Arrays are lists of variables of the same type that are placed next to each other in memory and accessible from a **single variable** by **indexing**.

Their syntax is:

```
datatype name [size];
```

The size must be **constant** and known at **compile time.**

They can be initialized with values **in {} separated by commas.**

For example:

```
//all elements are initialized
int arr[5] = {1, 2, 3, 4, 5};

//first three are initialized
// and the rest are zeros
int arr[5] = {1, 2, 3};

//size of the array is 5
//which is the size of the initializer
int arr[] = {1, 2, 3, 4, 5};
```

Array elements can be accessed by indexing using **[index]** syntax.

For example:

```
arr[3] = arr[1] + arr[2];
cout << arr[3];
```

Processing arrays is usually done with **for loops** to access each element of the array individually.

For example:

```
const int SIZE = 5; // constant size only
int arr[SIZE] = {1, 2, 3, 4, 5};

int sum = 0;
for(int i = 0; i < SIZE; i++){
    sum += arr[i];
}
cout << "sum = " << sum << endl;
```

we can replace the above loop with a **for each** loop which is usually preferred:

```
for(int element: arr){
    sum += element;
}
```

Arrays can't be copied directly, instead each element has to be copied individually:

```
int arr_copy[SIZE];
for(int i = 0; i < SIZE; i++){
    arr_copy[i] = arr[i];
}
```

Passing arrays to functions is done by passing the memory address of the first element of the array to the function.

And this is essentially passing by **reference** and **no new arrays are created** thus any change to the array inside the function **reflects on the original array.**

We can't deduce the size of the array inside the function and we have to pass it separately to the function.

```
//adds one to all elements in the array
void add_one(int arr[], int SIZE) {
    for(int i = 0; i < SIZE; i++)
        arr[i] = arr[i] + 1;
    //the change is reflected on the original array
}
```

To forbid a function from changing the content of an array we can pass it as a **const**:

```
//this function is forbidden from changing the
//content of array arr
void add_one(const int arr[], int SIZE) {
    for(int i = 0; i < SIZE; i++)
        arr[i] = arr[i] + 1;
    //this will produce an error and won't compile
}
```

We can’t return arrays from functions such a function is forbidden:

```
//this is illegal
int[] reversed(int arr[], const int SIZE);
```

for the function like the one above who returns a new array that is a reverse of the original array we can instead pass the reverse array as a parameter and the function can fill it’s content with the reversed values.

```
void reversed(int arr[], int reverse[], const int SIZE);
```

C-strings are essentially **arrays of char** that is terminated by the **null character ‘\0’**

They can be initialized by the array initializer {} or by the “” syntax

```
//this is legal (notice the ‘\0’ at the end)
char str[] = {'h', 'e', 'l', 'l', 'o', '\0'};

//this is also legal (notice the lack of ‘\0’)
char str[] = "hello";
```

Reading and Writing **c-string** is very similar to reading and writing a normal **string**.

We can write a c string exsactly as we write a string.

```
char cstr[] = "hello world";
cout << cstr;
```

reading a c-string is also similar but with some exceptions.

- We have to make sure the size of the c-string is big enough to fit the string we are reading
- We have to make sure to have one extra space in the c-string for the **null character ‘\0**

For example:

```
char cstr[256]; // make sure it's big enough
cin >> cstr; //reads a single word in the line
getline(cin, cstr, '\n'); //reads the entire line
```

once the cstr is read from the screen, a **null character ‘\0’** is placed at the end to mark the end of the c-string.

We can get the length of the string using the **strlen** function.

```
char cstr[256];
cin >> cstr;
// entered " hello " to the terminal
cout << strlen(cstr); //outputs 5
```

C-String functions:
The C standard library has many functions to help deal with c-strings and the most common of which are:-

# C-String Functions

| *Function* | *Description* |
|---|---|
| `size_t strlen(char s[])` | Returns the length of the string, i.e., the number of the characters before the null terminator. |
| `strcpy(char s1[], const char s2[])` | Copies string s2 to string s1. |
| `strncpy(char s1[], const char s2[], size_t n)` | Copies the first n characters from string s2 to string s1. |
| `strcat(char s1[], const char s2[])` | Appends string s2 to s1. |
| `strncat(char s1[], const char s2[], size_t n)` | Appends the first n characters from string s2 to s1. |
| `int strcmp(char s1[], const char s2[])` | Returns a value greater than 0, 0, or less than 0 if s1 is greater than, equal to, or less than s2 based on the numeric code of the characters. |
| `int strncmp(char s1[], const char s2[], size_t n)` | Same as strcmp, but compares up to n number of characters in s1 with those in s2. |
| `int atoi(char s[])` | Returns an int value for the string. |
| `double atof(char s[])` | Returns a double value for the string. |
| `long atol(char s[])` | Returns a long value for the string. |
| `void itoa(int value, char s[], int radix)` | Obtains an integer value to a string based on specified radix. |

Some examples on c-string functions:

```
char cstr[256];
strncpy(cstr, "hello world", 5);
cout << cstr << endl; //outputs hello

strcat(cstr, " world");
cout << cstr << endl; //outputs hello world

cout << strlen(cstr) << endl; //outputs 11

itoa(3049, cstr, 10);
cout << strlen(cstr) << endl; //outputs 4

//outputs -1
cout << strcmp("abcd", "abcz") << endl;

//outputs 0
cout << strcmp("abcd", "abcd") << endl;

//outputs 1
cout << strcmp("abcz", "abcd") << endl;
```