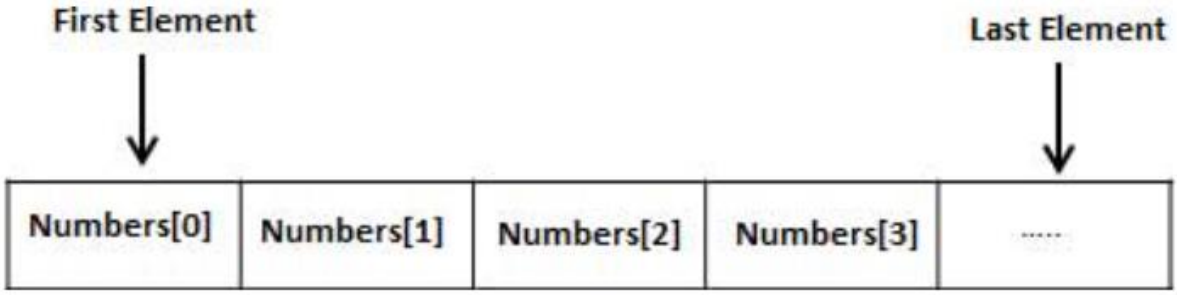
UNIT-2

Arrays

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

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. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

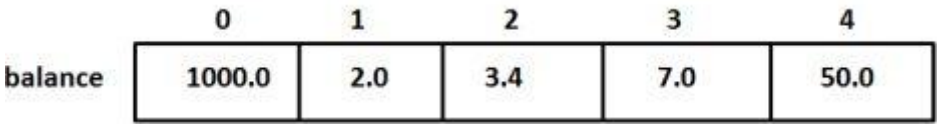
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above −



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays −

```
#include<stdio.h>

int main (){

int n[10];/* n is an array of 10 integers */
int i,j;

/* initialize elements of array n to 0 */
for( i =0; i <10; i++){
    n[ i ]= i +100;/* set element at location i to i + 100 */
}

/* output each array element's value */
for(j =0; j <10; j++){
    printf("Element[%d] = %d\n", j, n[j]);
}

return0;
}
```

Two-dimensional Arrays:

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows −

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4]={
{0,1,2,3},/*  initializers for row indexed by 0 */
{4,5,6,7},/*  initializers for row indexed by 1 */
{8,9,10,11}/*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a 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]);
}
}

return0;
}
```

Comparison Between C Arrays with List and Tuple in Python

List: A list is of an ordered collection data type that is mutable which means it can be easily modified and we can change its data values and a list can be indexed, sliced, and changed and each element can be accessed using its index value in the list. The following are the main characteristics of a List:

- The list is an ordered collection of data types.
- The list is mutable.
- List are dynamic and can contain objects of different data types.
- List elements can be accessed by index number.

Declaring a List:

```
# empty list
my_list = []

# list of integers
my_list = [1, 2, 3]

# list with mixed data types
my_list = [1, "Hello", 3.4]
```

Eg: list = ["mango", "strawberry", "orange",

"apple", "banana"]
print(list)

Tuple: A tuple is an ordered and an immutable data type which means we cannot change its values and tuples are written in round brackets. We can access tuple by referring to the index number inside the square brackets. The following are the main characteristics of a Tuple:

- Tuples are immutable and can store any type of data type.
- it is defined using ().
- it cannot be changed or replaced as it is an immutable data type.

Declaring a Tuple:

```
# Empty tuple
my_tuple = ()
print(my_tuple)

# Tuple having integers
my_tuple = (1, 2, 3)


# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
```

Eg: tuple = ("orange","apple","banana")

print(tuple)

Character Input/output Functions:

Input means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

In this tutorial, we will learn about such functions, which can be used in our program to take input from user and to output the result on screen.

All these built-in functions are present in C header files, we will also specify the name of header files in which a particular function is defined while discussing about it.

Formatted Functions		
Type	Input	Output
char	scanf()	printf()
int	scanf()	printf()
float	scanf()	printf()
string	scanf()	printf()

Unformatted Functions		
Type	Input	Output
char	getch() getche() getchar()	putch() putchar()
int	-	-
float	-	-
string	gets()	puts()

Non-formatted input and output can be carried out by standard input-output library functions in C. These can handle one character at a time. For the input functions

it does not require <Enter> to be pressed after the entry of the character. For output functions, it prints a single character on the console.

The Character Input/Output functions are:

getch():This input function reads, without echoing on the screen, a single character from the keyboard and immediately returns that character to the program. General statement form:

ch = getch(); /* 'ch' is a character variable */

Example code:

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

int main() {
   char ch = getch();
   printf("Received Input: %c\n", ch);
   return 0;
}
```

Example 2:

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

int main() {
   // Set op = {0, 0, 0, 0, 0, 0} = '\0\0\0\0\0\0' string
   char op[6] = {0};
   for (int i=0; i<5; i++) {
      op[i] = getch();
   }
   printf("Received 5 character Input: %s\n", op);
   return 0;
}
```

getche(): This input function reads, with echo on the screen, a single character from the keyboard and immediately returns that character to the program. General statement form:

ch = getche(); /* 'ch' is a character variable */

Example:

```
#include <stdio.h>
#include <conio.h>
// Example for getche() in C
int main()
{
printf("%c", getche());
return 0;
}
```

getchar(): Reads a *single* character from the user at the console, *and echoing it*, but needs an **Enter** key to be pressed at the end.

Syntax: `int getchar(void);`

`Example:`

```
// Example for getchar() in C
#include <stdio.h>
int main()
{
printf("%c", getchar());
return 0;
}
```

putchar(): The putchar() function displays the character passed to it on the screen and returns the same character. This function too displays only a single character at a time.

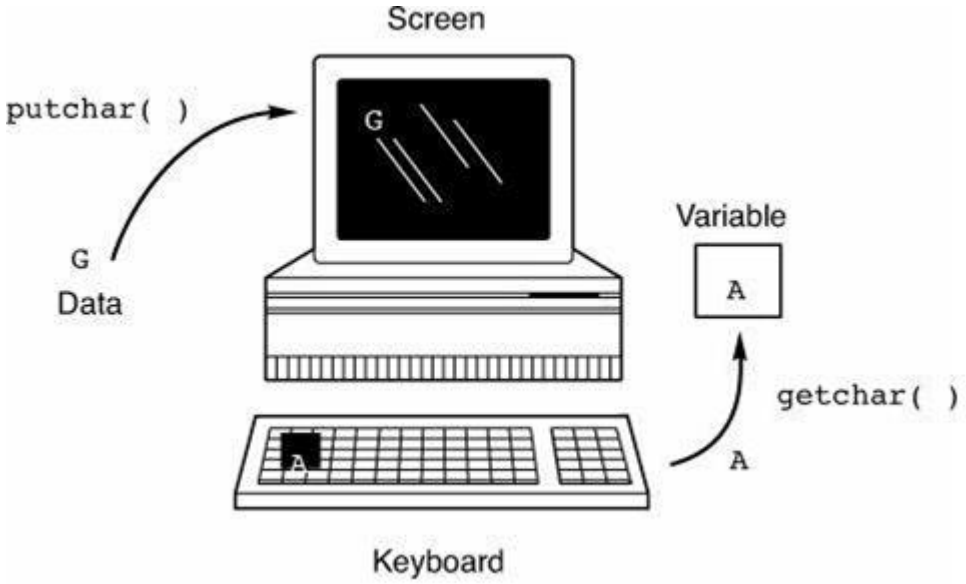
Syntax: putchar(variable_name);

Example:

```
#include <stdio.h>

void main()
{
   int c;
printf("Enter a character");
/*
      Take a character as input and
      store it in variable c
   */
   c =getchar();
/*
      display the character stored
      in variable c
   */
putchar(c);
}
```

Working of getchar() and putchar():



putch(): This output function writes the character directly to the screen. On success, the function putch() returns the character printed. On error, it returns EOF. General statement form:

putch(ch); /* 'ch' is a character variable */

Example code:

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

void main()
```

```
{
  char ch;
  clrscr(); /* Clear the screen */
  printf("Press any character: ");
  ch = getch();
  printf("\nPressed character is:");
  putch(ch);
  getch(); /* Holding output */
}
```

gets(): Reads a *single* string entered by the user at the console.

Syntax: char ***gets** (char *string)

Example:

```
#include<stdio.h>
void main ()
{
   char s[30];
   printf("Enter the string? ");
   gets(s);
   printf("You entered %s",s);
}
```

Puts():The puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console.

Syntax: int puts(char[])

Example code:

```
#include<stdio.h>
int main(){
char a[]="Hello";
puts(a);
return 0;
}
```

getc():getc functions is used to read a character from a file. In a C program, we read a character as below.

Declaration: int getc(FILE *stream)

putc(): putc(): putc function is used to display a character on standard output or is used to write into a file. In a C program, we can use putc as below.

Declaration: int putc (int ch, FILE *file)

EXAMPLE of getc() and putc()

```
#include<stdio.h>

int main (){
char c;

   printf("Enter character: ");
   c = getc(stdin);
   printf("Character entered: ");
   putc(c, stdout);

return(0);
}
```