CS 137

Programming Principles

Chapter 6

Arrays and Introduction to Pointers

Victoria Sakhnini

Table of Contents

Arrays

C language provides arrays with capabilities to define a set of ordered data items of the same type. All arrays consist of contiguous memory locations. Instead of declaring individual variables such as number0, number1, ..., and number99, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, ..., `numbers[99]` to represent individual variables. An index accesses a specific element in an array. **Arrays start at index 0.**

To declare an array, specify the type of the data items, the name, and the number of elements (array length):

```
type array_name[length];
```

For example, to define an array `a` of length 5 and initialize it with five integer values:

```
int a[5] = {10, -7, 3, 8, 42};
```

Variable Name	Memory Address	Value
a	10856	10
	10860	-7
	10864	3
	10868	8
	10872	42

📌 Note 1: Values are stored in memory contiguously. The compiler decides the start location; the lowest address corresponds to the first element and the highest address to the last element. This fact is helpful when using pointers with arrays.

Note 2: Each element of type int requires 4 bytes in memory.

We access the values via `a[0]`, `a[1]`, etc.

Declaration and Initialization Variants

```
int a[5] = {10, -7, 3, 8, 42};        // explicit initialization
int a[]  = {10, -7, 3, 8, 42};        // size inferred from initializer
int a[5] = {1, 2, 3};                 // last two entries are 0 by default
int a[5] = {0};                        // all-zero array
int a[5];                              // uninitialized — contains garbage
int a[5] = {[2] = 2, [4] = 3123};     // specific entries set; rest are 0
```

⚠ The following are NOT valid syntax:
int a[5]; a = {1,2,3,4,5};
int a[5]; a[5] = {1,2,3,4,5};

Initialization is done at compile-time and must be done all at once. Otherwise you must assign entries one by one, e.g. a[3] = 10;

Example 1 — Summing an Array

```c
#include <stdio.h>
int main(void)
{
    int a[] = { 10, -7, 3, 8, 42 };
    int sum = 0; // Must assign 0 — do not assume a declared variable
                 // is initialized to zero by default
    for (int i = 0; i < 5; i++)
    {
        sum += a[i];
    }
    printf("%d\n", sum);
    return 0;
}
```

Example 2 — Generating Fibonacci Numbers

```c
#include <stdio.h>
int main(void)
{
    int fib[10];
    int ind;
    fib[0] = 1;
    fib[1] = 1;
    for (ind = 2; ind < 10; ind++)
    {
        fib[ind] = fib[ind - 1] + fib[ind - 2];
    }
    // Expected output: 1 1 2 3 5 8 13 21 34 55
    for (ind = 0; ind < 10; ind++)
    {
        printf("%d ", fib[ind]);
    }
    printf("\n");
    return 0;
}
```

```
ind=2, fib[2] = fib[2-1] + fib[2-2]  => fib[2]=2
ind=3, fib[3] = fib[3-1] + fib[3-2]  => fib[3]=3
ind=4, fib[4] = fib[4-1] + fib[4-2]  => fib[4]=5
ind=5, fib[5] = fib[5-1] + fib[5-2]  => fib[5]=8
ind=6, fib[6] = fib[6-1] + fib[6-2]  => fib[6]=13
ind=7, fib[7] = fib[7-1] + fib[7-2]  => fib[7]=21
ind=8, fib[8] = fib[8-1] + fib[8-2]  => fib[8]=34
ind=9, fib[9] = fib[9-1] + fib[9-2]  => fib[9]=55
```

Sieve of Eratosthenes

Task: Find all prime numbers up to n.

Idea

- List all numbers from 2 to n.
- Start at 2 and cross out all multiples of 2.
- Choose the next smallest number not crossed out and repeat.
- Stop when the next number is greater than √n.
- The remaining numbers are prime.

Example 1 — n = 20

```
iteration 1: 2 3 [4] 5 [6] 7 [8]  9  [10] 11 [12] 13 [14] 15  [16] 17 [18] 19
[20]
iteration 2: 2 3 [4] 5 [6] 7 [8] [9] [10] 11 [12] 13 [14] [15] [16] 17 [18] 19
[20]
stop: next smallest (5) is greater than sqrt(20)
Primes: 2, 3, 5, 7, 11, 13, 17, 19
```

Example 2 — n = 35

```
iteration 1: cross multiples of 2
iteration 2: cross multiples of 3
iteration 3: cross multiples of 5
stop: next smallest (7) is greater than sqrt(35)
Primes: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31
```

Implementation

We use an array of size n where each index represents a number, and the value is 0 (crossed out) or 1 (not crossed out).

```
#include <stdio.h>

void sieve(int a[], int n);

int main(void)
{
    int n = 111;
    int a[n + 1];
    sieve(a, n + 1);
    printf("The prime numbers up to %d are: \n", n);
    for (int i = 0; i <= n; i++)
    {
        if (a[i])
            printf("%d\n", i);
    }
    return 0;
}
```

```
void sieve(int a[], int m)
{
    a[0] = 0;
    if (m == 1)
        return;
    a[1] = 0;
    if (m == 2)
        return;
    // Set potential primes
    for (int i = 2; i < m; i++)
        a[i] = 1;
    for (int i = 2; i * i <= m - 1; i++)
    {
        if (a[i])
        {
            // Strike out multiples
            for (int j = 2 * i; j < m; j += i)
                a[j] = 0;
        }
    }
}
```

📌 Notice that we passed an array and modified the original array. What is passed to the function is not the array itself but rather where the array exists in memory. Thus, changes inside the array change the values at the original memory location. More on this later.

A complete trace for `n=25`

```
 0  0  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
 i=2
 0  0  1  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1
 i=3
 0  0  1  1  0  1  0  1  0  0  0  1  0  1  0  0  0  1  0  1  0  0  0  1  0  1
 i=4
 0  0  1  1  0  1  0  1  0  0  0  1  0  1  0  0  0  1  0  1  0  0  0  1  0  1
 i=5
 0  0  1  1  0  1  0  1  0  0  0  1  0  1  0  0  0  1  0  1  0  0  0  1  0  0
The primes numbers up to 25 are:
 2
 3
 5
 7
 11
 13
 17
 19
 23
```

sizeof Operator

The `sizeof` operator computes the size of its argument in bytes (the number of bytes allocated in memory to the argument's data type). The result is of unsigned integral type `size_t`.

```
#include <stdio.h>

int main(void)
{
    int a[] = { -1, 2, -1, 2, -4, 1 };

    // Since all values in an array are the same type,
    // the following prints the length of array a (6).
    printf("%d\n", sizeof(a) / sizeof(a[0]));

    // output: 1 4 4 24
    printf("%d %d %d %d\n", sizeof(char), sizeof(int), sizeof(a[0]),
sizeof(a));

    // %zu is preferred for cross-platform compatibility
    // (works with both 32-bit and 64-bit systems for type size_t)
    // output: 1 4 4 24
    printf("%zu %zu %zu %zu\n", sizeof(char), sizeof(int), sizeof(a[0]),
sizeof(a));

    return 0;
}
```

📌 sizeof() may give different output depending on the machine. The examples above were run on a 64-bit gcc compiler. The size of array a is 24 because it includes 6 integers; each integer is 4 bytes, so 6 × 4 = 24.

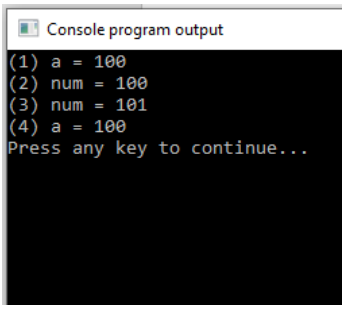
Passing Arrays to Functions

Consider the following example comparing how a regular integer vs. an array is passed to a function.

Passing a scalar (by value)

```
#include <stdio.h>

void plusOne(int num)
{
    printf("(2) num = %d\n", num);
    num++;
    printf("(3) num = %d\n", num);
}
```

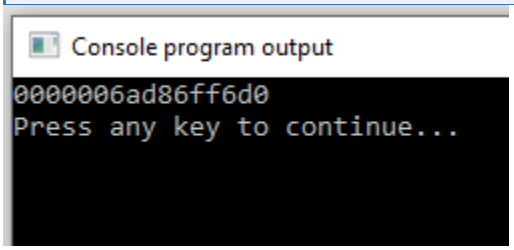
```
int main(void)
{
    int a = 100;
    printf("(1) a = %d\n", a);
    plusOne(a);
    printf("(4) a = %d\n", a);
    return 0;
}
```



The output shows that a copy of the value of `a` was passed to `num`; thus, any change to `num` did not affect the value of `a`.

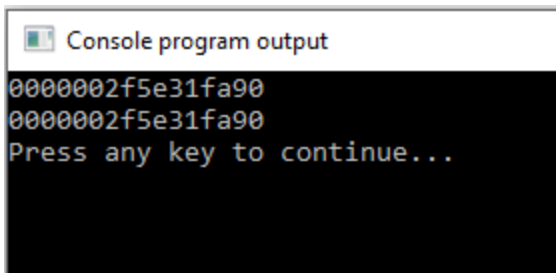
Array name is a pointer

```
#include <stdio.h>

int main(void)
{
    int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    printf("%p\n", a);   // %p prints a memory address
    return 0;
}
```

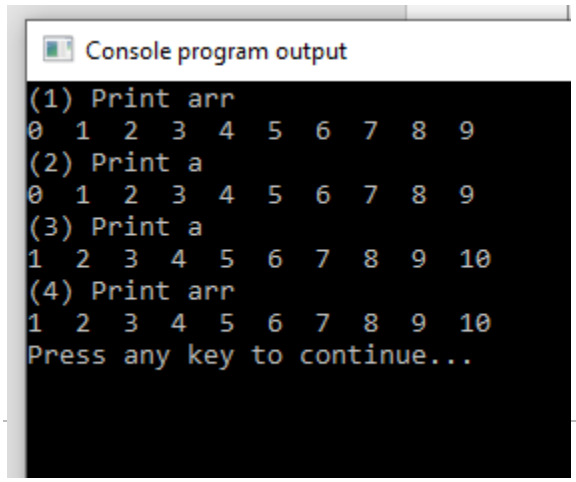


The output shows that an array name is a pointer to something in memory — specifically, it points to the first element. The following program confirms that `a` and `&a[0]` are the same address:

```
#include <stdio.h>

int main(void)
{
    int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    printf("%p\n", a);
    printf("%p\n", &a[0]);
    return 0;
}
```



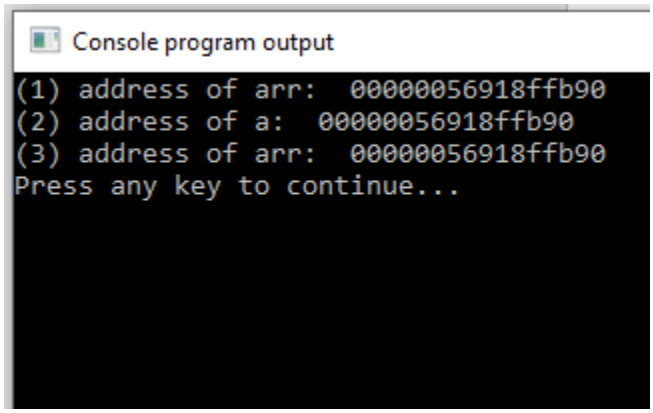

```c
#include <stdio.h>

void plusOne(int a[], int n)
{
    printf("(2) Print a\n");
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");

    for (int i = 0; i < n; i++)
        a[i]++;

    printf("(3) Print a\n");
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

int main(void)
{
    int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    printf("(1) Print arr\n");
    for (int i = 0; i < 10; i++)
        printf("%d ", arr[i]);
    printf("\n");

    plusOne(arr, 10);

    printf("(4) Print arr\n");
    for (int i = 0; i < 10; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```



When `plusOne` is called, a pointer to the values of `arr` is passed to `a`. Both `arr` and `a` point to the same data, so updating `a` updates `arr`.

Confirming the shared address

```
#include <stdio.h>

void plusOne(int a[], int n)
{
    printf("(2) address of a: %p\n", a);
    for (int i = 0; i < n; i++)
        a[i]++;
}

int main(void)
{
    int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    printf("(1) address of arr: %p\n", arr);
    plusOne(arr, 10);
    printf("(3) address of arr: %p\n", arr);
    return 0;
}
```



In C, these two function signatures are equivalent:
void plusOne(int a[], int n); void plusOne(int *a, int n); They are equivalent because the array name is a pointer to the first element of the array.

Rewritten using int *a

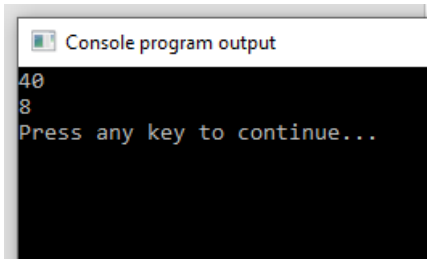
```
#include <stdio.h>

void plusOne(int *a, int n)
{
    printf("(2) address of a: %p\n", a);
    for (int i = 0; i < n; i++)
        a[i]++;
}

int main(void)
{
    int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    printf("(1) address of arr: %p\n", arr);
    plusOne(arr, 10);
    printf("(3) address of arr: %p\n", arr);
    return 0;
}
```

```
#include <stdio.h>

void plusOne(int *a, int n)
{
    printf("(2) Print a\n");
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");

    for (int i = 0; i < n; i++)
        a[i]++;

    printf("(3) Print a\n");
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

int main(void)
{
    int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    printf("(1) Print arr\n");
    for (int i = 0; i < 10; i++)
        printf("%d ", arr[i]);
    printf("\n");

    plusOne(arr, 10);

    printf("(4) Print arr\n");
    for (int i = 0; i < 10; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

Caution! Pointer Decay

What does the following program print?

```
#include <stdio.h>

void sizeofArray(int a[])
{
    printf("%zu\n", sizeof(a));  // output: 8
}

int main(void)
{
    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    printf("%zu\n", sizeof(a));  // output: 40
    sizeofArray(a);
    return 0;
}
```



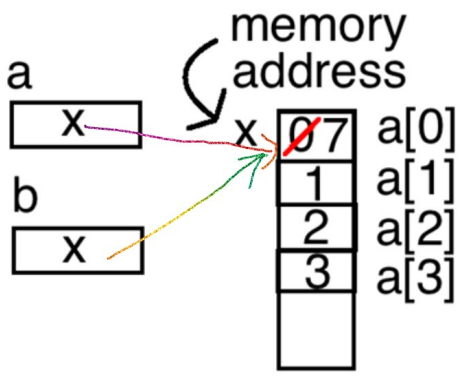
Probably not what you expected. The `a` inside the function is really just a memory address, so `sizeof(a)` gives the size of a pointer (8 bytes on a 64-bit system). The `sizeof(a)` inside `main` gives the size of the entire array (10 × sizeof(int) = 40 bytes). This awkwardness is why we must pass the size of the array to functions. This is called **pointer decay**.

More About Pointers — References

What does the following program print?

```
#include <stdio.h>

int main(void)
{
    int a[] = { 0, 1, 2, 3 };
    int *b = a;
    printf("%d\n", a[0]);
    b[0] = 7;
    printf("%d\n", a[0]);
    return 0;
}
```

Because `b` is pointing to the exact same location as `a` (the array of 4 integers), changing the value that `b` points to also changes the value in array `a`.



Copying Arrays

To make an exact copy of an array that is distinct (no aliasing — changing one doesn't change the other), create a new array of the same size and manually copy every element.

```
#include <stdio.h>
#define LEN 3

int main(void)
{
    int a[LEN] = { 2, 4, 6 };
    int b[LEN];
    for (int i = 0; i < LEN; i++)
        b[i] = a[i];
    a[0] = 19;
    printf("a[0]==%d\n", a[0]);
    printf("b[0]==%d\n", b[0]);  // b is a copy of a — no reference
    return 0;
}
```

Variable-Length Array

A **variable-length array (VLA)**, also called variable-sized or runtime-sized, is an array whose length is determined at run time instead of compile time. The C11 standard made this feature optional.

⚠ Variable-length arrays will NOT be used in this course. Any program you submit using variable-length arrays will receive 0, unless the array is allocated on the heap (covered later).

Example of a VLA (for reference only):

```
#include <stdio.h>

int main(void)
{
    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int arr[n];  // VLA — NOT permitted in this course
    for (int i = 0; i < n; i++)
        arr[i] = i * 3 + 5;
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

Multi-Dimensional Arrays

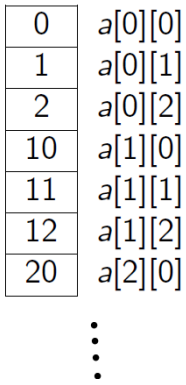
What we have learned so far covers linear (one-dimensional) arrays. C allows arrays of any dimension. In this section we focus on two-dimensional arrays, whose most natural application is the matrix.

Example — 4×3 Matrix Sum

```
#include <stdio.h>

int main(void)
{
    int matrix[4][3] = { {0, 1, 2},
                         {10, 11, 12},
                         {20, 21, 22},
                         {30, 31, 32} };
    int sum = 0;
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            sum += matrix[i][j];
        }
    }
    printf("Sum = %d\n", sum);
    return 0;
}
```

```
   sum    i     j   matrix[i][j]
    0
    0     0     0     0
    1     0     1     1
    3     0     2     2
   13     1     0    10
   24     1     1    11
   36     1     2    12
   56     2     0    20
   77     2     1    21
   99     2     2    22
  129     3     0    30
  160     3     1    31
  192     3     2    32
Sum = 192
```

📌 In memory, a two-dimensional array is stored in row-major order: all elements of row 0 come first, then all elements of row 1, and so on. This means the same traversal can also be written as matrix[0][i*3+j], though the nested-loop form above is far more readable.



Equivalent but less readable version using row-major indexing:

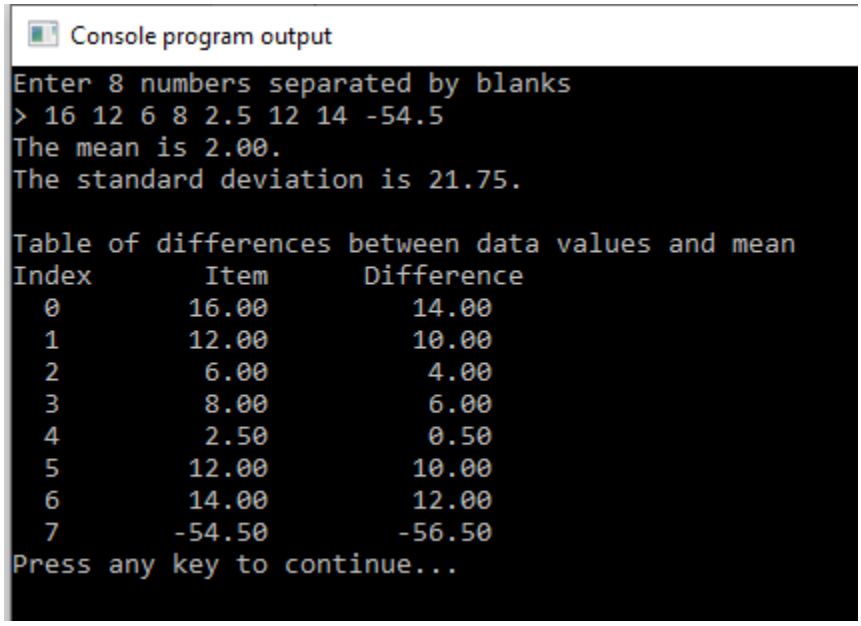
```c
#include <stdio.h>

int main(void)
{
    int matrix[4][3] = { {0, 1, 2},
                         {10, 11, 12},
                         {20, 21, 22},
                         {30, 31, 32} };
    int sum = 0;
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            sum += matrix[0][i * 3 + j];
        }
    }
    printf("Sum = %d\n", sum);
    return 0;
}
```

When initializing, all dimensions except possibly the first must be explicit. This applies to function definitions as well. Example: int a[][2] = {{1,2},{3,4}};

Additional Examples

Example 1 — Mean and Standard Deviation

```
#include <stdio.h>
#include <math.h>
#define MAX_ITEM 8

int main(void)
{
    double x[MAX_ITEM], mean, st_dev, sum, sum_sqr;
    int i;

    printf("Enter %d numbers separated by blanks\n> ", MAX_ITEM);
    for (i = 0; i < MAX_ITEM; ++i)
        scanf("%lf", &x[i]);

    sum = 0;
    sum_sqr = 0;
    for (i = 0; i < MAX_ITEM; ++i)
    {
        sum += x[i];
        sum_sqr += x[i] * x[i];
    }

    mean = sum / MAX_ITEM;
    st_dev = sqrt(sum_sqr / MAX_ITEM - mean * mean);
    printf("The mean is %.2f.\n", mean);
    printf("The standard deviation is %.2f.\n", st_dev);

    printf("\nTable of differences between data values and mean\n");
    printf("Index  Item  Difference\n");
    for (i = 0; i < MAX_ITEM; ++i)
        printf("%3d%4c%9.2f%5c%9.2f\n", i, ' ', x[i], ' ', x[i] - mean);

    return (0);
}
```



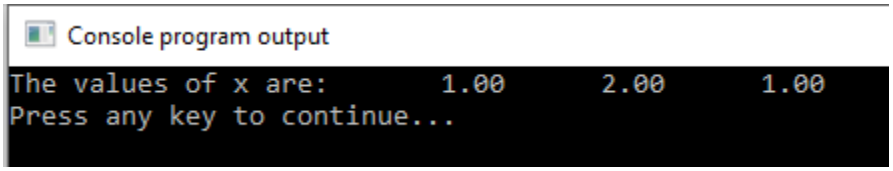
Example 2 — Gaussian Elimination

The following program solves a system of N linear equations using Gaussian elimination and back substitution.

```
#define FALSE 0
#define TRUE  1
#define N     3

#include <stdio.h>
#include <math.h>

/* Performs pivoting with respect to the pth row and column */
void pivot(double aug[N][N+1], int p, int *piv_foundp)
{
    double xmax, xtemp;
    int j, k, max_row;

    xmax    = fabs(aug[p][p]);
    max_row = p;
    for (j = p + 1; j < N; ++j)
    {
        if (fabs(aug[j][p]) > xmax)
        {
            xmax    = fabs(aug[j][p]);
            max_row = j;
        }
    }

    if (xmax == 0)
    {
        *piv_foundp = FALSE;
    }
    else
    {
```

```
        *piv_foundp = TRUE;
        if (max_row != p)
        {
            for (k = p; k < N + 1; ++k)
            {
                xtemp        = aug[p][k];
                aug[p][k]    = aug[max_row][k];
                aug[max_row][k] = xtemp;
            }
        }
    }
}

/* Triangularizes the augmented matrix */
void gauss(double aug[N][N+1], int *sol_existsp)
{
    int j, k, p;
    double piv_recip, xmult;

    *sol_existsp = TRUE;
    for (p = 0; *sol_existsp && p < (N - 1); ++p)
    {
        pivot(aug, p, sol_existsp);
        if (*sol_existsp)
        {
            piv_recip = 1.0 / aug[p][p];
            aug[p][p] = 1.0;
            for (k = p + 1; k < N + 1; ++k)
                aug[p][k] *= piv_recip;
            for (j = p + 1; j < N; ++j)
            {
                xmult    = -aug[j][p];
                aug[j][p] = 0;
                for (k = p + 1; k < N + 1; ++k)
                    aug[j][k] += xmult * aug[p][k];
            }
        }
    }

    if (aug[N-1][N-1] == 0)
    {
        *sol_existsp = FALSE;
    }
    else if (*sol_existsp)
    {
        piv_recip      = 1.0 / aug[N-1][N-1];
        aug[N-1][N-1]  = 1.0;
        aug[N-1][N]   *= piv_recip;
    }
}

/* Back substitution */
void back_sub(double aug[N][N+1], double x[N])
{
    double sum;
    int i, j;
    x[N-1] = aug[N-1][N];
    for (i = N - 2; i >= 0; --i)
    {
        sum = 0;
        for (j = i + 1; j < N; ++j)
```

```
            sum += aug[i][j] * x[j];
        x[i] = aug[i][N] - sum;
    }
}

int main(void)
{
    double aug[N][N+1] = { {1.0,  1.0,  1.0, 4.0},
                           {2.0,  3.0,  1.0, 9.0},
                           {1.0, -1.0, -1.0, -2.0} },
           x[N];
    int sol_exists;

    gauss(aug, &sol_exists);

    if (sol_exists)
    {
        back_sub(aug, x);
        printf("The values of x are: %10.2f%10.2f%10.2f\n", x[0], x[1], x[2]);
    }
    else
    {
        printf("No unique solution\n");
    }
    return (0);
}
```



Extra Practice Problems

Try to solve and test the problems before looking at the provided solutions. You learn the most by doing.

1.

Write a program that reads integers in the range 0–100 and prints how many times each integer was read, sorted by value.

Example input: 12 90 67 90 100 12 8 0 3
Example output:
```
0 1
3 1
8 1
12 2
67 1
90 2
100 1
```

2.

Write a program that reads an integer and prints how many times each digit (0–9) appears in it.

Example input: 56065890
Example output:
```
0 2  1 0  2 0  3 0  4 0
5 2  6 2  7 0  8 1  9 1
```

3.

Consider the following function that returns the max value of an array of integers.

```
#include <stdio.h>

int max_array(int a[], int n)
{
    int max = 0;
    for (int i = 0; i < n; i++)
        if (a[i] > max)
            max = a[i];
    return max;
}
```

a) What is wrong with this solution? Which case will not provide the correct results?
b) Fix the function.

a) When all integers are negative, the function returns 0, which is incorrect.

4.

Implement the function int index_max_array(int a[], int n), which returns the index of the maximal value. If the maximal value appears more than once, return the index of its first occurrence.

5.

Write a C program that checks if a given 9×9 Sudoku board is valid.
Requirements:

- Input a 9×9 grid where each cell contains 1–9 or 0 (empty).
- Check that each row contains digits 1–9 without repetition.
- Check that each column contains digits 1–9 without repetition.
- Check that each 3×3 sub-grid contains digits 1–9 without repetition.
- Output whether the board is valid or not.

Sample Input 1 (valid):
5 3 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 2 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 8 0 0 7 9
Output: "The Sudoku board is valid."

Sample Input 2 (invalid — last row has duplicate 8):
...same as above except last row: 0 0 0 0 8 0 0 8 9
Output: "The Sudoku board is invalid."

6.

Write a C program to implement a two-player Connect Four game on a 6×7 board.
Players take turns dropping pieces ('X' and 'O') into columns.
Implement win detection for horizontal connections of four only.
Complete the functions isValidMove, makeMove, and checkWin in the skeleton below.

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

#define ROWS    6
#define COLUMNS 7

char board[ROWS][COLUMNS];
char currentPlayer = 'X';

void initializeBoard() {
    for (int i = 0; i < ROWS; i++)
        for (int j = 0; j < COLUMNS; j++)
            board[i][j] = ' ';
}
```

```
void displayBoard() {
    printf("Connect Four Game\n");
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLUMNS; j++)
            printf("| %c ", board[i][j]);
        printf("|\n");
    }
    for (int j = 0; j < COLUMNS; j++)
        printf(" %d ", j);
    printf("\n");
}

bool isValidMove(int column) {
    // your code here
}

void makeMove(int column) {
    // your code here
}

bool checkWin() {
    // your code here
}

int main() {
    initializeBoard();
    displayBoard();
    int column;
    bool validMove, gameOver = false;
    while (!gameOver) {
        printf("Player %c, enter your move (column 0-6): ", currentPlayer);
        scanf("%d", &column);
        validMove = isValidMove(column);
        if (validMove) {
            makeMove(column);
            displayBoard();
            gameOver = checkWin();
            currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';
        } else {
            printf("Invalid move. Try again.\n");
        }
    }
    if (checkWin())
        printf("Player %c wins!\n", currentPlayer);
    else
        printf("It's a draw!\n");
    return 0;
}
```

7.

For all functions in this question:
- You cannot mutate any of the arrays.
- You must NOT define your own local arrays.
- Do not assume anything about the length (other than it is positive).
- Requires: all array parameters are valid (not NULL), all length parameters > 0.

a) Complete bool unique(int a[], int len);
Returns true if all elements of a are distinct (no duplicates).
Examples: {1, 3, 7} => true | {1, 3, 1} => false

b) Complete int longest_sorted(int a[], int len);
Returns the length of the longest consecutive subarray that is either non-descending or non-increasing.
Examples: {5, 1, 3, 7, 2} => 3 (1 <= 3 <= 7) | {3, 2, 2, 1} => 4 (3 >= 2 >= 2 >= 1)

8.

Complete int singleNumber(int *nums, int numsSize);
Given a non-empty array where every element appears twice except for one, find and return the single element.

Example 1: nums = [2, 2, 1] → 1
Example 2: nums = [4, 1, 2, 1, 2] → 4
Example 3: nums = [1] → 1

9.

Analyze survey data using 2D arrays (max 100 respondents, 100 items).
Implement:
- void inputSurveyData(int surveyData[][MAX_ITEMS], int respondents, int items)
- void surveyAnalyzer(int surveyData[][MAX_ITEMS], int respondents, int items)

surveyAnalyzer should display average rating per item, and the items with the highest and lowest ratings.

Example Input1 (the bold is the user input):

Enter the number of respondents and items: **2 2**

Enter survey data:

Respondent 1:

1 1

Respondent 2:

10 4

Example Output1:

Average Ratings:

Item 1: 5.5

Item 2: 2.5

Highest Rating: 10 (Item 1)

Lowest Rating: 1 (Item 1)

Example Input2:

Enter the number of respondents and items: **3 4**

Enter survey data:

Respondent 1:

1 2 3 4

Respondent 2:

5 2 7 3

Respondent 3:

1 9 8 9

Example output 2:

Average Ratings:

Item 1: 2.3

Item 2: 4.3

Item 3: 6.0

Item 4: 5.3

Highest Rating: 9 (Item 2)

Lowest Rating: 1 (Item 1)

```
#include <stdio.h>
#define MAX_RESPONDENTS 100
#define MAX_ITEMS       100

void inputSurveyData(int surveyData[][MAX_ITEMS], int respondents, int items)
{ }

void surveyAnalyzer(int surveyData[][MAX_ITEMS], int respondents, int items) {
}

int main(void) {
    int respondents, items;
    printf("Enter the number of respondents and items: ");
    scanf("%d %d", &respondents, &items);
    int surveyData[MAX_RESPONDENTS][MAX_ITEMS];
    inputSurveyData(surveyData, respondents, items);
    surveyAnalyzer(surveyData, respondents, items);
    return 0;
}
```

10.

Write bool smiley(int n, int m, int data[n][m]) that returns true if a specific smiley-face pattern of 1s exists in the given 2D array (n >= 5 rows, m >= 5 columns), and false otherwise.
The exact smiley pattern is as follows:

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

11.

There are `N` athletes, each coming one by one to drink water. The `i`-th athlete (`1 ≤ i ≤ N`) needs `drinks[i]` liters of water to fully quench their thirst.
Your task is to implement the function `int servingAthelete(int N, int M, int drinks[SIZE]);` which returns the number of athletes who can fully quench their thirst.
If an athlete cannot fully quench their thirst because insufficient water is left in the tank, they will drink whatever is left before moving on. However, only those athletes who can fully quench their thirst count towards the total.
You are also given `#define SIZE (100 + 1)` as the maximum size of the array `drinks[]`.
Function Signature:

```
int servingAthelete(int N, int M, int drinks[SIZE]);
```

Parameters:

- `N`: an integer representing the number of athletes.
- `M`: an integer representing the total amount of water (in liters) the tank can hold.
- `drinks[SIZE]`: an array of integers, where `drinks[i]` represents the amount of water (in liters) that the `i-th` athlete needs.

Return:

- The function should return an integer representing the number of athletes who can fully quench their thirst.

Constraints:

- `1 ≤ N ≤ 100`
- `1 ≤ M ≤ 1000`
- `1 ≤ drinks[i] ≤ 100`

Example:

Sample Input 1:

5 20
4 3 5 7 6

Sample Output 1:

4

Explanation: - The first athlete drinks 4 liters of water, leaving 16 liters in the tank. - The second athlete drinks 3 liters of water, leaving 13 liters in the tank. - The third athlete drinks 5 liters of water, leaving 8

liters in the tank. - The fourth athlete drinks 7 liters of water, leaving 1 liter in the tank. - The fifth athlete needs 6 liters of water, but only 1 liter is left, so they cannot fully quench their thirst.

Thus, the number of athletes can fully quench their thirst is 4.

Sample Input 2:

5 10
2 3 2 3 5

Sample Output 2:

4

Sample Input 3:

1 5
1

Sample Output 3:

1

Explanation:

- In **Sample Input 1**, 4 athletes can fully quench their thirst before the water runs out.
- In **Sample Input 2**, the first 4 athletes can fully quench their thirst, but the fifth one cannot.
- In **Sample Input 3**, the single athlete can fully quench their thirst as the tank has enough water.