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 `int`s with `float`s 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

```
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:

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

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

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

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

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

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

```
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):

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

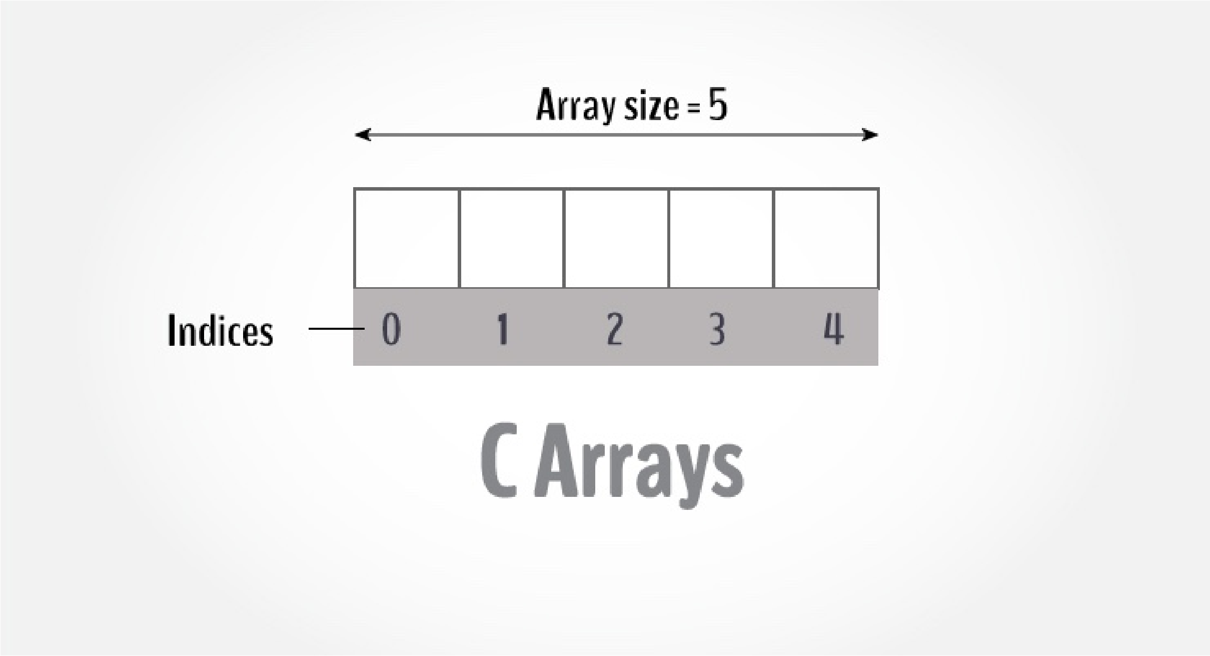

Figure 1: C Array

Array Access

- Now we know how to declare and initialize and 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:

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

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

```c
int i;
int arr[] = {3, 6, 9, 12, 15};

printf("sizeof(arr): %lu\n", sizeof(arr));
printf("sizeof(int): %lu\n", sizeof(int));

int arr_len = sizeof(arr) / sizeof(int);

printf("array is length %d\n", arr_len);

for (i = 0; i < arr_len; ++i)
{
  printf("arr[%d]: %d\n", i, arr[i]);
}
```

- 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:

```c
#include <stdio.h>

// [] after the variable name indicates the variable is an array
float average(float arr[], size_t arr_len);

int main()
{
    float avg;
    float arr[] = { 23.4, 55, 22.6, 3, 40.5, 18 };
    int arr_len = sizeof(arr) / sizeof(float);

    avg = average(arr, arr_len); /* Only name of array is passed as argument. */

    printf("Average age=%.2f", avg);
    return 0;
}

// [] after the variable name indicates the variable is an array
float average(float arr[], size_t arr_len)
{
    int i;
    float avg, sum = 0.0;
    for (i = 0; i < arr_len; ++i) {
        sum += arr[i];
    }
    avg = (sum / 6);
    return avg;
}
```

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-qualifer `*` 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:

```c
//NOTICE: the asterisk (star) next to int indicates we are returning an array
int* add_to_zeroth_element(int arr[], size_t arr_len, int value){
  // this is just a dummy array operation, in practice you'll do wonderful and amazing things here
  arr[0] += value;
  // NOTICE: return the array, we don't use [] here, just the name of the array.
  return arr;
}

int main(){
  int arr[] = {1,2,3};
  // notice the type here has to match the return type of the function. Exactly what's going on here will be covered with pointers.
  int* result = add_to_zeroth_element(arr, 3, 5);
}
```

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

- Muti-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:

```c
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:

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

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

```c
int two_d[2][3] = {{ 5, 2, 1 },
                   { 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

```c
int two_d[][3] = {{ 5, 2, 1 },
                  { 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

```c
#include <stdio.h>
void print_arr(int num[][2]);
int main()
{
  const int nr=2, nc=2;
  int num[nr][nc], i, j;
  for (i = 0; i < nr; i++)
  {
    for (j = 0; j < nc; j++)
    {
      printf("element - [%d][%d]: ", i, j);
      scanf("%d", &num[i][j]);
    }
  }
  // passing multi-dimensional array to function
  print_arr(num, nr);

  return 0;
}

void print_arr(int num[][2], size_t num_len)
{
  int i, j;
  for (i = 0; i < num_len; ++i)
  {
    for (j = 0; j < 2; ++j)
    {
      printf("%d  ", num[i][j]);
    }
    printf("\n");
  }
}
```

Returning multi-dimensional arrays from functions

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

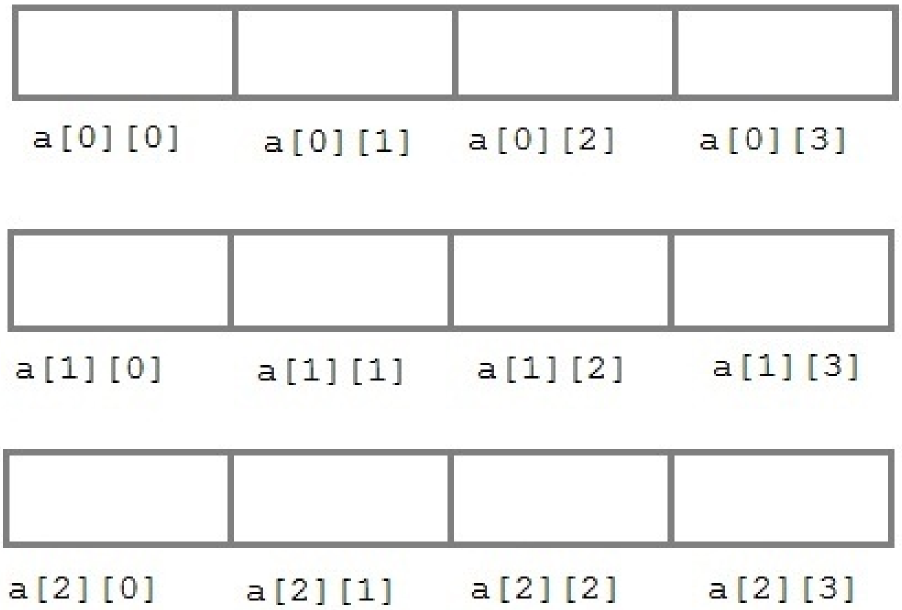

Figure 2: Multi-dimensional arrays

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.

```
#include <stdio.h>

```

```c
int  main()
{
  int arr[10];
  int i;
  printf("\n\nRead and Print elements of an array:\n");
  printf("-----------------------------------------\n");

  printf("Input 10 elements in the array :\n");
  for(i=0; i<10; i++)
  {
    printf("element - %d : ",i);
    scanf("%d", &arr[i]);
  }

  printf("\nElements in array are: ");
  for(i=0; i<10; i++)
  {
    printf("%d  ", arr[i]);
  }
  printf("\n");
}
```

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.

```c
#include <stdio.h>

int main()
{
  int arr1[100], n, count_ele = 0;
  int i, j, k;

  printf("Input the number of elements to be stored in the array (must be less than 100):");
  scanf("%d", &n);

  printf("Input %d elements in the array:\n", n);
  for (i = 0; i < n; i++)
  {
    printf("element - %d : ", i);
    scanf("%d", &arr1[i]);
  }

  /*Checking duplicate elements in the array */
  printf("\nThe unique elements found in the array are: \n");
  for (i = 0; i < n; i++)
  {
    count_ele = 0;

    /*Check duplicate before the current position and
     increase counter by 1 if found.*/
    for (j = i - 1; j >= 0; j--)
    {
      /*Increment the counter when the search value is duplicate.*/
      if (arr1[i] == arr1[j])
      {
        count_ele++;
      }
    }
    /*Check duplicate after the current position and increase counter by 1 if found.*/
    for (k = i + 1; k < n; k++)
    {
      /*Increment the counter when the search value is duplicate.*/
      if (arr1[i] == arr1[k])
      {
        count_ele++;
      }
    }
    /*Print the value of the current position of the array as unique value
     when counter remain contains its initial value (zero).*/
    if (count_ele == 0)
    {
      printf("%d ", arr1[i]);
    }
  }
  printf("\n\n");
}
```

3. 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 an columns of the matrix.

```
#include <stdio.h>

int main()
```

```
{
  const int n = 2;
  int i, j, k, arr1[n][n], rsum[n], csum[n];

  printf("Input elements in the 2x2 matrix:\n");
  for (i = 0; i < n; i++)
  {
    for (j = 0; j < n; j++)
    {
      printf("element - [%d][%d]: ", i, j);
      scanf("%d", &arr1[i][j]);
    }
  }
  printf("The matrix is:\n");
  for (i = 0; i < n; i++)
  {
    for (j = 0; j < n; j++)
      printf("% 4d", arr1[i][j]);
    printf("\n");
  }

  /* Sum of rows */
  for (i = 0; i < n; i++)
  {
    rsum[i] = 0;
    for (j = 0; j < n; j++)
      rsum[i] = rsum[i] + arr1[i][j];
  }

  /* Sum of Column */
  for (i = 0; i < n; i++)
  {
    csum[i] = 0;
    for (j = 0; j < n; j++)
      csum[i] = csum[i] + arr1[j][i];
  }

  printf("The sum of the rows the matrix is:\n");
  for (i = 0; i < n; i++)
  {
    printf("% 4d", rsum[i]);
    printf("\n");
  }
```

```c
  printf("\n");
  printf("The sum of the cols the matrix is: \n");
  for (j = 0; j < n; j++)
  {
    printf("% 4d", csum[j]);
  }
  printf("\n\n");
}
```