
Arrays

- Arrays enabling storing multiple values under a single variable
- If multiple values are stored in a single variable, we need a way to access each value
- We access values stored in an array using **indices**, called *subscripts*
- Values inside of an array are *homogeneous*, meaning they all have the same type
 - Can't mix **ints** with **floats** or vice-versa
- Later we will introduce the idea of a *pointer*, which extend the use of arrays

Declaration and Initialization

- C arrays are declared in the following form

```
1 type name[number of elements];
```

- `type` specifies the type of every element in the array (since arrays are homogeneous, we only specify one type)
- `name` is the identifier/variable name we will use to refer to the array
- `number of elements` is the number of `type` elements that the array can store
- To declare an array of 6 integers called `numbers` we would use:

```
1 int numbers[6];
```

- To declare an array of 6 characters called `letters` we would use:

```
1 char letters[6];
```

- We can initialize the array when we declare it using curly braces and initialization values using an initializer list:

```
1 int point[6] = {0,3,1,6,7,2};
```

- Or we can only initialize the first few elements (this initializes the first 3):

```
1 int parital[6] = {1,2};
```

- We can also omit the size of the array and use the size of the initializer as the size of the array (this will have space for 6 integers):

```
1 int point[] = {0,3,1,6,7,2};
```

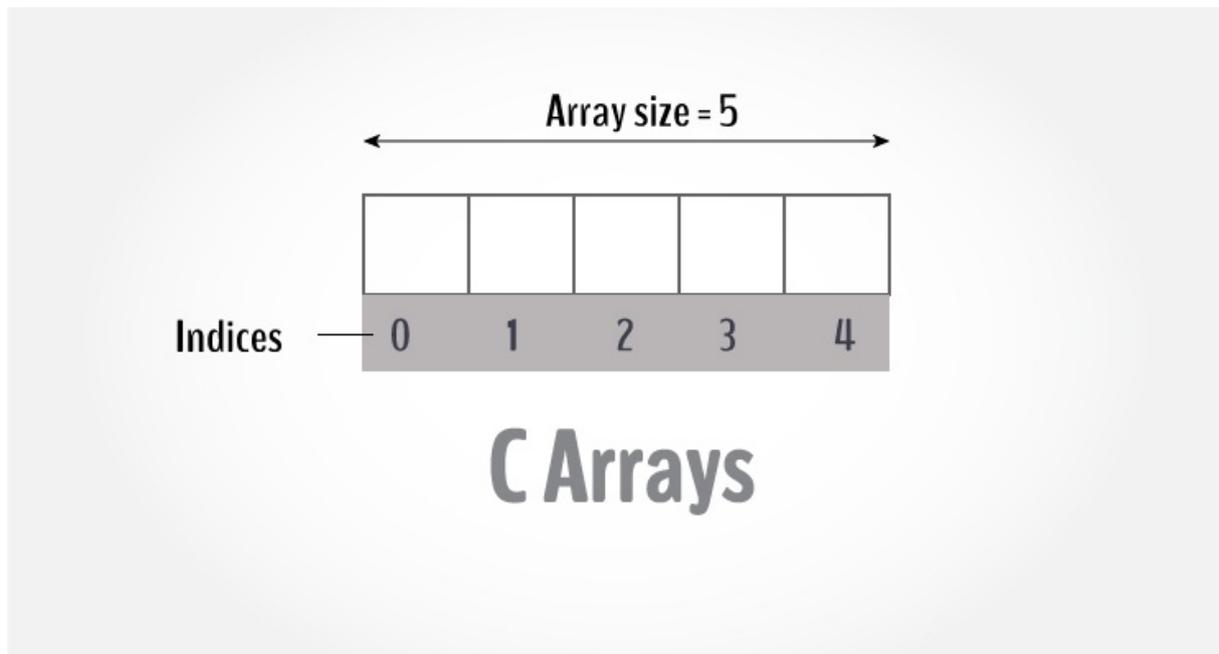


Figure 1: C Array

Array Access

- Now we know how to declare and initialize an array, but how do we access elements in an array?
- We'll use an *index* or *subscript* to specify which element of the array we want to access
- Arrays are 0-indexed in C, meaning the index of the first element in the array is 0, the second element in the array is 1, the third is 2, and so forth.
 - Important note: the last **valid** index in an array is the size-1. For instance, in an array of length 6 (an array that can store 6 elements), the last valid index is 5. 0-5 is 6 numbers.
- Example:

```
1 int point[6] = {0,3,1,6,7,2};  
2 int thirdEle = point[2]; // arrays are 0-indexed in C, so thirdEle will  
   have the value of 1
```

- What happens if you access an array with an index is out of the bounds of the array (i.e. use 6 as an index to the `point` array)?
 - It depends. Sometimes the compiler can catch the error, but it's not guaranteed to.
 - If your program executes, it will be in *undefined behavior* (UB), which means the rest of your program's output is rendered meaningless and unpredictable, even if it outputs the correct thing

-
- * Undefined behavior is a large and somewhat esoteric definition, but the point is that C makes zero guarantee about what will happen after you've triggered undefined behavior.

- Examples:

```
1 char y;
2 int z = 9;
3 char point[6] = { 1, 2, 3, 4, 5, 6 };
4 //examples of accessing outside the array. A compile error is not
  always raised
5 y = point[15];
6 y = point[-4];
7 y = point[z];
```

- Your program may continue running normally after these cases, but you have entered UB. This must be avoided at all costs!
- But there's got to be a better way to make sure we stay within the bounds...
 - Well not for every case, but for any type of loop, we can use `sizeof()` to as the limit on the number of iterations the loop executes
 - Here's an example:

```
1 int i;
2 int arr[] = {3, 6, 9, 12, 15};
3
4 printf("sizeof(arr): %lu\n", sizeof(arr));
5 printf("sizeof(int): %lu\n", sizeof(int));
6
7 int arr_len = sizeof(arr) / sizeof(int);
8
9 printf("array is length %d\n", arr_len);
10
11 for (i = 0; i < arr_len; ++i)
12 {
13     printf("arr[%d]: %d\n", i, arr[i]);
14 }
```

- This is a great way to ensure you stay within the bounds of the array!

Passing arrays to functions

- To pass an array to a function, we'll pass the name of the variable of the array.

-
- However, in the function signature, we must tell the compiler we are passing an array:

```
1 #include <stdio.h>
2
3 // [] after the variable name indicates the variable is an array
4 float average(float arr[], size_t arr_len);
5
6 int main()
7 {
8     float avg;
9     float arr[] = { 23.4, 55, 22.6, 3, 40.5, 18 };
10    int arr_len = sizeof(arr) / sizeof(float);
11
12    avg = average(arr, arr_len); /* Only name of array is passed as
13    argument. */
14
15    printf("Average age=%.2f", avg);
16    return 0;
17 }
18 // [] after the variable name indicates the variable is an array
19 float average(float arr[], size_t arr_len)
20 {
21     int i;
22     float avg, sum = 0.0;
23     for (i = 0; i < arr_len; ++i) {
24         sum += arr[i];
25     }
26     avg = (sum / 6);
27     return avg;
28 }
```

Returning arrays from functions

- We'll have to introduce a symbol we will talk in greater detail about when we discuss pointers and passing-by-reference. We need to cover this for the homework assignment, but the concept will be covered later.
- We'll use the pointer type-qualifier `*` as a part of the return type to indicate we wish to return an array.
- Inside of the function, we'll return the symbol of the array **without accessing an element using an index**

-
- Example:

```
1 //NOTICE: the asterisk (star) next to int indicates we are returning an
  array
2 int* add_to_zeroth_element(int arr[], size_t arr_len, int value){
3   // this is just a dummy array operation, in practice you'll do
  wonderful and amazing things here
4   arr[0] += value;
5   // NOTICE: return the array, we don't use [] here, just the name of
  the array.
6   return arr;
7 }
8
9 int main(){
10  int arr[] = {1,2,3};
11  // notice the type here has to match the return type of the function.
  Exactly what's going on here will be covered with pointers.
12  int* result = add_to_zeroth_element(arr, 3, 5);
13 }
```

Scope

- Lifetime of a variable
- Variables in callee's are not visible to the caller, and when the callee finishes, all local variables are freed from memory (meaning they will not exist in the caller).

Multi-dimensional arrays

- Multi-dimensional arrays are arrays-of-arrays.
- The most basic multi-dimensional is a 2-dimensional array, which creates a rectangular array. Each row has the same number of columns.
- To get an int array with 3 rows and 5 columns, we write:

```
1 int arr[3][5];
```

- To access/modify a value in the array, we need two subscripts: one for the row we wish to access, and a second for the column we wish to access:

```
1 arr[1][3] = 5; // sets the element in the second row and fourth column
  to 5
```

-
- We can also initialize a multi-dimensional array in a similar fashion as a single-dimension array using an initializer list:

```
1 int two_d[2][3] = {{ 5, 2, 1 },
2                 { 6, 7, 8 }};
```

- The amount of columns must be explicitly specified, but the compiler will sort out how many rows are needed based on the initializer list. We could have written

```
1 int two_d[][3] = {{ 5, 2, 1 },
2                 { 6, 7, 8 }};
```

Passing multi-dimensional arrays to functions

- Exactly the same as passing single-dimension, except we must specify the number of columns
 - Can also specify both rows and columns if you only want a

```
1 #include <stdio.h>
2 void print_arr(int num[][2]);
3 int main()
4 {
5     const int nr=2, nc=2;
6     int num[nr][nc], i, j;
7     for (i = 0; i < nr; i++)
8     {
9         for (j = 0; j < nc; j++)
10        {
11            printf("element - [%d][%d]: ", i, j);
12            scanf("%d", &num[i][j]);
13        }
14    }
15    // passing multi-dimensional array to function
16    print_arr(num, nr);
17
18    return 0;
19 }
20
21 void print_arr(int num[][2], size_t num_len)
22 {
23     int i, j;
24     for (i = 0; i < num_len; ++i)
25     {
```

```
26     for (j = 0; j < 2; ++j)
27     {
28         printf("%d ", num[i][j]);
29     }
30     printf("\n");
31 }
32 }
```

Returning multi-dimensional arrays from functions

- This is a bit trickier and we will cover this when we cover pointers

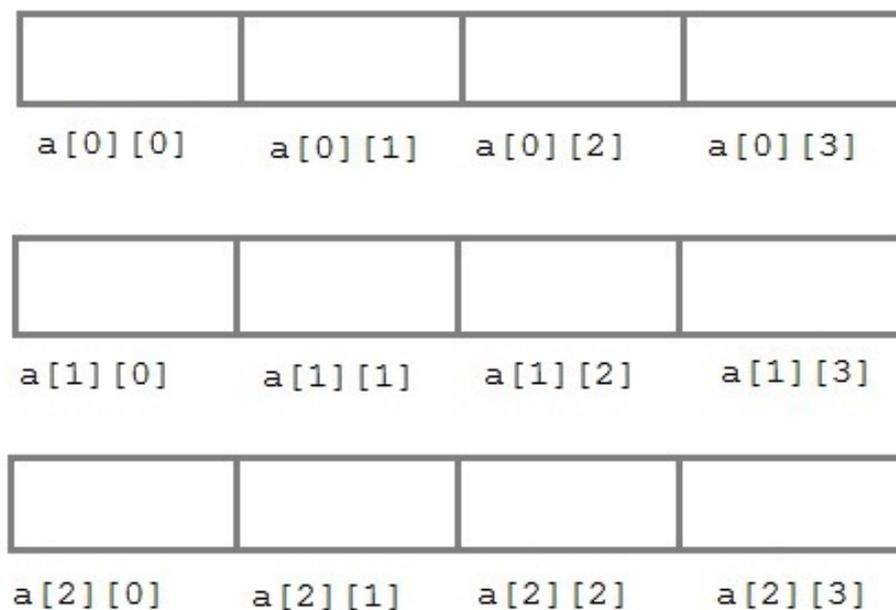


Figure 2: Multi-dimensional arrays

Exercises

1. Write a program in C to store 10 elements inputted by the user and write a function to print the contents of the array.

```
1 #include <stdio.h>
2
```

```

3  int main()
4  {
5      int arr[10];
6      int i;
7      printf("\n\nRead and Print elements of an array:\n");
8      printf("-----\n");
9
10     printf("Input 10 elements in the array :\n");
11     for(i=0; i<10; i++)
12     {
13         printf("element - %d : ",i);
14         scanf("%d", &arr[i]);
15     }
16
17     printf("\nElements in array are: ");
18     for(i=0; i<10; i++)
19     {
20         printf("%d ", arr[i]);
21     }
22     printf("\n");
23 }

```

2. Write a program in C to prompt for the number of elements the user wishes to input ($n < 100$) and then prompt for the user to input each element. Then print all unique elements in an array.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int arr1[100], n, count_ele = 0;
6      int i, j, k;
7
8      printf("Input the number of elements to be stored in the array (must
9          be less than 100):");
10     scanf("%d", &n);
11
12     printf("Input %d elements in the array:\n", n);
13     for (i = 0; i < n; i++)
14     {
15         printf("element - %d : ", i);
16         scanf("%d", &arr1[i]);
17     }

```

```

18  /*Checking duplicate elements in the array */
19  printf("\nThe unique elements found in the array are: \n");
20  for (i = 0; i < n; i++)
21  {
22      count_ele = 0;
23
24      /*Check duplicate before the current position and
25       increase counter by 1 if found.*/
26      for (j = i - 1; j >= 0; j--)
27      {
28          /*Increment the counter when the search value is duplicate.*/
29          if (arr1[i] == arr1[j])
30          {
31              count_ele++;
32          }
33      }
34      /*Check duplicate after the current position and increase counter
35       by 1 if found.*/
36      for (k = i + 1; k < n; k++)
37      {
38          /*Increment the counter when the search value is duplicate.*/
39          if (arr1[i] == arr1[k])
40          {
41              count_ele++;
42          }
43      }
44      /*Print the value of the current position of the array as unique
45       value
46       when counter remain contains its initial value (zero).*/
47      if (count_ele == 0)
48      {
49          printf("%d ", arr1[i]);
50      }
51  }
52  printf("\n\n");
53  }

```

- Write a program in C to store a 2x2 2-dimensional array. Elements are inputted by the user. Print the matrix and find the sum of rows and columns of the matrix.

```

1  #include <stdio.h>
2
3  int main()

```

```

4 {
5     const int n = 2;
6     int i, j, k, arr1[n][n], rsum[n], csum[n];
7
8     printf("Input elements in the 2x2 matrix:\n");
9     for (i = 0; i < n; i++)
10    {
11        for (j = 0; j < n; j++)
12        {
13            printf("element - [%d][%d]: ", i, j);
14            scanf("%d", &arr1[i][j]);
15        }
16    }
17    printf("The matrix is:\n");
18    for (i = 0; i < n; i++)
19    {
20        for (j = 0; j < n; j++)
21            printf("% 4d", arr1[i][j]);
22        printf("\n");
23    }
24
25    /* Sum of rows */
26    for (i = 0; i < n; i++)
27    {
28        rsum[i] = 0;
29        for (j = 0; j < n; j++)
30            rsum[i] = rsum[i] + arr1[i][j];
31    }
32
33    /* Sum of Column */
34    for (i = 0; i < n; i++)
35    {
36        csum[i] = 0;
37        for (j = 0; j < n; j++)
38            csum[i] = csum[i] + arr1[j][i];
39    }
40
41    printf("The sum of the rows the matrix is:\n");
42    for (i = 0; i < n; i++)
43    {
44        printf("% 4d", rsum[i]);
45        printf("\n");
46    }

```

```
47  printf("\n");
48  printf("The sum of the cols the matrix is: \n");
49  for (j = 0; j < n; j++)
50  {
51      printf("% 4d", csum[j]);
52  }
53  printf("\n\n");
54 }
```