CS 137

Arrays and Introduction to Pointers

Fall 2025

Victoria Sakhnini

Table of Contents

Arrays 2

Sieve of Eratosthenes 5

`sizeof` Operator 8

Passing Arrays to Functions 9

Caution! Pointer decay 14

More about pointers - References 15

Copying Arrays 16

Variable-length array 16

Multi-Dimensional Arrays 17

Additional Examples 19

Extra Practice Problems 23

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]`, and ..., `numbers[99]` to represent individual variables. An index accesses a specific element in an array. Arrays start index at `0`.

To declare an array, you need to specify the type of the data items, the name and the number of elements (array's length).

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

For example: to define an array (`a`) of length `5` and initialize it with five integer values, you do the following:

```
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: The values are stored in memory contiguously. However, the compiler decides the start location and the available memory (The addresses above are examples). Also, note that the lowest address corresponds to the first element and the highest address to the last element. This fact is helpful when we learn about using pointers with arrays.

Note 2: each element in the array of type integer requires 4 bytes in memory

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

More cases to consider:

- `int a[5] = {10,-7,3,8,42};`
- `int a[] = {10,-7,3,8,42};` (size is inferred)
- `int a[5] = {1,2,3};` (last two entries are 0 by default)
- `int a[5] = {0};` (creates an all 0 array)
- `int a[5];` (uninitialized array contains garbage entries)
- `int a[5] = {[2] = 2, [4] = 3123};` (specified the third and fifth entries - rest are 0)

Warning:

`int a[5]; a = {1,2,3,4,5};` is not a valid syntax

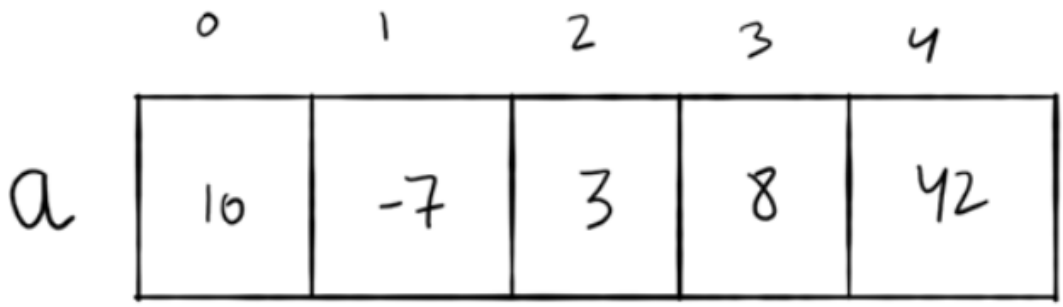
Nor this: `int a[5]; a[5] = {1,2,3,4,5};`

Note: initialization is done at compile-time and must be done at once (otherwise, you need to manually enter entries, one by one, such as `a[3]=10;` assign `10` as the fourth entry in array `a`)

Example1: Summing an Array:

```
1. #include <stdio.h>
2. int main(void)
3. {
4.          int a[] = { 10, -7, 3, 8, 42 };
5.          int sum = 0;    // You must assign 0, don't assume that the declared
   variable
                              will be initialized to zero by default
6.          for (int i = 0; i < 5; i++)
7.          {
8.                  sum += a[i];
9.          }
10.         printf("%d\n", sum);
11.         return 0;
12.  }
```

Tracing:



sum	i	a[i]
0		
10	0	10
3	1	-7
6	2	3
14	3	8
56	4	42

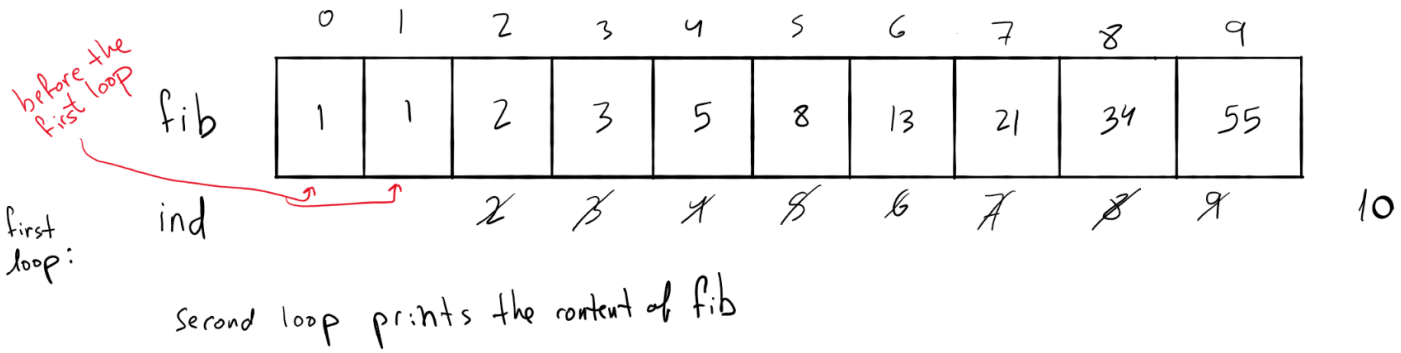
output

56

Example2: 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 12 21 34 55
        for (ind = 0; ind < 10; ind++)
        {
                printf("%d  ", fib[ind]);
        }
        printf("\n");
        return 0;
}
```

Tracing




```
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 the numbers from 2 to n.
- Start at 2 and cross out all the multiples of 2.
- Choose the following smallest number not stricken out and repeat the crossing.
- Stop when the following number is greater than $\sqrt{n}$
- The remaining numbers are prime.

Example1 (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 (because the next smallest number (5) is greater than $\sqrt{20}$

The prime numbers are: 2,3,5,7,11,13,17,19

Example2 (n==35),

iteration 1: 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ 27 ~~28~~ 29 ~~30~~ 31 ~~32~~ 33 ~~34~~ 35

iteration 2: 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ 25 ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~ 31 ~~32~~ ~~33~~ ~~34~~ 35

iteration 3: 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~ ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~ 31 ~~32~~ ~~33~~ ~~34~~ ~~35~~

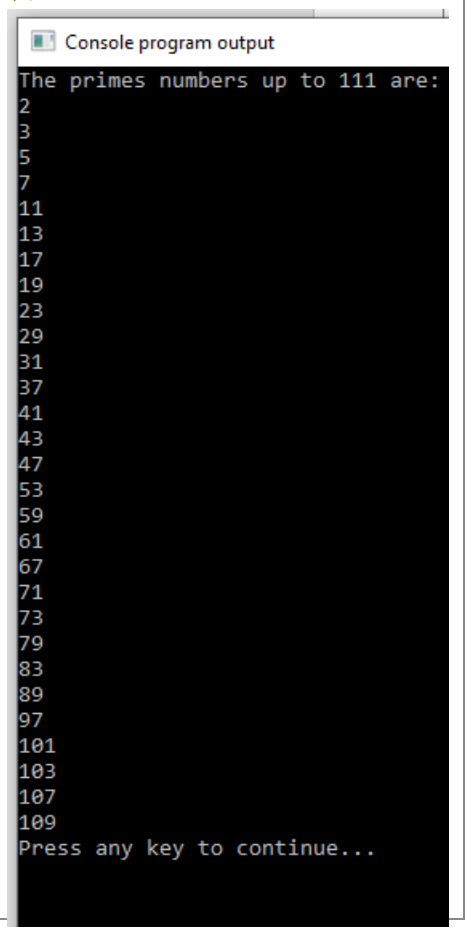
stop (because the next smallest number (7) is greater than $\sqrt{35}$

The prime numbers are: 2,3,5,7,11,13,17,19,23,29,31

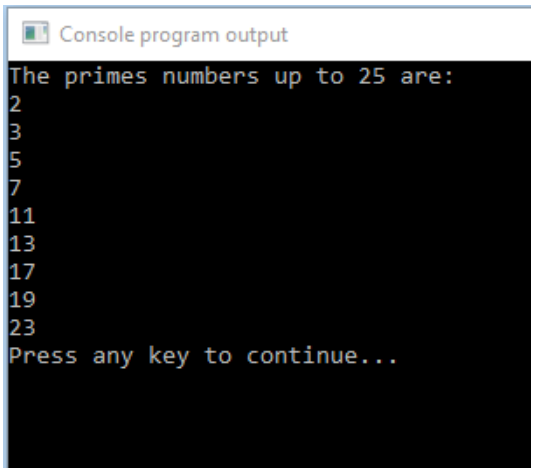
Implementation:

We use an array of size n where the indexes represent the numbers up to n, and the values are 0(stricken) or 1 (not stricken)

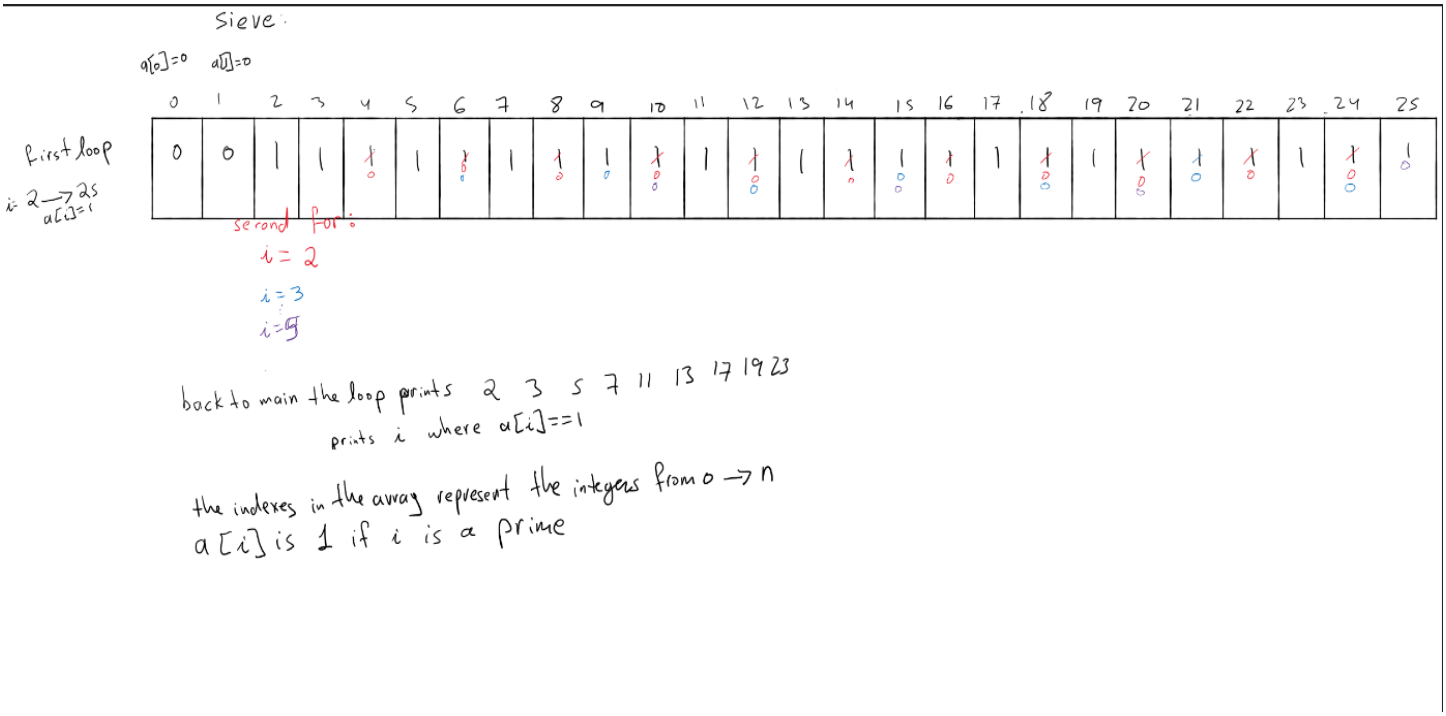
```c
#include <stdio.h>

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

int main(void)
{
        int n = 111;
        int a[n + 1];
        sieve(a, n + 1);
        printf("The primes 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;
                }
        }
}
```



The following is the output if in `main` we initialize `n` with `25`:



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
```

Important Note:

Notice in the code that we passed an array and modified our original array. What's being passed to the function is not the array but rather where the array exists in memory. Thus, when you make changes inside the array, it changes the memory location in which our array was specified to live. More on this later.

sizeof Operator

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

Example:

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.    int a[] = { -1, 2, -1, 2, -4, 1 };
6.
7.   // since all values in an array are of the same type
8.   // the following printf will print the length of array a
9.   // which is the number of elements in a (6).
10.   printf("%d\n", sizeof(a) / sizeof(a[0]));
11.
12.  // output: 1 4 4 24
13.   printf("%d %d %d %d\n", sizeof(char), sizeof(int), sizeof(a[0]), sizeof(a));
14.
15.
16.  // The %zu is preferred for cross-platform compatibility
17.  //  (with 32 bit and 64 bit systems for type size_t)
18.  // output: 1 4 4 24
19.   printf("%zu %zu %zu %zu\n", sizeof(char), sizeof(int), sizeof(a[0]), sizeof(a));
20.
21.   return 0;
22.  }
```

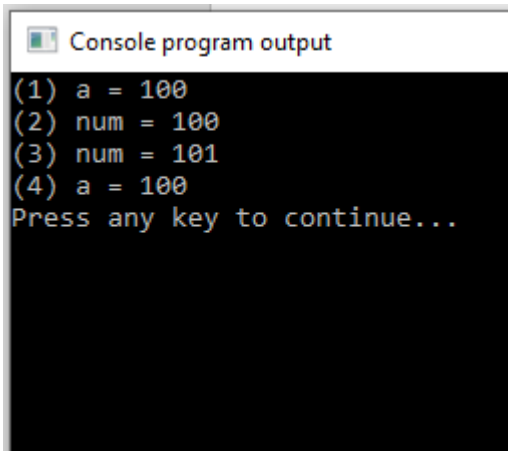
: `sizeof()` may give different output according to each machine; I have run the program on 64-bit gcc compiler.

The size of array `a` is 24 because it includes 6 integers; the size of each integer is 4; thus, 6*4 = 24.

Passing Arrays to Functions

Consider the following example and the output:

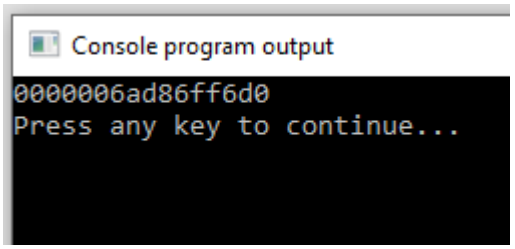
```
1. #include <stdio.h>
2.
3. void plusOne(int num)
4. {
5.          printf("(2) num = %d\n", num);
6.          num++;
7.          printf("(3) num = %d\n", num);
8. }
9.
10.  int main(void)
11.  {
12.         int a = 100;
13.         printf("(1) a = %d\n", a);
14.         plusOne(a);
15.         printf("(4) a = %d\n", a);
16.         return 0;
17.  }
```



The output indicates that a copy of the value of `a` was passed to `num`; thus, any change to `num` did not affect the value of `a`. All right… **Keep this in mind for later**.

Now consider the following example and the output:

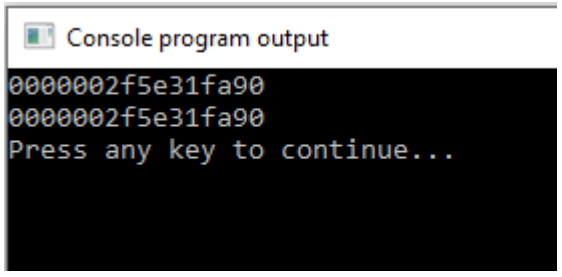
```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.   int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
6.
7.   printf("%p\n", a);
8.   return 0;
9. }
```



Note: %p to print a memory address value.

The output indicates that an array (by name) is a pointer to something in memory (In fact, it points at the first value in the array, *check the following program*). The output is the address in memory that `a` is pointing at. (A pointer in C is a type that stores a memory address. *When a pointer has a specific memory address, we say the pointer is pointing at the value in that particular address*. More about that later on).

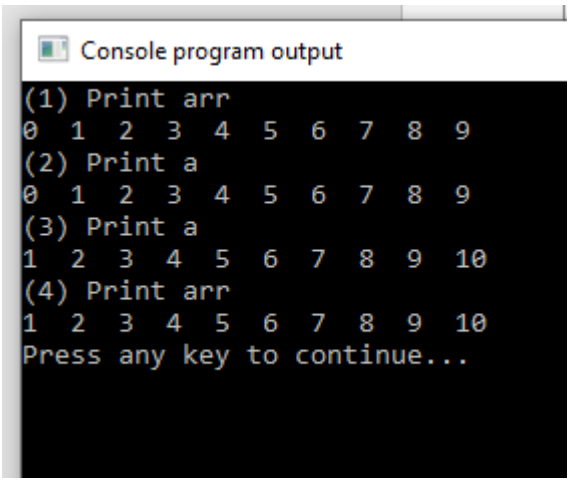
```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.   int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
6.   printf("%p\n", a);
7.   printf("%p\n", &a[0]);
8.   return 0;
9. }
10.
```



All right… Keep this in mind when you review the following program and its output.

Let's take a close look at passing an array into a function.

```
1. #include <stdio.h>
2.
3. void plusOne(int a[], int n)
4. {
5.          printf("(2) Print a\n");
6.          for (int i = 0; i < n; i++)
7.                  printf("%d  ", a[i]);
8.          printf("\n");
9.
10.         for (int i = 0; i < n; i++)
11.                 a[i] ++;
12.
13.         printf("(3) Print a\n");
14.         for (int i = 0; i < n; i++)
15.                 printf("%d  ", a[i]);
16.         printf("\n");
17. }
18.
19. int main(void)
20. {
21.         int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
22.         printf("(1) Print arr\n");
23.         for (int i = 0; i < 10; i++)
24.                 printf("%d  ", arr[i]);
25.         printf("\n");
26.
27.         plusOne(arr, 10);
28.
29.         printf("(4) Print arr\n");
30.         for (int i = 0; i < 10; i++)
31.                 printf("%d  ", arr[i]);
32.         printf("\n");
33.         return 0;
34. }
```

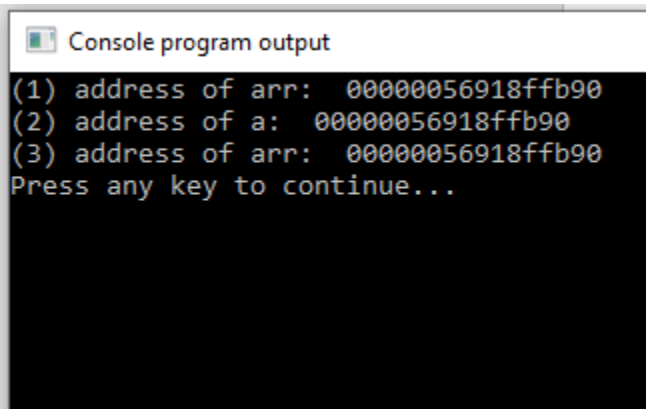


When `plusOne` is called, a pointer to the values of `arr` is passed to `a`, which means that **both** `arr` and `a` are pointing to the same data. Thus, updating array `a` was updating array `arr`.

The following example illustrates that both arr and a point to the same data by checking the addresses (location) of `arr` and `a`.

```c
#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;
}
```

Output:



In `C`, you can define a pointer to an integer: `int *p`; `p` can point to a value of type integer, precisely the same as saying that `p` has an address in memory where an integer is stored in that memory.

From all that we've learned so far, the following two definitions are equivalent in C:

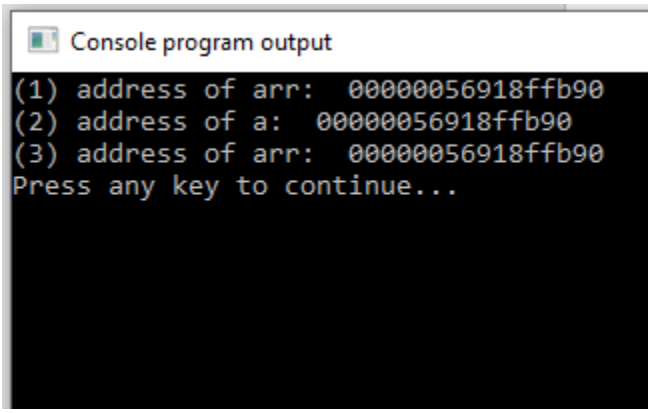
`void plusOne(int a[], int n);` is equivalent to `void plusOne(int *a, int n);`

Note: They are equivalent because the array name is a pointer to the first element in the array

Let us rewrite the last two examples using `int *a` instead of `int a[]`

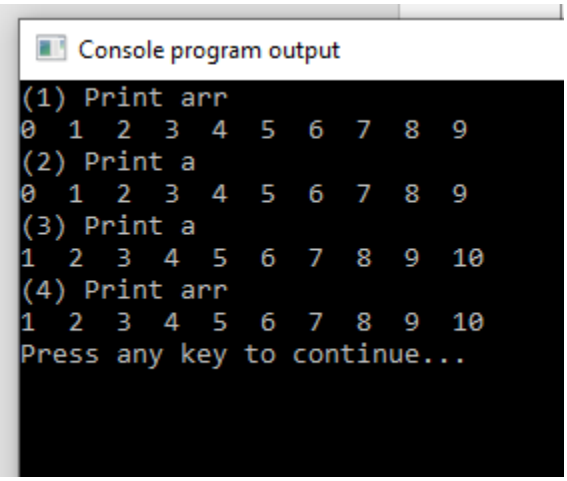
```
1. #include <stdio.h>
2.
3. void plusOne(int *a, int n)
4. {
5.   printf("(2) address of a:  %p\n", a);
6.   for (int i = 0; i < n; i++)
7.           a[i] ++;
8. }
9.
10.  int main(void)
11.  {
12.          int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
13.          printf("(1) address of arr:  %p\n", arr);
14.          plusOne(arr, 10);
15.          printf("(3) address of arr:  %p\n", arr);
16.          return 0;
17.  }
```

Output:



```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;
}
```

Output:

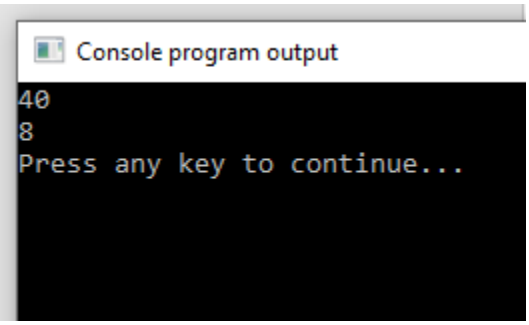


Caution! Pointer decay

What does the following program print?

```
1. #include <stdio.h>
2.
3. void sizeofArray(int a[])
4. {
5.          printf("%zu\n", sizeof(a));  // output: 8
6. }
7.
8. int main(void)
9. {
10.         int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
11.         printf("%zu\n", sizeof(a)); // output: 40
12.         sizeofArray(a);
13.         return 0;
14.  }
```

Output:

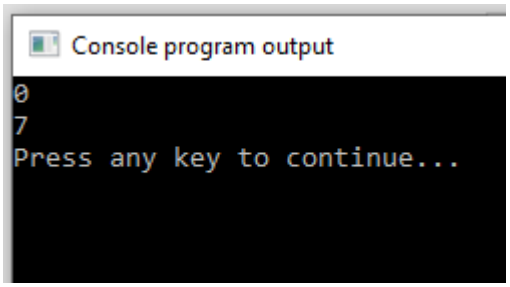


Probably not what you expected. Remember that the `a` in the function is really just a memory address, and so has the size of a memory address location (which in this case is 8 bytes). The `sizeof(a)` inside `main` is the size of the array (so the size of all the elements in the array). In this case, this is 10 times `sizeof(int)`. This awkwardness is why we need to pass the size of the array to functions. This is called **pointer decay**.

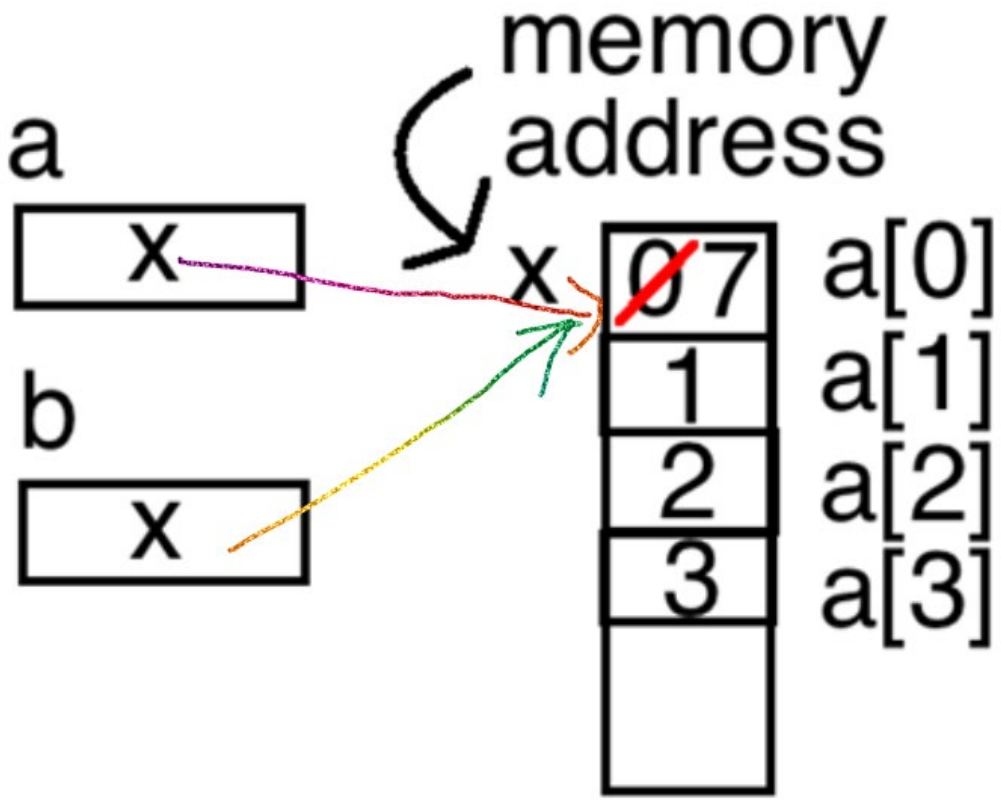
More about pointers - References

What does the following program print?

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.          int a[] = { 0, 1, 2, 3 };
6.          int *b = a;
7.          printf("%d\n", a[0]);
8.          b[0] = 7;
9.          printf("%d\n", a[0]);
10.         return 0;
11.  }
```



Why? Because `b` is pointing to the exact location where `a` is pointing at, the array of 4 integers. When we changed the value that `b` is pointing at, we changed the value in the array `a`.



`x` represents the address in memory where the first element of array `a` is allocated.

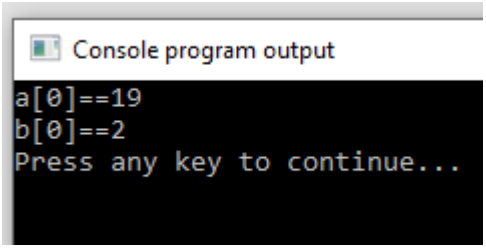
Copying Arrays

We want an exact copy of an array that is distinct (no reference; changing one array doesn't change the other).

One way to achieve this is to create a new array of the same size and manually copy everything into it.

Below are two versions:

```c
#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 case
        return 0;
}

```

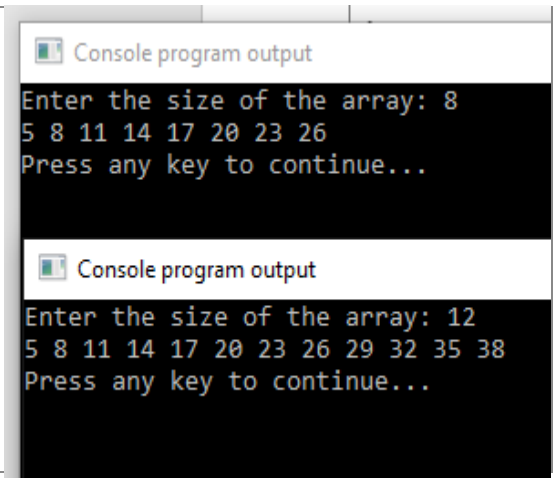


Variable-length array

In C, a variable-length array (VLA), also called variable-sized or runtime-sized, is an array data structure whose length is determined at run time (instead of at compile time). The official C standard rendered this feature as "optional" in C11 standard). Therefore, we will not be using variable-length arrays in this course. **Any program you submit using variable length arrays in this course will receive 0 unless allocated in the heap (we will learn later about this).**

Example of VLA:

```c
#include <stdio.h>

int main(void)
{
        int n;
        printf("Enter the size of the array: ");
        scanf("%d",&n);
        int arr[n];
        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 learned so far about arrays is what we call linear arrays, which all deal with a single dimension. The C language allows arrays of any dimension to be defined. In this section, we will learn about two-dimensional arrays. One of the most natural applications for a two-dimensional array arises in the matrix case.

The following is a program that defines an array `matrix` that has four rows and three columns. The program also calculates the sum of all the elements in `matrix`.

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int matrix[4][3] = { {0, 1, 2},
6.                          {10, 11, 12},
7.                          {20, 21, 22},
8.                          {30, 31, 32}
9.                          };
10.    int sum = 0;
11.    for (int i = 0; i < 4; i++)
12.    {
13.          for (int j = 0; j < 3; j++)
14.          {
15.                sum += matrix[i][j];
16.          }
17.    }
18.    printf("Sum = %d\n", sum);
19.    return 0;
20. }
```

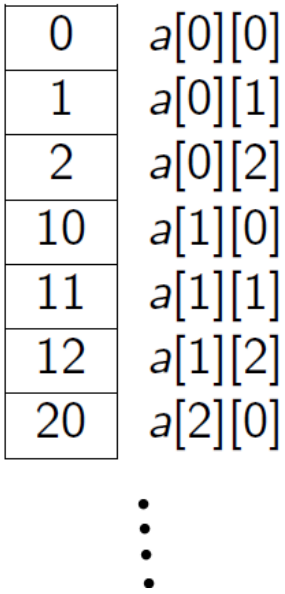
Conceptually:

0	1	2
10	11	12
20	21	22
30	31	32

Tracing:

```
   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:



Thus, the same program can be written like this (**not commonly used, as the suggestion above is more readable**):

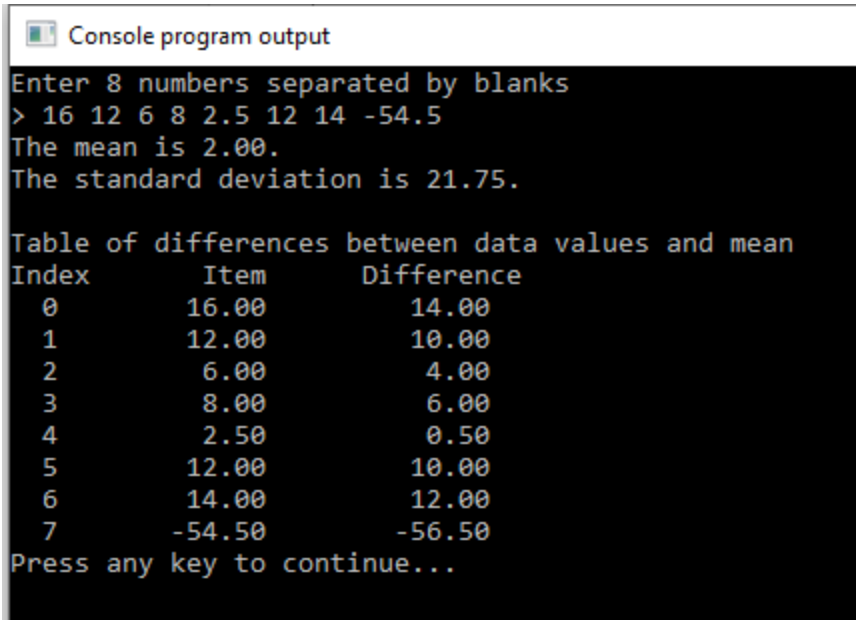
```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int matrix[4][3] = { {0, 1, 2},
6.                          {10, 11, 12},
7.                          {20, 21, 22},
8.                          {30, 31, 32}
9.                          };
10.    int sum = 0;
11.    for (int i = 0; i < 4; i++)
12.    {
13.          for (int j = 0; j < 3; j++)
14.          {
15.                  sum += matrix[0][i * 3 + j];
16.          }
17.    }
18.    printf("Sum = %d\n", sum);
19.    return 0;
20.  }
```

When initializing, all dimensions except possibly the first must be explicit. This holds also in function definitions.
Example: `int a[][2] = {{1,2},{3,4}};`

We will return in a few weeks to discuss pointers in more detail.

Additional Examples

```c
/*
 * Computes the mean and standard deviation of an array of data and
 * displays the difference between each value and the mean.
 */

#include <stdio.h>
#include <math.h>
#define MAX_ITEM  8      /* maximum number of items in list of data */

int main(void)
{
        double x[MAX_ITEM],      /* data list  */
               mean,     /* mean (average) of the data */
               st_dev,   /* standard deviation of the data */
               sum,      /* sum of the data    */
               sum_sqr;  /* sum of the squares of the data */
        int i;

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

        /* Computes the sum and the sum of the squares of all data    */
        sum = 0;
        sum_sqr = 0;
        for (i = 0; i < MAX_ITEM; ++i)
        {
                sum += x[i];
                sum_sqr += x[i] * x[i];
        }

        /* Computes and prints the mean and standard deviation    */
        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);

        /* Displays the difference between each item and the mean */
        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);
}
```




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

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

/* Performs pivoting with respect to the pth row and the pth column
 *  If no nonzero pivot can be found, FALSE is sent back through piv_foundp */

void pivot(double aug[N][N + 1], /* input/output - augmented matrix        */
        int p,   /* input   - current row  */
        int *piv_foundp) /* output  - whether or not nonzero pivot found   */
{
        double xmax, xtemp;
        int j, k, max_row;

        /* Finds maximum pivot    */
        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;
                }
        }

        /* Swaps rows if nonzero pivot was found  */
        if (xmax == 0)
        {
                *piv_foundp = FALSE;
        }
        else
```

```
        {
                *piv_foundp = TRUE;
                if (max_row != p)
                {       /* swap rows */
                        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 aug. If no unique solution exists,
 *  sends back FALSE through sol_existsp */

void gauss(double aug[N][N + 1],  /* input/output - augmented matrix representing
                                     system of N equations  */
        int *sol_existsp)         /* output - flag indicating whether system has a
                                                unique solution    */
{
        int j, k, p;
        double piv_recip,         /*  reciprocal of pivot  */
               xmult;
        /*  System is assumed nonsingular;  moves down the diagonal   */
        *sol_existsp = TRUE;
        for (p = 0; *sol_existsp && p < (N - 1); ++p)
        {

                /*  Pivots with respect to the pth row and the pth column  */
                pivot(aug, p, sol_existsp);
                if (*sol_existsp)
                {
                        /*  Scales pivot row  */
                        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;

                        /*  Eliminates coefficients beneath pivot  */
                        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 last coefficient is zero, there is no unique solution */
        if (aug[N - 1][N - 1] == 0)
        {
                *sol_existsp = FALSE;
        }
        else if (*sol_existsp)
        {       /* Scales last row */
```

```
                piv_recip = 1.0 / aug[N - 1][N - 1];
                aug[N - 1][N - 1] = 1.0;
                aug[N - 1][N] *= piv_recip;
        }
}

/*  Performs back substitution to compute a solution vector to a system of
 *  linear equations represented by the augmented matrix aug. Assumes that
 *  the coefficient portion of the augmented matrix has been triangularized,
 *  and its diagonal values are all 1.  */

void back_sub(double aug[N][N + 1],      /* input -  scaled, triangularized augmented matrix   */
        double x[N])    /* output - solution vector   */
{
        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;
        }
}

/* Solves a system of linear equations using Gaussian elimination and back substitution */
 int main(void)
{
        double aug[N][N + 1] = { {1.0, 1.0, 1.0, 4.0},    /* augmented matrix   */
        {2.0, 3.0, 1.0, 9.0},
        {1.0, -1.0, -1.0, -2.0}
        }, x[N];/* solution vector */
        int sol_exists;  /* flag indicating whether or not a solution can be found  */

        /* Calls gauss to scale and triangularize coefficient matrix within aug       */
        gauss(aug, &sol_exists);

        /* Finds and displays solution if one 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);
}
```

Console program output

```
The values of x are:       1.00      2.00      1.00
Press any key to continue...
```

Extra Practice Problems

[***Try to solve and test the problems before looking at the provided solutions. Remember, you learn the most by doing and not by looking at solutions***].

1) Write a program[i] to read integers (in the range 0-100) and print how many times each integer was read sorted by the integers' values.

For example:

Input	Output
12 90 67 90 100 12 8 0 3	0 1 3 1 8 1 12 2 67 1 90 2 100 1

2) Write a program[ii] that reads an integer and prints how many times each digit appears in that integer. For example:

Input	Output
56065890	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.

```
1. #include <stdio.h>
2.
3. int max_array(int a[], int n)
4. {
5.   int max = 0;
6.   for (int i = 0; i < n; i++)
7.           if (a[i] > max)
8.                   max = a[i];
9.   return max;
10.  }
```

a) What is wrong with this solution? Which case will not provide the correct results?[iii]

b) Fix the function.

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, it returns the index of the first occurrence of the maximal value.

5) Write a C program, `sudoku_checker.c`, that checks if a given 9x9 Sudoku board is valid. The program should verify that the board adheres to standard Sudoku rules.

Here are the requirements for the program:

1. Input a 9x9 Sudoku board as a grid where each cell can contain a digit from 1 to 9. Use '0' to represent empty cells.
2. Implement a function to check if the Sudoku board is valid according to the following rules:
    - Each row must contain digits from 1 to 9 without repetition.
    - Each column must contain digits from 1 to 9 without repetition.
    - Each of the nine 3x3 sub-grids must contain digits from 1 to 9 without repetition.
3. Output whether the Sudoku board is valid or not.

Sample Input 1 (Valid Sudoku):
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

Sample Output 1 (Valid Sudoku): "The Sudoku board is valid."
Sample Input 2 (Invalid Sudoku):
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 8 9
Sample Output 2 (Invalid Sudoku): "The Sudoku board is invalid."

6) **Simplified Connect Four Game in C**

Write a C program to implement a simple two-player Connect Four game. The game should have the following components:

1. **Game Board**: Implement a 6x7 game board where two players take turns dropping their pieces ('X' and 'O') into columns.
2. **Game Logic**: Implement the game logic to check for a win when four pieces of the same player are connected horizontally in a row.

Here is some sample code for you to use and fill in:

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

#define ROWS 6
#define COLUMNS 7

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

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

// Display the current state of the game board
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");
}

// Check if the column is valid for a move
bool isValidMove(int column) {
    // your code here
}

// Place a piece in the specified column
void makeMove(int column) {
// your code here

}

// Check if the current player has won the game (horizontal check only)
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();
            if (currentPlayer == 'X') {
                currentPlayer = 'O';
            } else {
                currentPlayer = '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 of the 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 of the array (other than it is positive)

The following applies to all functions:

requires: all array parameters are valid (not NULL)

all array length parameters (e.g., len) are > 0

all arrays have a length >= length parameter [not asserted]

Complete `bool unique(int a[], int len);` which determines whether all of the elements of `a` are "unique," or in other words, if there are no duplicates in `a`.

examples: `{1, 3, 7} => true`

`{1, 3, 1} => false`

Complete `int longest_sorted(int a[], int len);` which finds the length of the longest consecutive sequence or "subarray" that contains elements that are sorted in either non-descending (e.g., ascending) or non-increasing order

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);` which gives a non-empty array of integers `nums`, every element appears twice except for one. Find that single one.

Example 1:

Input: `nums = [2, 2, 1]`

Output: `1`

Example 2:

Input: `nums = [4, 1, 2, 1, 2]`

Output: `4`

Example 3:

Input: `nums = [1]`

Output: `1`

9) Develop a C program, `survey_stats.c`, to analyze survey data using two-dimensional arrays and calculate basic statistics.

Create a function to input survey data into a 2D array, where rows represent respondents and columns represent survey items. Write a function to compute the average rating for each item. And Implement functions to find the highest and lowest ratings in the dataset.

Use these functions in another function called `surveyAnalyzer` to display average ratings and the items with the highest and lowest ratings.

Assume each respondent rates all items. Ratings are integers. The maximum number of respondents and the maximum number of items are 100. The first input is the number of respondents, then the number of items, then the ratings for each item for each respondent.

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)

Solve the question by completing the following program:

```
#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) Recall from your notes that you can conceptualize a 2-dimensional array as follows.

$$\begin{bmatrix} 0 & 1 & 2 & \dots & m-1 \\ 10 & 11 & 12 & \dots & \dots \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ (n-1)1 & \dots & \dots & \dots & (n-1)(m-1) \end{bmatrix}$$

Suppose you are given a 2-dimensional matrix filled with either ones or zeros.

The following pattern is called a smiley (hopefully, it bares a slight resemblance to a smiley face!):

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

Write a function `bool smiley(int n, int m, int data[n][m])` that returns `true` if a smiley face exists in the given 2-dimensional array and is `false` otherwise. Let `n>=5` be the number of elements in the first dimension (column number), and let `m>=5` be the number of elements in the second (row number). (`data[n][m]`)

11) You have a water tank that can provide exactly `M` liters of water.

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.

Answers

---

i

```c
#include <stdio.h>

int main(void)
{
    int a[101] = { 0 };
    int num;
    printf("Enter integers: ");
    while (scanf("%d", &num) == 1)
    {
        a[num] ++;
    }
    printf("The output:\n");
    for (int i = 0; i <= 100; i++)
    {
        if (a[i] > 0)
            printf("%d %d\n", i, a[i]);
    }
    printf("\n");
    return 0;
}
```

ii

```c
#include <stdio.h>

int main(void)
{
    int a[10] = { 0 };
    int num;
    printf("Enter  an integer: ");
    scanf("%d", &num);
    while (num > 0)
    {
        a[num % 10] ++;
        num = num / 10;
    }
    printf("The output:\n");
    for (int i = 0; i < 10; i++)
    {
        printf("%d %d\n", i, a[i]);
    }
    printf("\n");
    return 0;
}
```

[iii] When all integers are negative the function returns 0 which is incorrect!