scores[i] < 0) {
            quality_scores[i] = 0;
        } else if (quality_scores[i] > 100) {
            quality_scores[i] = 100;
        }
    }
}

int main() {
    int quality_scores[] = {95, 105, -5, 80, 50};
```

```
    int batch_size = sizeof(quality_scores) / sizeof(quality_scores[0]);

    // Update the quality metrics
    update_quality_metrics(quality_scores, batch_size);

    // Print updated quality scores
    for (int i = 0; i < batch_size; i++) {
        printf("Product %d: Quality score = %d\n", i + 1, quality_scores[i]);
    }

    return 0;
}
```

16. 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 adjust_space_allocation(int* space_used, int num_sections, int total_capacity) {
    int current_total = 0;

    // Calculate the current total space used
    for (int i = 0; i < num_sections; i++) {
        current_total += space_used[i];
    }

    // Check if the total space used exceeds the warehouse capacity
    if (current_total > total_capacity) {
        int excess_space = current_total - total_capacity;

        // Adjust space allocations by redistributing or reducing space
        for (int i = 0; i < num_sections; i++) {
            if (space_used[i] > excess_space) {
                space_used[i] -= excess_space;
                break;
            } else {
                excess_space -= space_used[i];
                space_used[i] = 0;
            }
        }
    }

    // Print the updated space allocation
    printf("Updated space allocation:\n");
    for (int i = 0; i < num_sections; i++) {
        printf("Section %d: %d units\n", i + 1, space_used[i]);
    }
}

int main() {
    int space_used[] = {50, 100, 30, 60}; // Example space used for each section
    int num_sections = sizeof(space_used) / sizeof(space_used[0]);
    int total_capacity = 200; // Warehouse total capacity
```

```
    adjust_space_allocation(space_used, num_sections, total_capacity);
    return 0;
}
```

17. 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 safe operating limits
#define MIN_SPEED 0.0f
#define MAX_SPEED 100.0f
#define MIN_TENSION 5.0f
#define MAX_TENSION 50.0f

// Function to update machine settings
void updateSettings(float *speed, float *tension) {
    // Validate speed
    if (*speed < MIN_SPEED) {
        printf("Warning: Speed is too low. Setting speed to %.2f.\n", MIN_SPEED);
        *speed = MIN_SPEED;
    } else if (*speed > MAX_SPEED) {
        printf("Warning: Speed is too high. Setting speed to %.2f.\n", MAX_SPEED);
        *speed = MAX_SPEED;
    }

    // Validate wrap tension
    if (*tension < MIN_TENSION) {
        printf("Warning: Tension is too low. Setting tension to %.2f.\n", MIN_TENSION);
        *tension = MIN_TENSION;
    } else if (*tension > MAX_TENSION) {
        printf("Warning: Tension is too high. Setting tension to %.2f.\n", MAX_TENSION);
        *tension = MAX_TENSION;
    }

    // Display updated settings
    printf("Updated Settings:\n");
    printf("Speed: %.2f\n", *speed);
    printf("Wrap Tension: %.2f\n", *tension);
}

int main() {
    // Sample machine settings
    float speed = 120.0f;  // Invalid speed
    float tension = 60.0f; // Invalid tension

    // Call the function to update the settings
    updateSettings(&speed, &tension);

    return 0;
}
```

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

```c
#include <stdio.h>

// Function to adjust the temperature
void adjust_temperature(float *temp, float min_temp, float max_temp) {
    // Check if the temperature is less than the minimum range
    if (*temp < min_temp) {
        *temp = min_temp;
    }
    // Check if the temperature is greater than the maximum range
    else if (*temp > max_temp) {
        *temp = max_temp;
    }
}

int main() {
    float temperature;
    float min_temp, max_temp;

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

    printf("Enter the minimum temperature range: ");
    scanf("%f", &min_temp);

    printf("Enter the maximum temperature range: ");
    scanf("%f", &max_temp);

    // Adjust the temperature using the adjust_temperature function
    adjust_temperature(&temperature, min_temp, max_temp);

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

    return 0;
}
```

19. 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.

```c
#include <stdio.h>

// Function to update scrap count based on a delta value
// It uses pointers to modify the original scrap counts
void updateScrapCount(int *scrapCounts, int size, int *deltas) {
    for (int i = 0; i < size; i++) {
        scrapCounts[i] += deltas[i]; // Update the scrap count with the delta
        if (scrapCounts[i] < 0) {
```

```
            scrapCounts[i] = 0; // Ensure non-negative scrap count
        }
    }
}

int main() {
    int scrapCounts[] = {100, 200, 150, 80, 120}; // Example scrap counts for materials
    int deltas[] = {-50, -300, 20, -100, 40}; // Example changes to the scrap counts
    int size = sizeof(scrapCounts) / sizeof(scrapCounts[0]); // Calculate size of the array

    printf("Original scrap counts:\n");
    for (int i = 0; i < size; i++) {
        printf("Material %d: %d\n", i + 1, scrapCounts[i]);
    }

    // Call the update function to modify the scrap counts
    updateScrapCount(scrapCounts, size, deltas);

    printf("\nUpdated scrap counts:\n");
    for (int i = 0; i < size; i++) {
        printf("Material %d: %d\n", i + 1, scrapCounts[i]);
    }

    return 0;
}
```

20. 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>
#include <stdlib.h>

// Function to calculate the total production and average production
void updatePerformanceMetrics(int *data, int numShifts, float *totalProduction, float *averageProduction)
{
    if (data == NULL || numShifts <= 0 || totalProduction == NULL || averageProduction == NULL) {
        printf("Invalid input data.\n");
        return;
    }

    *totalProduction = 0;  // Initialize total production to 0

    // Loop through the array of production data for each shift and calculate total production
    for (int i = 0; i < numShifts; i++) {
        if (data[i] < 0) {
            printf("Invalid production data at shift %d: Production cannot be negative.\n", i + 1);
            return;
        }
        *totalProduction += data[i];
    }

    // Calculate average production
    *averageProduction = *totalProduction / numShifts;
```

```
}

// Function to display the performance metrics
void displayPerformanceMetrics(float totalProduction, float
    averageProduction) { printf("Total Production: %.2f\n",
    totalProduction);
    printf("Average Production per Shift: %.2f\n", averageProduction);
}

int main() {
    int numShifts;

    // Request number of shifts
    (positive integer)
    printf("Enter the number of
    shifts: ");
    if (scanf("%d", &numShifts) != 1 ||
        numShifts <= 0) { printf("Invalid
        number of shifts.\n");
        return 1;
    }

    // Dynamically allocate memory for production data based
    on the number of shifts int *productionData = (int
    *)malloc(numShifts * sizeof(int));
    if (productionData
        == NULL) {
        printf("Memory
        allocation
        failed.\n");
        return 1;
    }

    // Input the production data
    for each shift printf("Enter
    the production data for
    each shift:\n"); for (int i = 0;
    i < numShifts; i++) {
        printf("Shift %d production: ", i + 1);
        if (scanf("%d", &productionData[i]) != 1 ||
            productionData[i] < 0) { printf("Invalid
            input for shift %d.\n", i + 1);
            free(productionData);
            return 1;
        }
    }

    // Declare variables to store
    the performance metrics float
    totalProduction,
```

```
    averageProduction;

    // Calculate the performance metrics using the data
    updatePerformanceMetrics(productionData, numShifts,
    &totalProduction, &averageProduction);

    // Display the updated performance
    metrics
    displayPerformanceMetrics(totalProductio
    n, averageProduction);

    // Free the
    dynamically
    allocated memory
    free(productionDa
    ta);

    return 0;
}
```