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 num = 10;

   int *ptr = &num;

   printf("Value of num using pointer: %d\n", *ptr);

   return 0;

}
```

Output:

Value of num using pointer:10

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

```
#include <stdio.h>

int main() {

   float num = 5.5;

   float *ptr = &num;

   printf("Original value: %.2f\n", num);

   *ptr = 7.5;

   printf("Modified value: %.2f\n", num);

   return 0;

}
```

Output:

Original value: 5.50

Modified value: 7.50

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 sum(int *arr, int size) {

   int total = 0;
```

```
    for (int i = 0; i < size; i++) {
        total += *(arr + i);
    }
    return total;
}
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Sum of array elements: %d\n", sum(arr, size));
    return 0;
}
```

Output:

Sum of array elements: 15

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;
    if (ptr == NULL) {
        printf("Pointer is null cannot dereference.\n");
    } else {
        printf("Value at pointer: %d\n", *ptr);
    }
    return 0;
}
```

Output:

Pointer is null, cannot dereference.

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() {
```

```c
    int *ptr;
    printf("Dereferencing wild pointer: %d\n", *ptr);
    return 0;
}
```

Output:

Dereferencing wild pointer: 805212160

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.

```c
#include <stdio.h>
int main() {
    int num = 20;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    printf("Value of num using pointer to pointer: %d\n", **ptr2);
    return 0;
}
```

Output:

Value of num using pointer to pointer: 20

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.

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *arr;
    int size = 5;
    arr = (int *)malloc(size * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
```

```
    for (int i = 0; i < size; i++) {
        arr[i] = i + 1;
        printf("%d  ", arr[i]);
    }
    printf("\n");
    free(arr);
    return 0;
}
```

Output:

1 2 3 4 5

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

```
#include <stdio.h>
int add(int a, int b) {
    return a + b;
}
int main() {
    int (*func_ptr)(int,  int) = add;
        int result = func_ptr(5,  7);
    printf("Result  of addition:  %d\n", result);
    return 0;
}
```

Output:

Result of addition: 12

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 num1 = 10;
    int num2 = 20;
```

```
    int *const ptr1 = &num1;

    const int *ptr2 = &num1;

    printf("Value pointed by ptr2: %d\n", *ptr2);

    return 0;

}
```

Output:

Value pointed by ptr2: 10

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 b = 20;

    int *ptr1 = &a;

    int *ptr2 = &b;

    if (ptr1 < ptr2) {

        printf("Pointer ptr1 points to a smaller address than ptr2.\n");

    } else if (ptr1 > ptr2) {

        printf("Pointer ptr1 points to a larger address than ptr2.\n");

    } else {

        printf("Pointers ptr1 and ptr2 point to the same address.\n");

    }

    return 0;

}
```

Output:

Pointer ptr1 points to a larger address than ptr2.

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 num = 10;

   int num2 = 20;

   // Declare a constant pointer to integer

   int *const ptr = &num;

   *ptr = 25; // This is allowed

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

   return 0;

}
```

Output:

Value of num after modification: 25

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 num = 10;

   // Pointer to constant integer

   const int *ptr = &num;

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

   return 0;

}
```

Output:

Value pointed by ptr: 10

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 num = 10;

   // Constant pointer to constant integer

   const int *const ptr = &num;

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

```
    return 0;
}
```

Output:

Value pointed by ptr: 10

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

```
#include <stdio.h>

int main() {

    int a = 10, b = 20, c = 30;  // Declare integer variables

    // Declare a constant pointer to an integer

    int* const ptr = &a;

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

    *ptr = 100;

    printf("After modifying a: a = %d\n", a);

    int* ptr_array[] = {&a, &b, &c};

    for (int i = 0; i < 3; i++) {

        *ptr_array[i] = 200 * (i + 1);

    }

    printf("After modifying a, b, and c:\n");

    printf("a = %d, b = %d, c = %d\n", a, b, c);

    return 0;

}
```

Output:

Initial values: a = 10, b = 20, c = 30

After modifying a: a = 100

After modifying a, b, and c:

a = 200, b = 400, c = 600

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, num2 = 20;

   const int *ptr1 = &num1;

   const int *ptr2 = &num2;

   if (*ptr1 == *ptr2) {

      printf("pointers are equal.\n");

   } else {

      printf("pointers are not equal.\n");

   }

   return 0;

}
```

Output:

The values pointed to by the pointers are 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 num = 10;

   int *const ptr = &num;

   if (*ptr == 10) {

      printf("Pointer is pointing to the value 10.\n");

   } else {

      printf("Pointer is not pointing to the value 10.\n");

   }

   return 0;

}
```

Output:

Pointer is pointing to the value 10.

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

```
#include <stdio.h>

int main() {
    int num1 = 10, num2 = 20;
    const int *const ptr1 = &num1;
    const int *const ptr2 = &num2;
    if (ptr1 < ptr2) {
        printf("ptr1 points is lower address than ptr2.\n");
    } else if (ptr1 > ptr2) {
        printf("ptr1 points is higher address than ptr2.\n");
    } else {
        printf("ptr1 and ptr2 point to the same address.\n");
    }
    return 0;
}
```

Output:

ptr1 points is higher address than ptr2.

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()
{
    int a = 10, b = 20, c = 30, d = 40;
    int *const p[] = {&a, &b, &c, &d};
    int i;
    for (i = 0; i < 4; i++)
    {
        int *const q = p[i];
        printf("Value of variable %d: %d\n", i + 1, *q);
    }
    return 0;
}
```

O/P:

Value of variable 1: 10

Value of variable 2: 20

Value of variable 3: 30

Value of variable 4: 40

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

```
#include <stdio.h>

int main()

{

  // Declare multiple variables

   int a = 10, b = 20, c = 30, d = 40;

  int *p[] = {&a, &b, &c, &d};

  int **ptr = p;

  int i;

  for (i = 0; i < 4; i++)

  {

     printf("Value of variable %d: %d\n", i + 1, **ptr);

     ptr++;

  }

  return 0;

}
```

Output:

Value of variable 1: 10

Value of variable 2: 20

Value of variable 3: 30

Value of variable 4: 40

1. Machine Efficiency Calculation

Requirements:

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

Sol: #include <stdio.h>

```
// Function to calculate machine efficiency
void calculateEfficiency(float *inputPower, float *outputPower, float *efficiency) {
  if (*inputPower != 0) {
    *efficiency = (*outputPower / *inputPower) * 100;
  } else {
    *efficiency = 0; // To avoid division by zero
  }
}

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

  // Input from the user
  printf("Enter the input power of the machine (in watts): ");
  scanf("%f", &inputPower);

  printf("Enter the output power of the machine (in watts): ");
  scanf("%f", &outputPower);

  // Calculate efficiency
  calculateEfficiency(&inputPower, &outputPower, &efficiency);
```

```
    // Display the result
    if (efficiency != 0) {
        printf("The efficiency of the machine is: %.2f%%\n", efficiency);
    } else {
        printf("Invalid input: Input power cannot be zero.\n");
    }

    return 0;
}
```

O/p:

Enter the input power of the machine (in watts): 300

Enter the output power of the machine (in watts): 200

The efficiency of the machine is: 66.67%

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).

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

// Function to adjust conveyor belt speed
void adjustSpeed(float *currentSpeed, float *adjustment) {
    *currentSpeed += *adjustment;
    if (*currentSpeed > 100) {
        *currentSpeed = 100; // Cap at maximum allowable speed
    } else if (*currentSpeed < 0) {
        *currentSpeed = 0; // Ensure speed doesn't go below zero
    }
}
```

```c
int main() {
    float currentSpeed, adjustment;

    // Input for current speed and adjustment value
    printf("Enter the current speed of the conveyor belt (0 to 100 units): ");
    scanf("%f", &currentSpeed);

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

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

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

    return 0;
}
```

O/p: Enter the current speed of the conveyor belt (0 to 100 units): 95

Enter the adjustment value for the speed: 34

The updated speed of the conveyor belt is: 100.00 units

3. Inventory Management

Requirements:

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

Sol: #include <stdio.h>

```
// Function to update inventory levels
void updateInventory(int *inventory, int size, int *change) {
    for (int i = 0; i < size; i++) {
        inventory[i] += change[i];
        if (inventory[i] < 0) {
            inventory[i] = 0; // Ensure no inventory level drops below zero
        }
    }
}

int main() {
    int inventory[] = {10, 20, 15, 5}; // Example current inventory levels
    int size = sizeof(inventory) / sizeof(inventory[0]);

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

    int change[] = {-5, -10, 20, -8}; // Example changes (negative for consumption, positive for
production)

    updateInventory(inventory, size, change);

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

    return 0;
```

```
}
```

O/p: Current inventory levels:

Item 1: 10

Item 2: 20

Item 3: 15

Item 4: 5

Updated inventory levels:

Item 1: 5

Item 2: 10

Item 3: 35

Item 4: 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.

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

// Function to update coordinates with given deltas
void 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
   *x += deltaX;
   *y += deltaY;
   *z += deltaZ;

   // Ensure the coordinates are within workspace boundaries
   if (*x < minX) *x = minX;
   if (*x > maxX) *x = maxX;
```

```
    if (*y < minY) *y = minY;
    if (*y > maxY) *y = maxY;
    if (*z < minZ) *z = minZ;
    if (*z > maxZ) *z = maxZ;
}

int main() {
    int x = 5, y = 10, z = 15;
    int deltaX, deltaY, deltaZ;

    // Workspace boundaries
    int minX = 0, maxX = 20, minY = 0, maxY = 20, minZ = 0, maxZ = 20;

    // Input delta values
    printf("Enter delta values for X, Y, Z: ");
    scanf("%d %d %d", &deltaX, &deltaY, &deltaZ);

    // Update coordinates
    updateCoordinates(&x, &y, &z, deltaX, deltaY, deltaZ, minX, maxX, minY, maxY, minZ, maxZ);

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

    return 0;
}
```

O/p:

Enter delta values for X, Y, Z: 3 -10 5

Updated Coordinates: X = 8, Y = 0, Z = 20

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.

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

// Function to adjust temperature within the safety limits
void adjustTemperature(float *currentTemp, float minTemp, float maxTemp) {
   // Ensure the temperature is within the safety limits
   if (*currentTemp < minTemp) {
      *currentTemp = minTemp;
   } else if (*currentTemp > maxTemp) {
      *currentTemp = maxTemp;
   }
}

int main() {
   float currentTemp;
   float minTemp = 20.0;  // Safety lower limit
   float maxTemp = 100.0; // Safety upper limit

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

   // Adjust temperature within the safety range
   adjustTemperature(&currentTemp, minTemp, maxTemp);

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

```
    return 0;
}
```

O/p: Enter current temperature: 100

Adjusted Temperature: 100.00

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.

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

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

    // Ensure remaining life does not go below zero
    if (remainingLife < 0) {
        remainingLife = 0;
    }

    // Update the pointer with remaining life
    *currentUsage = remainingLife;
}

int main() {
    int currentUsage, maxLifeSpan;

    // Input current usage and maximum life span
```

```
    printf("Enter current tool usage hours: ");
    scanf("%d", &currentUsage);
    printf("Enter maximum life span in hours: ");
    scanf("%d", &maxLifeSpan);

    // Update remaining life
    updateRemainingLife(&currentUsage, maxLifeSpan);

    // Output remaining life
    printf("Remaining tool life: %d hours\n", currentUsage);

    return 0;
}
```

O/p:

Enter current tool usage hours: 80

Enter maximum life span in hours: 120

Remaining tool life: 40 hours

7. Material Weight Calculator

Requirements:

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

Sol:

```
#include <stdio.h>

// Function to calculate total weight
void calculateTotalWeight(float *weights, int size, float *totalWeight) {
    *totalWeight = 0.0; // Initialize total weight to 0

    for (int i = 0; i < size; i++) {
```

```
        if (weights[i] < 0) {
            printf("Error: Negative weight detected at index %d. Setting to 0.\n", i);
            weights[i] = 0;  // Set negative weight to 0
        }
        *totalWeight += weights[i];  // Add the weight to total weight
    }
}

int main() {
    int n;
    float totalWeight;

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

    // Declare array to store material weights
    float weights[n];

    // Input weights of materials
    for (int i = 0; i < n; i++) {
        printf("Enter weight of material %d (in kg): ", i + 1);
        scanf("%f", &weights[i]);

        // Validate weight value
        if (weights[i] < 0) {
            printf("Error: Weight cannot be negative. Setting to 0.\n");
            weights[i] = 0;  // Set negative values to 0
        }
    }
```

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

    // Output total weight
    printf("Total weight of materials: %.2f kg\n", totalWeight);

    return 0;
}
```

O/p: Enter the number of materials: 5

Enter weight of material 1 (in kg): 30

Enter weight of material 2 (in kg): 50

Enter weight of material 3 (in kg): 40

Enter weight of material 4 (in kg): 25

Enter weight of material 5 (in kg): 10

Total weight of materials: 155.00 kg

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.

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

// Function to update welding machine configuration
void updateMachineConfig(float *voltage, float *current, float minVoltage, float maxVoltage, float minCurrent, float maxCurrent) {
    // Validate voltage within specified range
    if (*voltage < minVoltage) {
        *voltage = minVoltage;
    } else if (*voltage > maxVoltage) {
```

```
        *voltage = maxVoltage;
    }

    // Validate current within specified range
    if (*current < minCurrent) {
        *current = minCurrent;
    } else if (*current > maxCurrent) {
        *current = maxCurrent;
    }
}

int main() {
    float voltage, current;
    float minVoltage = 10.0, maxVoltage = 50.0;  // Voltage range
    float minCurrent = 5.0, maxCurrent = 200.0;  // Current range

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

    // Update the machine configuration
    updateMachineConfig(&voltage, &current, minVoltage, maxVoltage, minCurrent, maxCurrent);

    // Output the updated configuration
    printf("Updated machine configuration: Voltage = %.2f V, Current = %.2f A\n", voltage, current);

    return 0;
}O/p: Enter the number of materials: 5
Enter weight for material 1: 12
```

Enter weight for material 2: 14

Enter weight for material 3: 20

Enter weight for material 4: 30

Enter weight for material 5: 60

Total weight of materials: 136.00

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.

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

void calculateDefectRate(int *total, int *defective, float *defectRate) {
   if (*total > *defective) {
      *defectRate = ((float)*defective / *total) * 100;
   } else {
      printf("Error: Total products must be greater than defective products.\n");
      *defectRate = -1.0; // Indicating an error
   }
}

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

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

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

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

    // Output the defect rate if no error
    if (defectRate >= 0) {
        printf("Defect Rate: %.2f%%\n", defectRate);
    }

    return 0;
}
```

O/p: Enter voltage (in volts): 100

Enter current (in amperes): 50

Updated machine configuration: Voltage = 50.00 V, Current = 50.00 A

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.

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

// Function to adjust the timing intervals ensuring they remain positive
void adjustTimingIntervals(float *timingIntervals, int size) {
    for (int i = 0; i < size; i++) {
        if (timingIntervals[i] < 0) {
            timingIntervals[i] = 0; // Adjust negative timing intervals to 0
            printf("Error: Negative timing interval at index %d. Setting to 0.\n", i);
        }
    }
```

```
}

int main() {
  int n;

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

  // Declare an array to store the timing intervals
  float timingIntervals[n];

  // Input the timing intervals
  for (int i = 0; i < n; i++) {
    printf("Enter timing interval for station %d: ", i + 1);
    scanf("%f", &timingIntervals[i]);

    // Validate input to ensure the timing interval is positive
    if (timingIntervals[i] < 0) {
      printf("Error: Timing interval cannot be negative. Setting to 0.\n");
      timingIntervals[i] = 0; // Set negative values to 0
    }
  }

  // Adjust the timing intervals
  adjustTimingIntervals(timingIntervals, n);

  // Output the adjusted timing intervals
  printf("Adjusted Timing Intervals:\n");
  for (int i = 0; i < n; i++) {
    printf("Station %d: %.2f seconds\n", i + 1, timingIntervals[i]);
```

```
    }

    return 0;
}
```

O/p: Enter the number of stations: 5

Enter timing interval for station 1: 12

Enter timing interval for station 2: 14

Enter timing interval for station 3: 32

Enter timing interval for station 4: 34

Enter timing interval for station 5: 54

Adjusted Timing Intervals:

Station 1: 12.00 seconds

Station 2: 14.00 seconds

Station 3: 32.00 seconds

Station 4: 34.00 seconds

Station 5: 54.00 seconds

11. CNC Machine Coordinates

Requirements:

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

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

// Function to update coordinates within machine limits
void updateCoordinates(float *x, float *y, float *z, float minX, float maxX, float minY, float maxY, float minZ, float maxZ) {
    // Ensure coordinates are within the specified limits
    if (*x < minX) *x = minX;
    if (*x > maxX) *x = maxX;
```

```
    if (*y < minY) *y = minY;
    if (*y > maxY) *y = maxY;
    if (*z < minZ) *z = minZ;
    if (*z > maxZ) *z = maxZ;
}

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

    // Define machine's coordinate limits
    float minX = 0.0, maxX = 100.0;
    float minY = 0.0, maxY = 100.0;
    float minZ = 0.0, maxZ = 50.0;

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

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

    // Output updated coordinates
    printf("Updated Coordinates: X = %.2f, Y = %.2f, Z = %.2f\n", x, y, z);

    return 0;
}
```

O/p: Enter current x coordinate: 12

Enter current y coordinate: -5

Enter current z coordinate: 3

Updated Coordinates: X = 12.00, Y = 0.00, Z = 3.00

12. Energy Consumption Tracker

Requirements:

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

Sol:

```
#include <stdio.h>

// Function to calculate total energy consumption, ensuring no negative values
void calculateTotalEnergy(float *energyUsage, int size, float *totalEnergy) {
  *totalEnergy = 0.0; // Initialize total energy to 0

  for (int i = 0; i < size; i++) {
    if (energyUsage[i] < 0) {
      printf("Error: Negative energy usage detected at index %d. Setting to 0.\n", i);
      energyUsage[i] = 0; // Set negative energy usage to 0
    }
    *totalEnergy += energyUsage[i]; // Add the energy usage to total energy
  }
}

int main() {
  int n;
  float totalEnergy;

  // Input the number of machines
  printf("Enter the number of machines: ");
```

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

    // Declare an array to store energy usage
    float energyUsage[n];

    // Input energy usage data for each machine
    for (int i = 0; i < n; i++) {
        printf("Enter energy usage for machine %d (in kWh): ", i + 1);
        scanf("%f", &energyUsage[i]);

        // Validate energy usage value
        if (energyUsage[i] < 0) {
            printf("Error: Energy usage cannot be negative. Setting to 0.\n");
            energyUsage[i] = 0; // Set negative values to 0
        }
    }

    // Calculate total energy consumption
    calculateTotalEnergy(energyUsage, n, &totalEnergy);

    // Output the total energy consumed
    printf("Total energy consumed by machines: %.2f kWh\n", totalEnergy);

    return 0;
}
```

O/p:

```
Enter the number of machines: 5
Enter energy usage for machine 1 (in kWh): 200
Enter energy usage for machine 2 (in kWh): 120
Enter energy usage for machine 3 (in kWh): 130
Enter energy usage for machine 4 (in kWh): 90
```

Enter energy usage for machine 5 (in kWh): 85

Total energy consumed by machines: 625.00 kWh

13. Production Rate Monitor

Requirements:

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

Sol:
```c
#include <stdio.h>

// Function to modify production rate
void updateProductionRate(int *rate, float adjustmentFactor, int minRate, int maxRate) {
    *rate += (int)(*rate * adjustmentFactor);

    // Ensure production rate is within permissible limits
    if (*rate < minRate) *rate = minRate;
    if (*rate > maxRate) *rate = maxRate;
}

int main() {
    int rate;
    float adjustmentFactor;
    int minRate = 50, maxRate = 500;

    printf("Enter current production rate: ");
    scanf("%d", &rate);
    printf("Enter adjustment factor (e.g., 0.1 for 10%): ");
    scanf("%f", &adjustmentFactor);

    updateProductionRate(&rate, adjustmentFactor, minRate, maxRate);
```

```
    printf("Updated production rate: %d\n", rate);
    return 0;
}
```

O/p:

Enter current production rate: 200

Enter adjustment factor (e.g., 0.1 for 10%): 0.2

Updated production rate: 240

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.

Sol:
```
#include <stdio.h>
#include <string.h>

// Function to update maintenance schedule
void updateMaintenanceSchedule(char *currentDate, char *nextDate) {
    if (strcmp(nextDate, currentDate) <= 0) {
        printf("Error: Next maintenance date must be later than the current date.\n");
    } else {
        printf("Maintenance schedule updated: Current: %s, Next: %s\n", currentDate, nextDate);
    }
}

int main() {
    char currentDate[20], nextDate[20];

    printf("Enter current maintenance date (YYYY-MM-DD): ");
```

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

    updateMaintenanceSchedule(currentDate, nextDate);

    return 0;
}
```

O/p:

Enter current maintenance date (YYYY-MM-DD): 202401-01-01

Enter next maintenance date (YYYY-MM-DD): 2025-01-01

Maintenance schedule updated: Current: 2024-01-01, Next: 2025-01-01

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.

Sol:

```
#include <stdio.h>

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

int main() {
    int score;

    printf("Enter quality score for the product: ");
    scanf("%d", &score);
```

```
    updateQualityMetrics(&score);

    printf("Updated quality score: %d\n", score);
    return 0;
}
```

O/p:

Enter quality score for the product: 200

Updated quality score: 100

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.

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

// Function to update space allocation
void updateSpaceAllocation(int *spaceUsed, int numSections, int maxCapacity) {
    int totalSpace = 0;

    for (int i = 0; i < numSections; i++) {
        totalSpace += spaceUsed[i];
    }

    if (totalSpace > maxCapacity) {
        printf("Error: Total space exceeds warehouse capacity.\n");
        return;
    }
```

```
    printf("Updated warehouse space allocation.\n");
}

int main() {
    int numSections, maxCapacity = 1000;

    printf("Enter number of sections: ");
    scanf("%d", &numSections);

    int spaceUsed[numSections];

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

    updateSpaceAllocation(spaceUsed, numSections, maxCapacity);

    return 0;
}
```

Enter number of sections: 4

Enter space used for section 1: 100

Enter space used for section 2: 200

Enter space used for section 3: 300

Enter space used for section 4: 4- 00

Updated warehouse space allocation.

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.

O/p:
```
#include <stdio.h>

// Function to update packaging machine settings
void updatePackagingSettings(float *speed, float *tension, float minSpeed, float maxSpeed, float minTension, float maxTension) {
   if (*speed < minSpeed) *speed = minSpeed;
   if (*speed > maxSpeed) *speed = maxSpeed;
   if (*tension < minTension) *tension = minTension;
   if (*tension > maxTension) *tension = maxTension;
}

int main() {
   float speed, tension;
   float minSpeed = 1.0, maxSpeed = 100.0, minTension = 5.0, maxTension = 50.0;

   printf("Enter speed (m/s): ");
   scanf("%f", &speed);
   printf("Enter wrap tension (kg): ");
   scanf("%f", &tension);

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

   printf("Updated settings - Speed: %.2f m/s, Tension: %.2f kg\n", speed, tension);

   return 0;
}
```

O/p:
```
Enter speed (m/s): 50
Enter wrap tension (kg): 75
Updated settings - Speed: 50.00 m/s, Tension: 50.00 kg
```

18. Process Temperature Control

Requirements:

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

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

// Function to adjust temperature
void adjustTemperature(float *temp, float minTemp, float maxTemp) {
   if (*temp < minTemp) *temp = minTemp;
   if (*temp > maxTemp) *temp = maxTemp;
}

int main() {
   float temperature;
   float minTemp = 10.0, maxTemp = 50.0;

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

   adjustTemperature(&temperature, minTemp, maxTemp);

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

   return 0;
}
```

o/p:

```
Enter current temperature: 60
Adjusted temperature: 50.00
```

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.

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

// Function to update scrap count
void updateScrapCount(int *scrapCount, int size) {
   for (int i = 0; i < size; i++) {
      if (scrapCount[i] < 0) {
         scrapCount[i] = 0;  // Prevent negative scrap count
      }
   }
}

int main() {
   int n;

   printf("Enter number of material types: ");
   scanf("%d", &n);

   int scrapCount[n];

   for (int i = 0; i < n; i++) {
      printf("Enter scrap count for material %d: ", i + 1);
      scanf("%d", &scrapCount[i]);
   }
```

```
    updateScrapCount(scrapCount, n);

    printf("Updated scrap count:\n");
    for (int i = 0; i < n; i++) {
        printf("Material %d: %d\n", i + 1, scrapCount[i]);
    }

    return 0;
}
```

O/p:

```
Enter number of material types: 4
Enter scrap count for material 1: 123
Enter scrap count for material 2: 143
Enter scrap count for material 3: 80
Enter scrap count for material 4: 98
Updated scrap count:
Material 1: 123
Material 2: 143
Material 3: 80
Material 4: 98
```

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.

Sol:

```
#include <stdio.h>

// Function to calculate overall performance
```

```c
void calculatePerformance(int *data, int size, float *performance) {
    int total = 0;

    for (int i = 0; i < size; i++) {
        if (data[i] < 0) {
            printf("Error: Invalid production data at index %d\n", i);
            return;
        }
        total += data[i];
    }

    *performance = (float)total / size;
}

int main() {
    int n;

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

    int data[n];
    float performance;

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

    calculatePerformance(data, n, &performance);

    printf("Overall shift performance: %.2f\n", performance);
```

```
   return 0;
}
```

O/p: Enter number of shifts: 5

Enter production data for shift 1: 100

Enter production data for shift 2: 300

Enter production data for shift 3: 400

Enter production data for shift 4: 56

Enter production data for shift 5: 78

Overall shift performance: 186.80