# Array

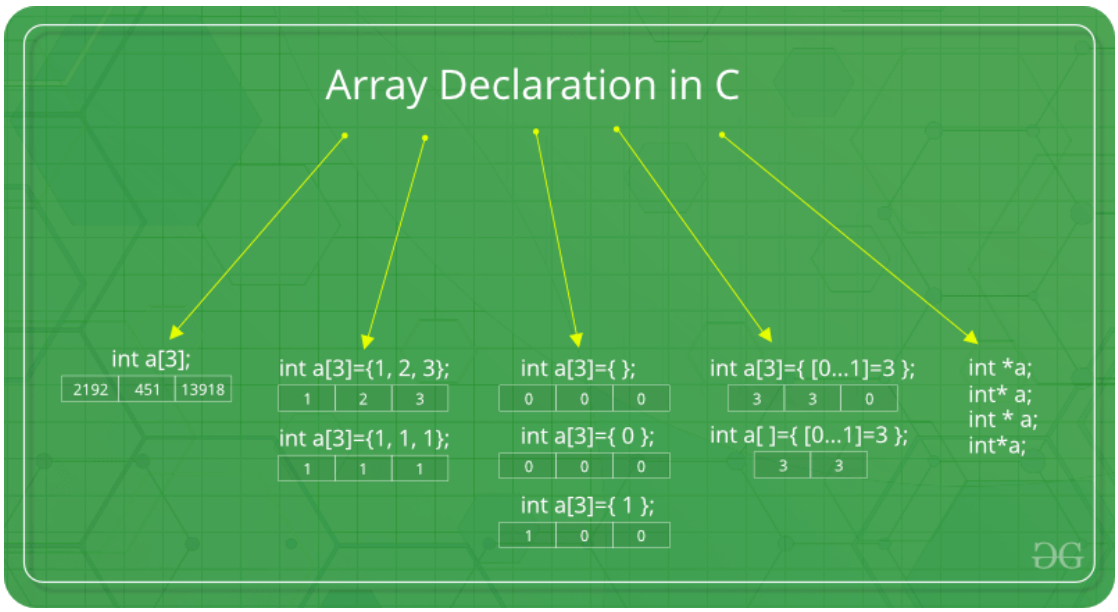
## What is an Array in C?

- An array in C is a collection of data items of similar data type.
- One or more values same data type, which may be primary data types (int, float, char), or user-defined types such as **struct** or **pointers** can be stored in an array.
- In C, the type of elements in the array should match with the data type of the array itself.
- The size of the array, also called the length of the array, must be specified in **the declaration** itself.
- Once declared, the size of a C array cannot be changed.
- When an array is declared, the compiler allocates a continuous block of memory required to store the declared number of elements.

## the main properties of arrays:

1) Collection of Same Data Type
2) Contiguous Memory Allocation
3) Fixed Size
4) Length Depends on Type
5) Indexing
6) Pointer Relationship
7) Lower and Upper Bounds
8) Multi-dimensional Array
9) Implementation of Complex Data Structures

## Declaration




Or you can initialize the array with fewer elements, and the remaining elements will be initialized with 0:

```
int arr[5] = {1, 2};
```

## INITIALIZING THE ARRAY:

- **There are two ways to initialize an array in C:**

1. Static Initialization: In static initialization, values are assigned to the array at the time of declaration.

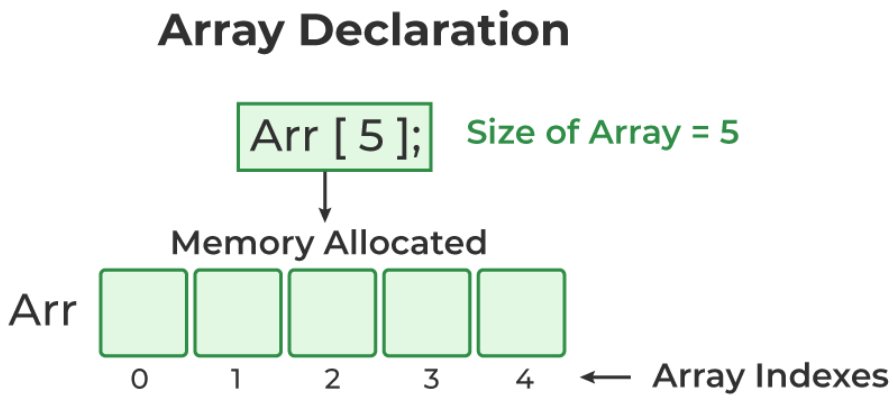
```
int arr[5] = {1, 2, 3, 4, 5};
```

- In this example, an array of size 5 is declared and initialized with values 1, 2, 3, 4, 5.

2. Dynamic Initialization: In dynamic initialization, values are assigned to the array after declaration.

```
int arr[5];
arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
arr[3] = 4;
arr[4] = 5;
```

- an array of size 5 is declared and then values are assigned to each element of the array one by one




## Collection of Same Data Type

All elements of an array must be of the same data type. This ensures consistent access and operations on the data.

If an array is declared as follows –

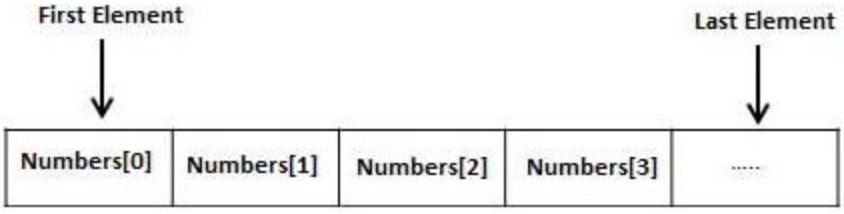
```
int arr[] = {50, 67.55, "hello", 21};
```

Compiler issues a warning –

```
initialization of 'int' from 'char *' makes integer from pointer without a cast
[-Wint-conversion]|
```

## Contiguous Memory Allocation

- All elements of an array are stored in contiguous memory locations, meaning they occupy a block of memory next to each other.
- This allows for efficient random access and memory management.



## Fixed Size

- The size of an array is fixed at the time of declaration and cannot be changed during the program's execution.
- This means you need to know the maximum number of elements you need beforehand.
- In C, an array cannot have a size defined in terms of a variable.

//This is accepted

```
#define SIZE = 10
int arr[SIZE];
```

//This is also accepted

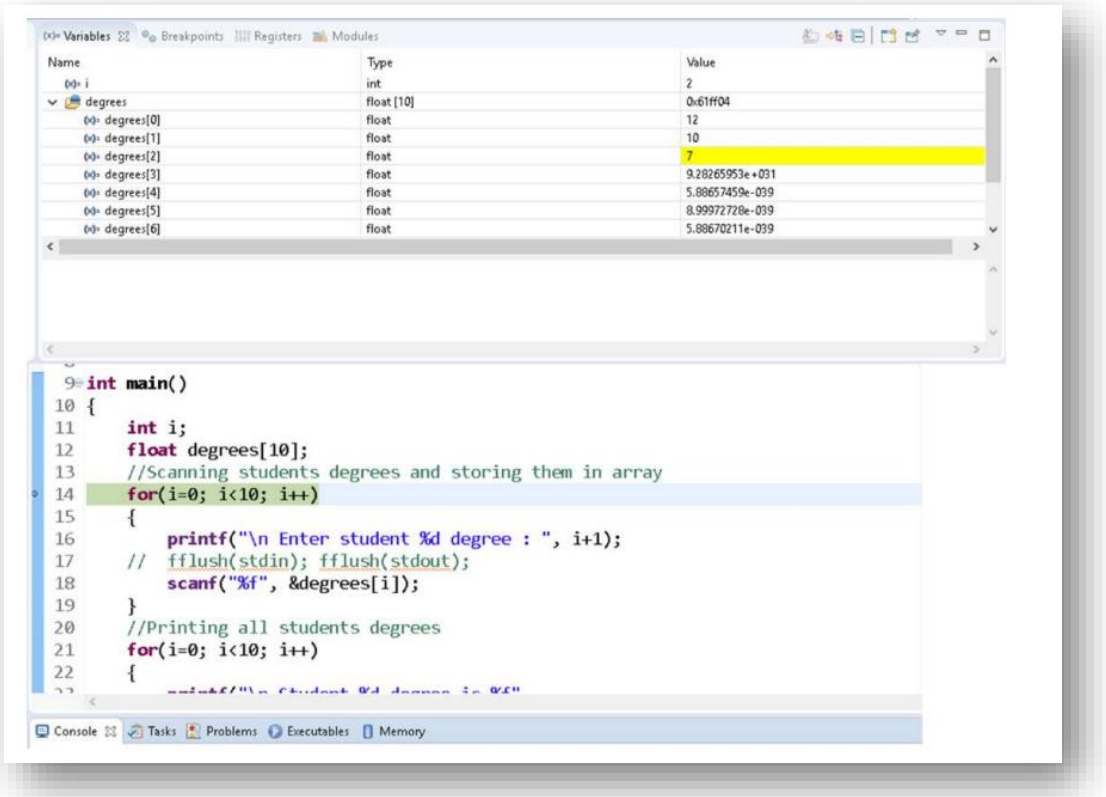
```
const SIZE = 10;
int arr[SIZE];
```

//This is not accepted

```
int SIZE = 10;
int arr[SIZE];
```

//The size must be an integer. This will give error

```
float num[10.5] = {50, 55, 67, 73, 45, 21, 39, 70, 49, 51};
```



## Length Depends on Type

Since an array can store all the elements of same type, the total memory occupied by it depends on the data type.

```c
#include<stdio.h>
int main() {
   int num[10] = {50, 55, 67, 73, 45, 21, 39, 70, 49, 51};
   int size = sizeof(num) / sizeof(int);
   printf("element at lower bound num[0]: %d \n", num[0]);
   printf("at upper bound: %d byte \n", num[size-1]);
   printf("length of int array: %ld \n", sizeof(num));
   double nm[10] = {50, 55, 67, 73, 45, 21, 39, 70, 49, 51};
   size = sizeof(nm) / sizeof(double);
   printf("element at lower bound nm[0]: %f \n", nm[0]);
   printf("element at upper bound: %f \n", nm[size-1]);
   printf("byte length of double array: %ld \n", sizeof(nm));

   return 0;
}
```

Output

```
element at lower bound num[0]: 50
at upper bound: 51 byte
length of int array: 40
element at lower bound nm[0]: 50.000000
element at upper bound: 51.000000
byte length of double array: 80
```

## Indexing

- Each element in an array has a unique index, starting from 0.
- You can access individual elements using their index within square brackets.
- Usually, array is traversed with a for loop running over its length and using the loop variable as the index.

Example

```c
#include <stdio.h>

int  main() {
   int a[] = {1,2,3,4,5};
   int i;

   for (i=0; i<4; i++){
      printf("a[%d]: %d \n", i, a[i]);
   }
   return 0;
}
```

## Pointer Relationship

- The name of an array is equivalent to a constant pointer to its first element.
- This lets you use array names and pointers interchangeably in certain contexts.

```c
#include <stdio.h>
int  main() {
   int num[10] = {50, 55, 67, 73, 45, 21, 39, 70, 49, 51};
   printf("num[0]: %d Address of 0th element: %d\n", num[0], &num[0]);
   printf("Address of array: %d", num);
   return 0;
}
```

Output

```
num[0]: 50 Address of 0th element: 6422000
Address of array: 6422000
```

## Lower and Upper Bounds

- Each element in an array is identified by an index starting with 0.
- The lower bound of and array is the index of its first element, which is always 0. The last element in the array size -1 as its index.

```c
#include <stdio.h>
int  main() {
   int num[10] = {50, 55, 67, 73, 45, 21, 39, 70, 49, 51};
   int size = sizeof(num) / sizeof(int);
   printf("element at lower bound num[0]: %d at upper bound: %d Size of array: %d",
   num[0], num[size-1], size);
   return 0;
}
```

Output

```
element at lower bound num[0]: 50 at upper bound: 51 Size of array: 10
```

# ❖ Multidimensional Arrays in C

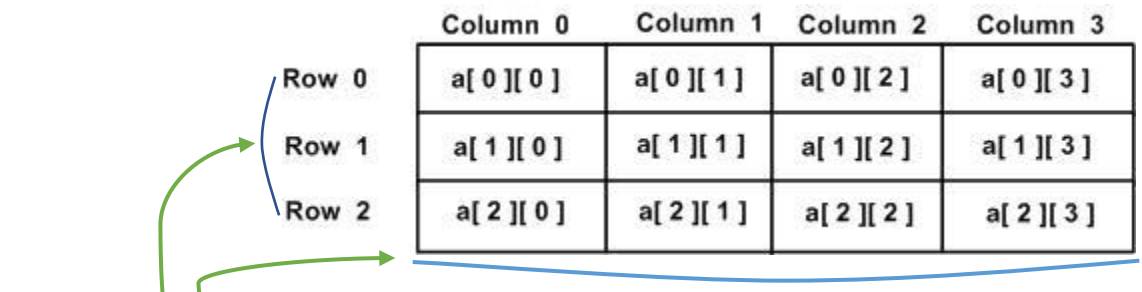
Multi-dimensional arrays can be termed as nested arrays.

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional integer array –

```
int threedim[3][3][3];
```

## Two-dimensional Array in C




```
int arr[3][5] = {1,2,3,4,5, 10,20,30,40,50, 5,10,15,20,25};
```

- The arr array has three rows and five columns.
- In C, a two-dimensional array is a row-major array.

- The first square bracket always represents the dimension size of rows, and the second is the number of columns.

✓ Obviously, the array has 3 X 5 = 15 elements.
✓ Elements are read into the array in a row-wise manner, which means the first 5 elements are stored in first row, and so on.

- the first dimension is optional in the array declaration.

```
int arr[ ][5] = {1,2,3,4,5, 10,20,30,40,50, 5,10,15,20,25};
```

The numbers are logically arranged in a tabular manner as follows –

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 10 | 20 | 30 | 40 | 50 |
| 5 | 10 | 15 | 20 | 25 |

The cell with row index 1 and column index 2 has 30 in it.

## Example of Printing Elements of Two-dimensional Array

```
#include <stdio.h>
int main () {

   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   int i, j;

   /* output each array element's value */
   for ( i = 0; i < 5; i++ ) {
      for ( j = 0; j < 2; j++ ) {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
      }
   }

   return 0;
}
```

Output

```
a[0][0] = 0
a[0][1] = 0
a[1][0] = 1
a[1][1] = 2
a[2][0] = 2
a[2][1] = 4
a[3][0] = 3
a[3][1] = 6
a[4][0] = 4
a[4][1] = 8
```

In case of a two or multi-dimensional array, the compiler assigns a memory block of the size which is the product of dimensions multiplied by the size of the data type. In this case, the size is 3 X 5 X 4 = 60 bytes, 4 being the size of int data type.

## Three-Dimensional Array In C

```
Students[hall][row][column]
```

### Example of Three-dimensional Array

```c
#include<stdio.h>

int main(){
  int i, j, k;
  int arr[3][3][3]= {
    {
      {11, 12, 13},
      {14, 15, 16},
      {17, 18, 19}
    },
    {
      {21, 22, 23},
      {24, 25, 26},
      {27, 28, 29}
    },
    {
      {31, 32, 33},
      {34, 35, 36},
      {37, 38, 39}
    },
  };

  printf("Printing 3D Array Elements\n");

  for(i=0;i<3;i++) {
    for(j=0;j<3;j++){
      for(k=0;k<3;k++){
        printf("%4d",arr[i][j][k]);
      }
      printf("\n");
    }
    printf("\n");
  }
  return 0;
}
```

Output

```
Printing 3D Array Elements
  11  12  13
  14  15  16
  17  18  19

  21  22  23
  24  25  26
  27  28  29

  31  32  33
  34  35  36
  37  38  39
```

# Strings in C

A **string in C** is a one-dimensional array of char type, with the last character in the array being a "null character" represented by '\0'or 0.

Thus, a string in C can be defined as a null-terminated sequence of char type values.

```
char string[] = {'H', 'e', 'l', 'l', 'o', '\0'};
Or
char string = "Hello";
```

## Example

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

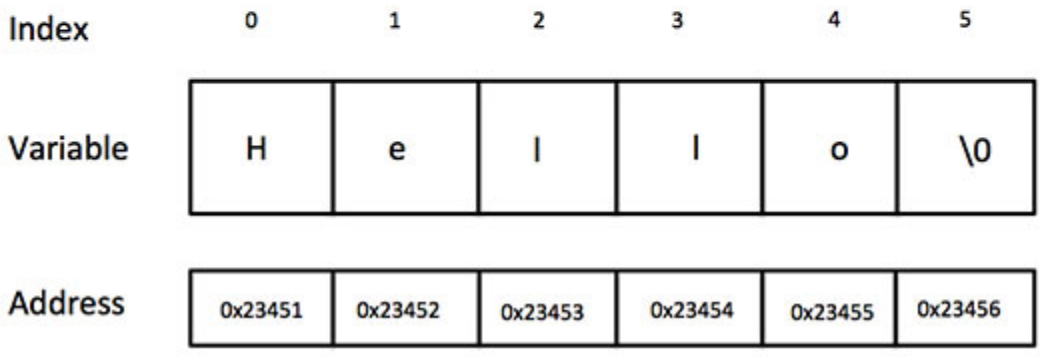
### Initializing String Without Specifying Size

C lets you initialize an array without declaring the size, in which case the compiler automatically determines the array size.

## Example

```
char greeting[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The array created in the memory can be schematically shown as follows –




- If the string is not terminated by "\0", it results in unpredictable behavior.

## Printing a String (Using %s Format Specifier)

C provides a format specifier "**%s**" which is used to print a string when you're using functions like **printf()** or **fprintf()** functions.

The "%s" specifier tells the function to iterate through the array, until it encounters the null terminator (\0) and printing each character. This effectively prints the entire string represented by the character array without having to use a loop.

```c
#include <stdio.h>

int main (){

   char greeting[] = {'H', 'e', 'l', 'l', 'o', '\0'};
   printf("Greeting message: %s\n", greeting );

   return 0;
}
```

**Output**

It will produce the following output –

```
Greeting message: Hello
```

## String Input Using scanf()

- String Input Using gets() and fgets() Functions
- To accept a string input with whitespaces in between, we should use the gets() function.
- It is called an unformatted console input function, defined in the "stdio.h" header file.

### Example: String Input Using gets() Function

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

int main(){

   char name[20];

   printf("Enter a name:\n");
   gets(name);

   printf("You entered: \n");
   printf("%s", name);

   return 0;
}
```

**Output**

Run the code and check its output –

```
Enter a name:
Sachin Tendulkar

You entered:
Sachin Tendulkar
```

## Example: String Input Using fgets () Function

The **fgets() function** can be used to accept input from any input stream, such as stdin (keyboard) or FILE (file stream).

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

int main(){

   char name[20];

   printf("Enter a name:\n");
   fgets(name, sizeof(name), stdin);

   printf("You entered: \n");
   printf("%s", name);

   return 0;
}
```

Output

Run the code and check its output –

```
Enter a name:
Virat Kohli

You entered:
Virat Kohli
```

## Example: String Input Using scanf("%[^\n]s")

You may also use **scanf("%[^\n]s")** as an alternative. It reads the characters until a newline character ("\n") is encountered.

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

int main (){

   char name[20];

   printf("Enter a name: \n");
   scanf("%[^\n]s", name);

   printf("You entered \n");
   printf("%s", name);

   return 0;
}
```

Output

Run the code and check its output –

```
Enter a name:
Zaheer Khan

You entered
Zaheer Khan
```

## gets()

- **وصف :تقرأ سلسلة نصية كاملة من الإدخال القياسي حتى تصل إلى سطر جديد** (newline) **أو نهاية الملف** (EOF).
- **الأمان :غير آمنة لأنها لا تتحقق من حجم المخزن المؤقت، مما يؤدي إلى احتمال تجاوز سعة المخزن المؤقت** (buffer overflow).

## fgets()

- **وصف :تقرأ سلسلة نصية من التدفق المحدد (مثل ملف أو الإدخال القياسي) حتى تصل إلى سطر جديد أو نهاية الملف أو حتى يتم قراءة عدد معين من الأحرف.**
- **الأمان :أكثر أمانًا لأنها تتيح تحديد الحد الأقصى لعدد الأحرف التي سيتم قراءتها، مما يمنع تجاوز سعة المخزن المؤقت.**
- **الاستخدام;**(fgets(buffer, size, stdin :

## scanf()

- **وصف :تقرأ بيانات من الإدخال القياسي بناءً على تنسيقات محددة .**(format specifiers) **يمكنها قراءة أنواع مختلفة من البيانات مثل الأعداد الصحيحة والنصوص والأحرف.**
- **الأمان :يمكن أن تكون غير آمنة إذا لم يتم التعامل مع الإدخال بشكل صحيح، حيث لا تتحقق من حجم المخزن المؤقت تلقائيًا عند قراءة السلاسل النصية.**
- **الاستخدام** scanf("%s", buffer); **:لقراءة سلسلة نصية، أو** scanf("%d", &number); **لقراءة عدد صحيح.**

```c
int main() {
    char str[100];

    printf("Enter a string: ");
    gets(str);  // يستخدم لقراءة السلسلة النصية من المستخدم

    printf("You entered: %s\n", str);

    return 0;
}
```

```c
#include <stdio.h>

int main() {
    char str[100];

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);  // يستخدم لقراءة السلسلة النصية من المستخدم مع تحديد الحد الأقصى للطول

    // إزالة \n إذا كان موجوداً
    str[strcspn(str, "\n")] = '\0';

    printf("You entered: %s\n", str);

    return 0;
}
```

## Declare and Initialize an Array of Strings

### Syntax

To construct an array of strings, the following syntax is used –

```
char strings [no_of_strings] [max_size_of_each_string];
```

```
char langs [10][15] = {
   "PYTHON", "JAVASCRIPT", "PHP",
   "NODE JS", "HTML", "KOTLIN", "C++",
   "REACT JS", "RUST", "VBSCRIPT"
};
```

## Printing An Array of Strings

```
#include <stdio.h>

int main (){

   char langs [10][15] = {
      "PYTHON", "JAVASCRIPT", "PHP",
      "NODE JS", "HTML", "KOTLIN", "C++",
      "REACT JS", "RUST", "VBSCRIPT"
   };

   for (int i = 0; i < 10; i++){
      printf("%s\n", langs[i]);
   }

   return 0;
}
```

### Output

When you run this code, it will produce the following output –

```
PYTHON
JAVASCRIPT
PHP
NODE JS
HTML
KOTLIN
C++
REACT JS
RUST
VBSCRIPT
```

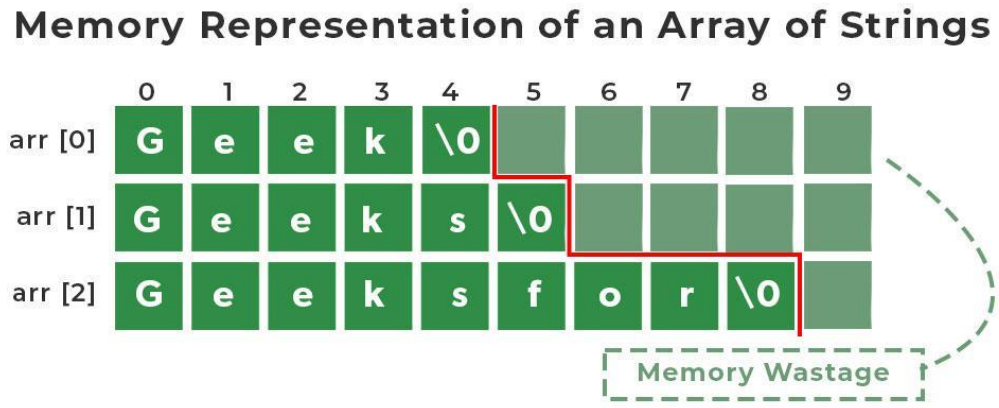
## How an Array of Strings is Stored in Memory?

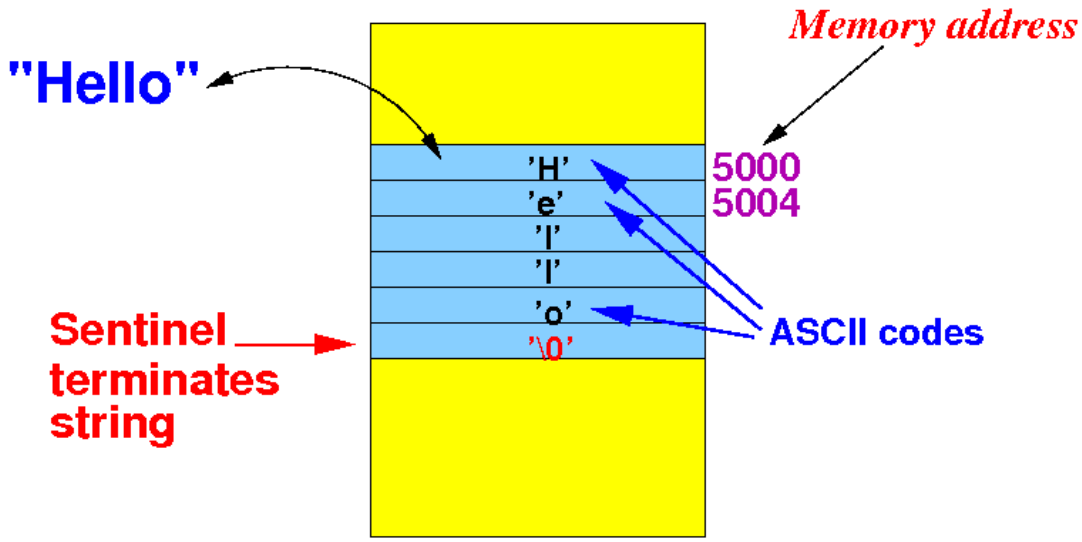
We know that each char type occupies 1 byte in the memory.

Hence, this array will be allocated a block of 150 bytes. Although this block is contagious memory locations, each group of 15 bytes constitutes a row.

Assuming that the array is located at the memory address 1000, the logical layout of this array can be shown as in the following figure –

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000 | 0 | P | Y | T | H | O | N | \0 | | | | | | | | |
| 1015 | 1 | J | A | V | A | S | C | R | I | P | T | \0 | | | | |
| 1030 | 2 | P | H | P | \0 | | | | | | | | | | | |
| 1045 | 3 | N | O | D | E | | J | S | \0 | | | | | | | |
| 1060 | 4 | H | T | M | L | \0 | | | | | | | | | | |
| 1075 | 5 | K | O | T | L | I | N | \0 | | | | | | | | |
| 1090 | 6 | C | + | + | \0 | | | | | | | | | | | |
| 1105 | 7 | R | E | A | C | T | | J | S | \0 | | | | | | |
| 1120 | 8 | R | U | S | T | \0 | | | | | | | | | | |
| 1135 | 9 | V | B | S | C | R | I | P | T | \0 | | | | | | |






```c
int main(int argc, char *argv[])
{
   char a[10] = { 'H', 'e', 'l', 'l', 'o', '\0' } ; // String variable
                // Don't forget the sentinel '\0'
                // that ends the string !

   printf( "sizeof(a) = %d\n", sizeof(a) );
   printf( "String a = %s\n", &a[0] );      // Print string starting at a[0]
   printf( "Strange string: %s\n", &a[3] ); // Print string starting at a[3]
}
```

**Output:**

```
sizeof(a) = 10
String a = Hello
Strange string: lo
```

**Explanation:**

```
The string in array a[] is as follows:

      a[0]        a[3]     a[5]
      +---+---+---+---+---+---+---+---+---+---+
      | H | e | l | l | o | \0| ? | ? | ? | ? |
      +---+---+---+---+---+---+---+---+---+---+

-------------------------------------------------

The string starting at location a[0] is:

      a[0]        a[3]
      +---+---+---+---+---+---+---+---+---+---+
      | H | e | l | l | o | \0| ? | ? | ? | ? |
      +---+---+---+---+---+---+---+---+---+---+

So the program prints Hello

-------------------------------------------------

The string starting at location a[3] is:

      a[0]        a[3]
      +---+---+---+---+---+---+---+---+---+---+
      | H | e | l | l | o | \0| ? | ? | ? | ? |
      +---+---+---+---+---+---+---+---+---+---+

So the program prints lo
```

**Note:**

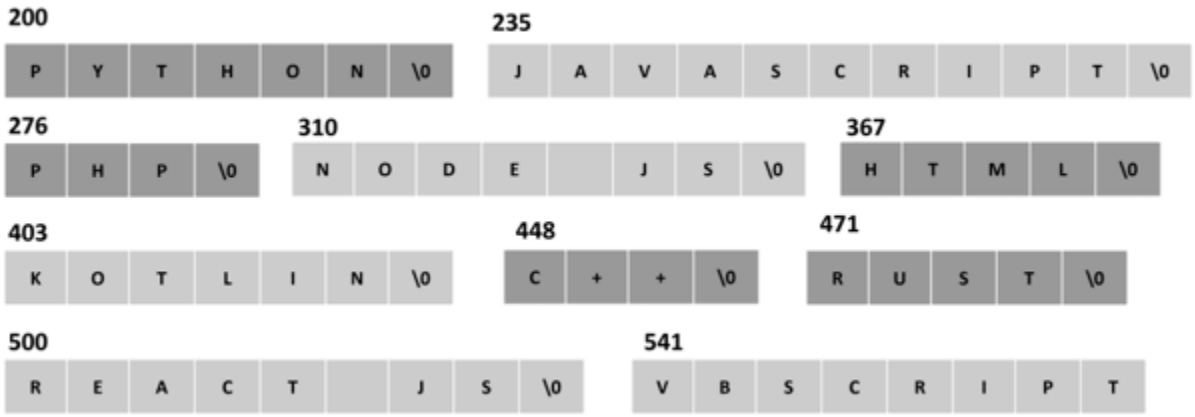
- Because the **sentinal** '\0' marks the **end of a string**, the data inside `a[6]`, `a[7]`, `a[8]` and `a[9]` are **ignored** by the `printf()` function.

## An Array of Strings with Pointers

To use the memory more efficiently, we can use the pointers. Instead of a 2D char array, we declare a 1D array of "char *" type.

```
char *langs[10] = {
   "PYTHON", "JAVASCRIPT", "PHP",
   "NODE JS", "HTML", "KOTLIN", "C++",
   "REACT JS", "RUST", "VBSCRIPT"
};
```

In the 2D array of characters, the strings occupied 150 bytes. As against this, in an array of pointers, the strings occupy far less number of bytes, as each string is randomly allocated memory as shown below –




**Note:** Here, **lang[ ]** is an array of pointers of individual strings.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 200 | 235 | 276 | 310 | 367 | 403 | 448 | 471 | 500 | 541 |
| 5000 | 5004 | 5008 | 5012 | 5016 | 5020 | 5024 | 5028 | 5032 | 5036 |

char *langs[]

# C String Functions

| No. | Function | Description |
|---|---|---|
| 1) | strlen(string_name) | returns the length of string name. |
| 2) | strcpy(destination, source) | copies the contents of source string to destination string. |
| 3) | strcat(first_string, second_string) | concats or joins first string with second string. The result of the string is stored in first string. |
| 4) | strcmp(first_string, second_string) | compares the first string with second string. If both strings are same, it returns 0. |
| 5) | strrev(string) | returns reverse string. |
| 6) | strlwr(string) | returns string characters in lowercase. |
| 7) | strupr(string) | returns string characters in uppercase. |

## 3. strcmp(first_string, second_string)

**خصائص:**

- **وظيفة :يقارن بين سلسلتين نصيتين حرف بحرف.**
- **الإدخال :سلسلتان نصيتان.**
- **الإخراج :عدد صحيح (صفر إذا كانتا متطابقتين، موجب إذا كانت الأولى أكبر، وسالب إذا كانت الثانية أكبر).**
- **مزايا :مفيد للمقارنة والفرز.**
- **قيود :حساس لحالة الأحرف.**(case-sensitive)

**int strcmp(const char *str1, const char *str2)**

It takes two strings as parameters: str1 and str2. It returns an integer value indicating the comparison result.

**Here are the possible return values and their meanings:**

0: Both strings are identical

Positive integer: The first non-matching character in str1 has a greater ASCII value than the corresponding character in str2.

Negative integer: The first non-matching character in str1 has a smaller ASCII value than the corresponding character in str2.

### باستخدام الدالة الجاهزة:

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

int main() {
    char str1[] = "Hello";
    char str2[] = "World";
    int result = strcmp(str1, str2);
    printf("Comparison result: %d\n", result);
    return 0;
}
```

### تصميم الدالة يدويًا:

```c
#include <stdio.h>

int my_strcmp(const char *str1, const char *str2) {
    while (*str1 && (*str1 == *str2)) {
        str1++;
        str2++;
    }
    return *(unsigned char *)str1 - *(unsigned char *)str2;
}

int main() {
    char str1[] = "Hello";
    char str2[] = "World";
    int result = my_strcmp(str1, str2);
    printf("Comparison result: %d\n", result);
    return 0;
}
```

## strlen(string_name)

**خصائص:**

- **وظيفة :تحسب وتعيد طول السلسلة النصية (عدد الأحرف قبل '\0').**
- **الإدخال :سلسلة نصية.**
- **الإخراج :عدد صحيح يمثل طول السلسلة.**
- **مزايا :سريع وبسيط في الاستخدام.**
- **قيود :لا يحسب المساحة المخصصة وإنما طول النص الفعلي فقط.**

### باستخدام الدالة الجاهزة:

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

int main() {
    char str[] = "Hello, World!";
    int length = strlen(str);
    printf("Length of string: %d\n", length);
    return 0;
}
```

### تصميم الدالة يدويًا:

```c
#include <stdio.h>

int my_strlen(const char *str) {
    int length = 0;
    while (str[length] != '\0') {
        length++;
    }
    return length;
}

int main() {
    char str[] = "Hello, World!";
    int length = my_strlen(str);
    printf("Length of string: %d\n", length);
    return 0;
}
```

## strcpy(destination, source)

### خصائص

- وظيفة :تنسخ محتوى السلسلة النصية المصدر إلى السلسلة النصية الوجهة.
- الإدخال :سلسلتان نصيتان، المصدر والوجهة.
- الإخراج :مؤشر إلى السلسلة النصية الوجهة.
- مزايا :سهل الاستخدام للنسخ الكامل للسلاسل النصية.
- قيود :يجب أن تكون السلسلة الوجهة كبيرة بما يكفي لاستيعاب السلسلة المصدر.

### باستخدام الدالة الجاهزة:

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

int main() {
    char source[] = "Hello, World!";
    char destination[20];
    strcpy(destination, source);
    printf("Copied string: %s\n", destination);
    return 0;
}
```

### تصميم الدالة يدويًا:

```c
#include <stdio.h>

char* my_strcpy(char *dest, const char *src) {
    int i = 0;
    while (src[i] != '\0') {
        dest[i] = src[i];
        i++;
    }
    dest[i] = '\0';
    return dest;
}

int main() {
    char source[] = "Hello, World!";
    char destination[20];
    my_strcpy(destination, source);
    printf("Copied string: %s\n", destination);
    return 0;
}
```

## strcat(first_string, second_string)

خصائص:

- وظيفة :يدمج (يضيف) السلسلة النصية الثانية إلى نهاية السلسلة النصية الأولى.
- الإدخال :سلسلتان نصيتان، الأولى والثانية.
- الإخراج :مؤشر إلى السلسلة النصية الأولى بعد الدمج.
- مزايا :يسمح بدمج سلاسل نصية بسهولة.
- قيود :يجب أن تكون السلسلة الأولى كبيرة بما يكفي لاستيعاب محتوى السلسلة الثانية.

### باستخدام الدالة الجاهزة:

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

int main() {
    char str1[20] = "Hello, ";
    char str2[] = "World!";
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1);
    return 0;
}
```

### تصميم الدالة يدويًا:

```c
#include <stdio.h>

char* my_strcat(char *dest, const char *src) {
    int i = 0, j = 0;
    while (dest[i] != '\0') {
        i++;
    }
    while (src[j] != '\0') {
        dest[i + j] = src[j];
        j++;
    }
    dest[i + j] = '\0';
    return dest;
}

int main() {
    char str1[20] = "Hello, ";
    char str2[] = "World!";
    my_strcat(str1, str2);
    printf("Concatenated string: %s\n", str1);
    return 0;
}
```

## strrev(string)

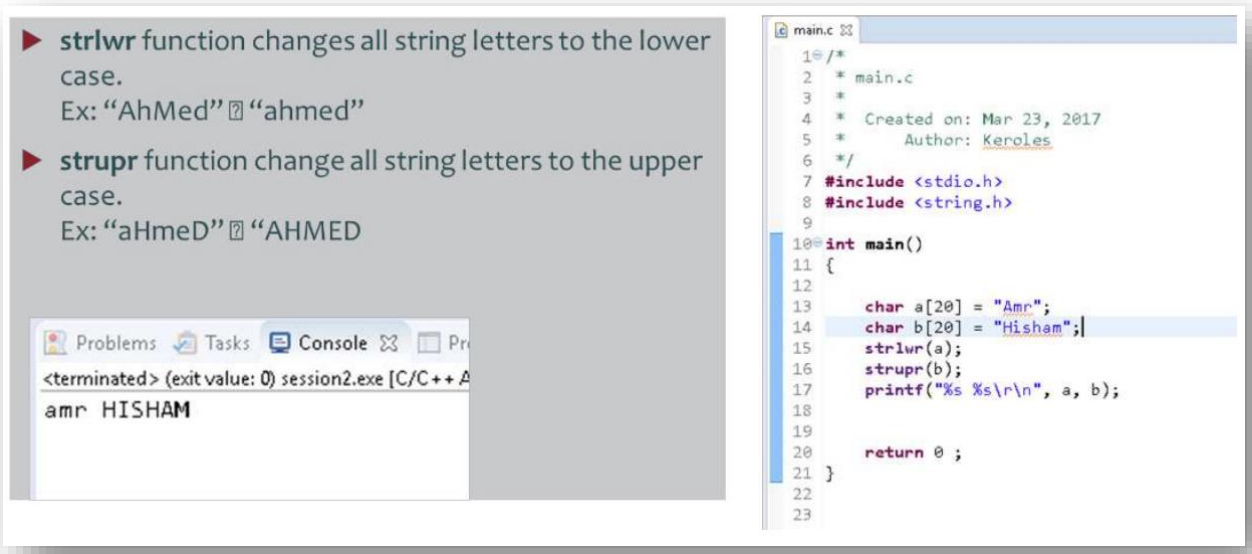
باستخدام الدالة الجاهزة:

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

char* strrev(char *str) {
    int i, j;
    char temp;
    int len = strlen(str);
    for (i = 0, j = len - 1; i < j; i++, j--) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
    return str;
}

int main() {
    char str[] = "Hello, World!";
    printf("Reversed string: %s\n", strrev(str));
    return 0;
}
```

## strlwr(string) & strupr(string)



To change the case of a string in C, you can use the ctype.h library which provides two functions:

- tolower()
- toupper()

These functions convert a character to its lower- or upper-case equivalent, respectively.

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

void string_to_upper(char str[]) {
    int i = 0;
    while (str[i]) {
        str[i] = toupper(str[i]);
        i++;
    }
}

int main() {
    char str[] = "hello world";
    printf("Original string: %s\n", str);
    string_to_upper(str);
    printf("Uppercase string: %s\n", str);
    return 0;
}
```

## Converting String To Integer Value

In C, you can convert a string to **an integer value** using the atoi() function. Here is an example:

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

int main() {
    char str[] = "12345";
    int num = atoi(str);
    printf("The integer value is: %d\n", num);
    return 0;
}
```

**The atoi()** function takes a string as an argument and returns its integer equivalent.

If the string is not a valid integer, the function **returns 0.**

**In this example,**

- ✓ the string "12345" is converted to the integer value 12345 using **atoi().**
- ✓ The result is stored in the variable num, which is then printed to the console using printf().

## Converting String To Real Value

To convert a string to **a real value** in C, you can use the atof() function from the stdlib.h library. Here's an example:

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

int main() {
    char str[] = "3.14159";
    double real_value = atof(str);
    printf("The real value is: %f\n", real_value);
    return 0;
}
```

In this example,

the **str** variable contains the string "3.14159", and we use atof() to convert it to a double value.

The resulting value is stored in the real_value variable, which is then printed to the console using printf().

**The output of this program will be:**

```
The real value is: 3.141590
```


" من ضيع الأصول حرم الوصول ومن ترك الدليل ضل السبيل"