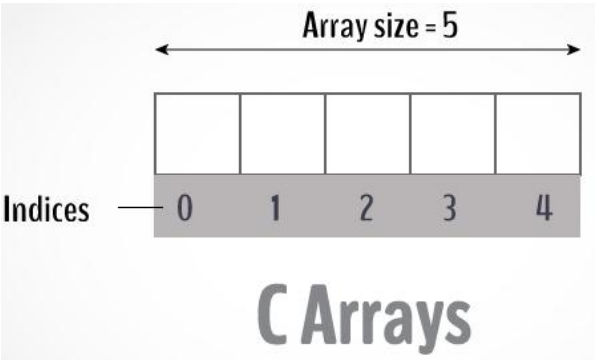
Module 4

Arrays



An array is a collection of data that holds fixed number of values of same type.

For example: if you want to store marks of 100 students, you can create an array for it.

float marks[100];

The size and type of arrays cannot be changed after its declaration.

Arrays are of two types:

1. One-dimensional arrays
2. Multidimensional arrays

Declare An Array

data_type array_name[array_size];

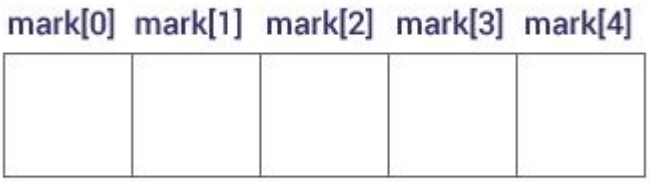
For example,

float mark[5];

Here, we declared an array, **mark**, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

Elements of an Array and to access them

You can access elements of an array by indices.
Suppose you declared an array **mark** as above. The first element is **mark[0],** second element is **mark[1]** and so on.



- Arrays have 0 as the first index not 1. In this example, **mark[0]**
- If the size of an array is *n*, to access the last element, (n-1) index is used. In this example, **mark[4]**
- Suppose the starting address of mark[0] is 2120d. Then, the next address, a[1], will be 2124d, address of a[2] will be 2128d and so on. It's because the size of a float is 4 bytes.

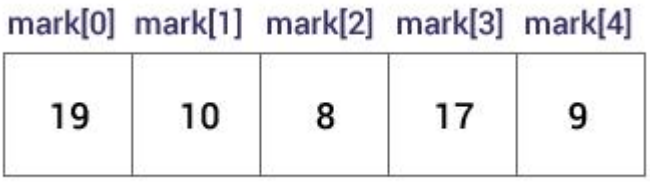
Initialize an Array
It's possible to initialize an array during declaration.

For example,

int mark[5] = {19, 10, 8, 17, 9};

Another method to initialize array during declaration:

int mark[] = {19, 10, 8, 17, 9};



Here,
mark[0] is equal to 19
mark[1] is equal to 10
mark[2] is equal to 8
mark[3] is equal to 17
mark[4] is equal to 9

Insert and Print Array Elements

```
int mark[5] = {19, 10, 8, 17, 9}

// insert different value to third element
mark[3] = 9;

// take input from the user and insert in third element
scanf("%d", &mark[2]);

// take input from the user and insert in (i+1)th element
scanf("%d", &mark[i]);

// print first element of an array
printf("%d", mark[0]);

// print ith element of an array
printf("%d", mark[i-1]);
```

Example: C Arrays

// Program to find the average of n (n < 10) numbers using arrays

```
#include <stdio.h>
int main()
{
   int marks[10], i, n, sum = 0, average;
   printf("Enter n: ");
   scanf("%d", &n);
   for(i=0; i<n; ++i)
   {
      printf("Enter number%d: ",i+1);
      scanf("%d", &marks[i]);
      sum += marks[i];
   }
   average = sum/n;

   printf("Average marks = %d", average);
```

```
    return 0;
}
```

Output

```
Enter n: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39
```

Important thing to remember when working with C arrays

Suppose you declared an array of 10 elements. Let's say,

int testArray[10];

You can use the array members from testArray[0] to testArray[9].

If you try to access array elements outside of its bound, let's say testArray[12], the compiler may not show any error. However, this may cause unexpected output (undefined behavior).

Multidimensional Arrays

In C programming, you can create an array of arrays known as multidimensional array.

For example,

float x[3][4];

Here, *x* is a two-dimensional (2D) array. The array can hold 12 elements. You can think the array as table with 3 row and each row has 4 column.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Similarly, you can declare a three-dimensional (3d) array.
For example,

float y[2][4][3];

Here, The array **y** can hold 24 elements.

You can think this example as: Each 2 elements have 4 elements, which makes 8 elements and each 8 elements can have 3 elements. Hence, the total number of elements is 24.

Initialize a Multidimensional Array

There is more than one way to initialize a multidimensional array.

Initialization of a two dimensional array
// Different ways to initialize two dimensional array

int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};

Above code are three different ways to initialize a two dimensional arrays.

Initialization of Three Dimensional Array.

You can initialize a three dimensional array in a similar way like a two dimensional array.

Here's an example,

```
int test[2][3][4] = {
                    { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} },
                    { {13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9} }
                    };
```

Example #1: Two Dimensional Array to store and display values

```
// C program to store temperature of two cities for a week and display it.
#include <stdio.h>

const int CITY = 2;
const int WEEK = 7;

int main()
{
   int temperature[CITY][WEEK];
   for (int i = 0; i < CITY; ++i) {
      for(int j = 0; j < WEEK; ++j) {
         printf("City %d, Day %d: ", i+1, j+1);
         scanf("%d", &temperature[i][j]);
      }
   }

   printf("\nDisplaying values: \n\n");
   for (int i = 0; i < CITY; ++i) {
      for(int j = 0; j < WEEK; ++j)
      {
         printf("City %d, Day %d = %d\n", i+1, j+1, temperature[i][j]);
      }
   }
   return 0;
}
```

Output

```
City 1, Day 1: 33
City 1, Day 2: 34
City 1, Day 3: 35
City 1, Day 4: 33
City 1, Day 5: 32
City 1, Day 6: 31
City 1, Day 7: 30
```

City 2, Day 1: 23
City 2, Day 2: 22
City 2, Day 3: 21
City 2, Day 4: 24
City 2, Day 5: 22
City 2, Day 6: 25
City 2, Day 7: 26

Displaying values:

City 1, Day 1 = 33
City 1, Day 2 = 34
City 1, Day 3 = 35
City 1, Day 4 = 33
City 1, Day 5 = 32
City 1, Day 6 = 31
City 1, Day 7 = 30
City 2, Day 1 = 23
City 2, Day 2 = 22
City 2, Day 3 = 21
City 2, Day 4 = 24
City 2, Day 5 = 22
City 2, Day 6 = 25
City 2, Day 7 = 26

Example #2: Sum of two matrices using Two dimensional arrays

C program to find the sum of two matrices of order 2*2 using multidimensional arrays.

```
#include <stdio.h>
int main()
{
  float a[2][2], b[2][2], c[2][2];
  int i, j;

  // Taking input using nested for loop
  printf("Enter elements of 1st matrix\n");
```

```
  for(i=0; i<2; ++i)
   for(j=0; j<2; ++j)
   {
     printf("Enter a%d%d: ", i+1, j+1);
     scanf("%f", &a[i][j]);
   }

  // Taking input using nested for loop
   printf("Enter elements of 2nd matrix\n");
   for(i=0; i<2; ++i)
    for(j=0; j<2; ++j)
    {
      printf("Enter b%d%d: ", i+1, j+1);
      scanf("%f", &b[i][j]);
    }

   // adding corresponding elements of two arrays
   for(i=0; i<2; ++i)
    for(j=0; j<2; ++j)
    {
      c[i][j] = a[i][j] + b[i][j];
    }

   // Displaying the sum
   printf("\nSum Of Matrix:");

   for(i=0; i<2; ++i)
    for(j=0; j<2; ++j)
    {
      printf("%.1f\t", c[i][j]);

      if(j==1)
        printf("\n");
    }
return 0;
}
```

Ouput

```
Enter elements of 1st matrix
Enter a11: 2;
Enter a12: 0.5;
Enter a21: -1.1;
Enter a22: 2;
Enter elements of 2nd matrix
Enter b11: 0.2;
Enter b12: 0;
Enter b21: 0.23;
Enter b22: 23;

Sum Of Matrix:
2.2     0.5
-0.9    25.0
```

Example #3: Three Dimensional Array

C Program to store values entered by the user in a three-dimensional array and display it.

```
#include <stdio.h>
int main()
{
   // this array can store 12 elements

   int i, j, k, test[2][3][2];

   printf("Enter 12 values: \n");

   for(i = 0; i < 2; ++i) {
      for (j = 0; j < 3; ++j) {
         for(k = 0; k < 2; ++k ) {
            scanf("%d", &test[i][j][k]);
         }
      }
   }

   // Displaying values with proper index.
```

```
   printf("\nDisplaying values:\n");

   for(i = 0; i < 2; ++i) {
      for (j = 0; j < 3; ++j) {
         for(k = 0; k < 2; ++k ) {
            printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
         }
      }
   }

   return 0;
}
```

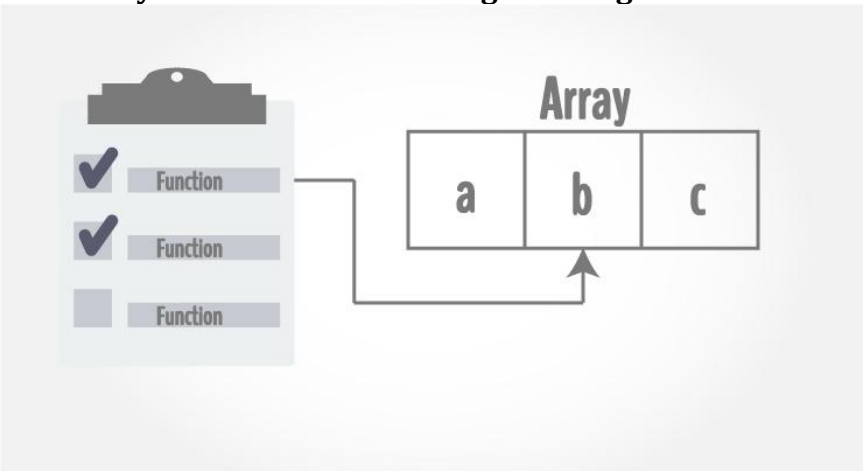
Output

```
Enter 12 values:
1
2
3
4
5
6
7
8
9
10
11
12

Displaying Values:
test[0][0][0] = 1
test[0][0][1] = 2
test[0][1][0] = 3
test[0][1][1] = 4
test[0][2][0] = 5
test[0][2][1] = 6
test[1][0][0] = 7
test[1][0][1] = 8
test[1][1][0] = 9
```

test[1][1][1] = 10
test[1][2][0] = 11
test[1][2][1] = 12

Pass arrays to a function in C Programming



In C programming, a single array element or an entire **array** can be passed to a **function**.
This can be done for both one-dimensional array or a multi-dimensional array.

Passing One-dimensional Array in Function
Single element of an array can be passed in similar manner as passing variable to a function.

C program to pass a single element of an array to function

```
#include <stdio.h>
void display(int age)
{
   printf("%d", age);
}

int main()
{
   int ageArray[] = { 2, 3, 4 };
```

```
  display(ageArray[2]); //Passing array element ageArray[2] only.
  return 0;
}
```

Output

4

Passing an entire one-dimensional array to a function

While passing arrays as arguments to the function, only the name of the array is passed (,i.e, starting address of memory area is passed as argument).

C program to pass an array containing age of person to a function. This function should find average age and display the average age in main function.

```
#include <stdio.h>
float average(float age[]);

int main()
{
  float avg, age[] = { 23.4, 55, 22.6, 3, 40.5, 18 };
  avg = average(age); /* Only name of array is passed as argument. */
  printf("Average age=%.2f", avg);
  return 0;
}

float average(float age[])
{
  int i;
  float avg, sum = 0.0;
  for (i = 0; i < 6; ++i) {
    sum += age[i];
  }
  avg = (sum / 6);
  return avg;
}
```

Output

Average age=27.08

Passing Multi-dimensional Arrays to Function

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

Example: Pass two-dimensional arrays to a function

```
#include <stdio.h>
void displayNumbers(int num[2][2]);
int main()
{
   int num[2][2], i, j;
   printf("Enter 4 numbers:\n");
   for (i = 0; i < 2; ++i)
      for (j = 0; j < 2; ++j)
         scanf("%d", &num[i][j]);
   // passing multi-dimensional array to displayNumbers function
   displayNumbers(num);
   return 0;
}

void displayNumbers(int num[2][2])
{
   // Instead of the above line,
   // void displayNumbers(int num[][2]) is also valid
   int i, j;
   printf("Displaying:\n");
   for (i = 0; i < 2; ++i)
      for (j = 0; j < 2; ++j)
         printf("%d\n", num[i][j]);
}
```

Output

Enter 4 numbers:
2
3
4
5
Displaying:
2
3
4
5

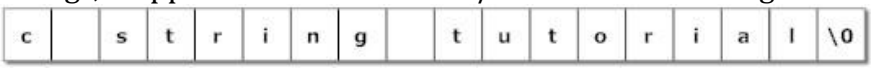
Strings

In C programming, array of characters is called a string. A string is terminated by a null character /0.

For example:

"c string tutorial"

Here, "c string tutorial" is a string. When, compiler encounter strings, it appends a null character /0 at the end of string.

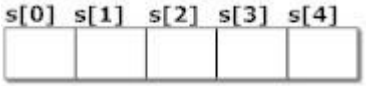


Declaration of strings

Before we actually work with strings, we need to declare them first. Strings are declared in a similar manner as **arrays**. Only difference is that, strings are of char **type**.

Using arrays

char s[5];



Using pointers

Strings can also be declared using **pointer**.

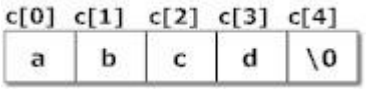
char *p;

Initialization of strings

In C, string can be initialized in a number of different ways. For convenience and ease, both initialization and declaration are done in the same step.

Using arrays

```
char c[] = "abcd";
OR,
char c[50] = "abcd";
OR,
char c[] = {'a', 'b', 'c', 'd', '\0'};
OR,
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```



The given string is initialized and stored in the form of arrays as above.

Using pointers

String can also be initialized using pointers as:

```
char *c = "abcd";
```

Reading Strings from user

You can use the scanf() function to read a string like any other data types. However, the scanf() function only takes the first entered word. The function terminates when it encounters a white space (or just space).

Reading words from user

```
char c[20];
scanf("%s", c);
```

Example #1: Using scanf() to read a string

Write a C program to illustrate how to read string from terminal.

```
#include <stdio.h>
int main()
{
   char name[20];
   printf("Enter name: ");
   scanf("%s", name);
   printf("Your name is %s.", name);
   return 0;
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Here, program ignores Ritchie because, scanf() function takes only a single string before the white space, i.e. Dennis.

Reading a line of text

An approach to reading a full line of text is to read and store each character one by one.

Example #2: Using getchar() to read a line of text

1. C program to read line of text character by character.

```
#include <stdio.h>
int main()
{
   char name[30], ch;
   int i = 0;
   printf("Enter name: ");
   while(ch != '\n')    // terminates if user hit enter
   {
      ch = getchar();
      name[i] = ch;
```

```
      i++;
   }
   name[i] = '\0';      // inserting null character at end
   printf("Name: %s", name);
   return 0;
}
```

In the program above, using the function getchar(), ch gets a single character from the user each time. This process is repeated until the user enters return (enter key). Finally, the null character is inserted at the end to make it a string.
This process to take string is tedious.

Example #3: Using standard library function to read a line of text

2. C program to read line of text using gets() and puts()
To make life easier, there are predefined functions gets() and puts in C language to read and display string respectively.

```
#include <stdio.h>
int main()
{
   char name[30];
   printf("Enter name: ");
   gets(name);    //Function to read string from user.
   printf("Name: ");
   puts(name);    //Function to display string.
   return 0;
}
```

Both programs have the same output below:

Output

```
Enter name: Tom Hanks
Name: Tom Hanks
```

Passing Strings to Functions

Strings are just char arrays. So, they can be passed to a function in a similar manner as arrays.

```
#include <stdio.h>
void displayString(char str[]);

int main()
{
   char str[50];
   printf("Enter string: ");
   gets(str);
   displayString(str);    // Passing string c to function.
   return 0;
}
void displayString(char str[]){
   printf("String Output: ");
   puts(str);
}
```

Here, string c is passed from main() function to user-defined function displayString(). In function declaration, str[] is the formal argument.

String handling functions

You need to often manipulate strings according to the need of a problem. Most, if not all, of the time string manipulation can be done manually but, this makes programming complex and large. To solve this, C supports a large number of string handling functions in the **standard library "string.h".**

Few commonly used string handling functions are discussed below:

Function	Work of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

Strings handling functions are defined under "string.h" header file.

#include <string.h>

Note: You have to include the code below to run string handling functions.

gets() and puts()

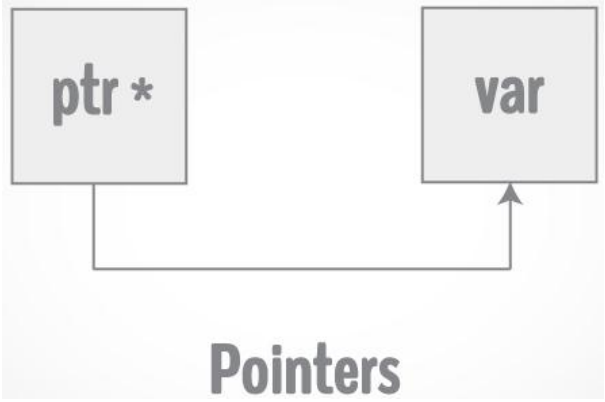
Functions gets() and puts() are two string functions to take string input from the user and display it respectively as mentioned.

```
#include<stdio.h>

int main()
{
   char name[30];
   printf("Enter name: ");
   gets(name);    //Function to read string from user.
   printf("Name: ");
   puts(name);    //Function to display string.
   return 0;
}
```

Note: Though, gets() and puts() function handle strings, both these functions are defined in "stdio.h" header file.

Pointers



Pointers are powerful features of C programming that differentiates it from other popular programming languages like: Java and Python.

Pointers are used in C program to access the memory and manipulate the address.

Address in C

Before you get into the concept of pointers, let's first get familiar with address in C.

If you have a variable **var** in your program, **&var** will give you its address in the memory, where **&** is commonly called the **reference operator**.

You must have seen this notation while using scanf() function. It was used in the function to store the user inputted value in the address of **var**.

scanf("%d", &var);

/* Example to demonstrate use of reference operator in C programming. */

```
#include <stdio.h>
int main()
{
  int var = 5;
  printf("Value: %d\n", var);
  printf("Address: %u", &var);   //Notice, the ampersand(&) before var.
  return 0;
}
```

Output
Value: 5
Address: 2686778

Note: You may obtain different value of address while using this code.
In above source code, value 5 is stored in the memory location 2686778. **var** is just the name given to that location.

Pointer variables
In C, there is a special variable that stores just the address of another variable. It is called **Pointer variable** or, simply, a **pointer**.

Declaration of Pointer

```
data_type* pointer_variable_name;
int* p;
```

Above statement defines, **p** as pointer variable of type int.
Reference operator (&) and **Dereference operator (*)**

As discussed, **&** is called **reference operator**. It gives you the address of a variable.
Likewise, there is another operator that gets you the value from the address, it is called a **dereference operator (*).**

Below example clearly demonstrates the use of pointers, reference operator and dereference operator.

Note: The * sign when declaring a pointer is not a dereference operator. It is just a similar notation that creates a pointer.

Example To Demonstrate Working of Pointers

```
/* Source code to demonstrate, handling of pointers in C program */

#include <stdio.h>
int main(){
  int* pc;
  int c;
  c=22;
  printf("Address of c:%u\n",&c);
  printf("Value of c:%d\n\n",c);
  pc=&c;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  c=11;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  *pc=2;
  printf("Address of c:%u\n",&c);
  printf("Value of c:%d\n\n",c);
  return 0;
}
```

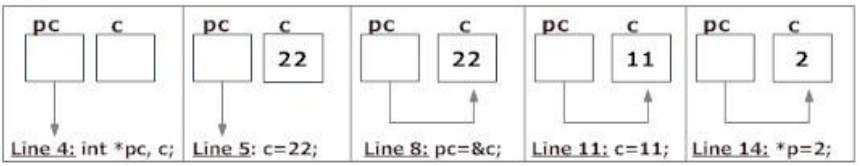
Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11
```

Address of c: 2686784
Value of c: 2



Explanation of program and figure

1. **int* pc;** creates a pointer **pc** and **int c;** creates a normal variable **c**.
   Since **pc** and **c** are both not initialized, pointer **pc** points to either no address or a random address. Likewise, variable **c** is assigned an address but contains a random/garbage value.
2. **c=22;** assigns 22 to the variable **c**, i.e., 22 is stored in the memory location of variable **c**.
   Note that, when printing &c (address of **c**), we use %u rather than %d since address is usually expressed as an unsigned integer (always positive).
3. **pc=&c;** assigns the address of variable to **c** to the pointer **pc**. When printing, you see value of **pc** is the same as the address of **c** and the content of **pc** (*pc) is 22 as well.
4. **c=11;** assigns 11 to variable **c**.
   We assign a new value to **c** to see its effect on pointer **pc**.
5. Since, pointer **pc** points to the same address as **c**, value pointed by pointer **pc** is 11 as well. Printing the address and content of **pc** shows the updated content as 11.
6. ***pc=2;** changes the contents of the memory location pointed by pointer **pc** to 2. Since the address of pointer **pc** is same as address of **c**, value of **c** also changes to 2.

Common mistakes when working with pointers

Suppose, you want pointer pc to point to the address of c. Then,

int c, *pc;

// Wrong! pc is address whereas, c is not an address.

pc = c;

// Wrong! *pc is the value pointed by address whereas, %amp;c is an address.

***pc = &c;**

// Correct! pc is an address and, %amp;pc is also an address.

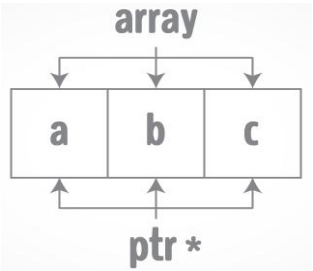
pc = &c;

// Correct! *pc is the value pointed by address and, c is also a value.

***pc = c;**

In both cases, pointer pc is not pointing to the address of c.

Pointers and Arrays



Arrays are closely related to **pointers** in C programming but the important difference between them is that, a pointer variable takes different addresses as value whereas, in case of array it is fixed.

This can be demonstrated by an example:

```
#include <stdio.h>
int main()
{
  char charArr[4];
  int i;

  for(i = 0; i < 4; ++i)
  {
    printf("Address of charArr[%d] = %u\n", i, &charArr[i]);
  }

  return 0;
}
```

When you run the program, the output will be:
Address of charArr[0] = 28ff44
Address of charArr[1] = 28ff45
Address of charArr[2] = 28ff46
Address of charArr[3] = 28ff47

Note: You may get different address of an array.
Notice, that there is an equal difference (difference of 1 byte) between any two consecutive elements of array **charArr**.
But, since pointers just point at the location of another variable, it can store any address.

Relation between Arrays and Pointers
Consider an array:

int arr[4];

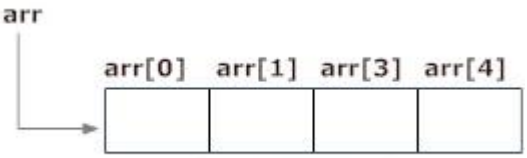


Figure: Array as Pointer

In C programming, name of the array always points to address of the first element of an array.

In the above example, **arr** and **&arr[0]** points to the address of the first element.

&arr[0] is equivalent to arr

Since, the addresses of both are the same, the values of **arr** and

&arr[0] are also the same.

arr[0] is equivalent to ***arr** (value of an address of the pointer)

Similarly,
&arr[1] is equivalent to **(arr + 1)** AND**, arr[1]** is equivalent to ***(arr + 1).**
&arr[2] is equivalent to **(arr + 2)** AND, **arr[2]** is equivalent to ***(arr + 2).**
&arr[3] is equivalent to **(arr + 3)** AND, **arr[3]** is equivalent to ***(arr + 3).**

&arr[i] is equivalent to **(arr + i)** AND, **arr[i]** is equivalent to ***(arr + i).**

In C, you can declare an array and can use pointer to alter the data of an array.

Example: Program to find the sum of six numbers with arrays and pointers

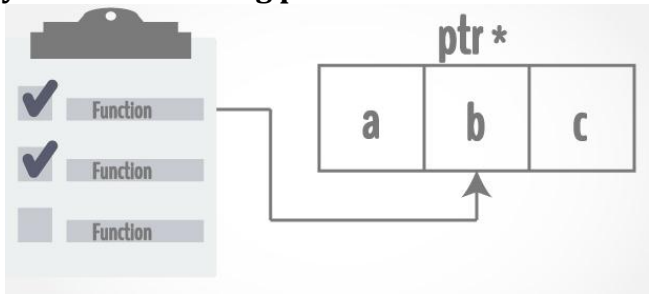
```
#include <stdio.h>
int main()
{
 int i, classes[6],sum = 0;
 printf("Enter 6 numbers:\n");
 for(i = 0; i < 6; ++i)
 {
    // (classes + i) is equivalent to &classes[i]
    scanf("%d",(classes + i));

    // *(classes + i) is equivalent to classes[i]
    sum += *(classes + i);
 }
 printf("Sum = %d", sum);
 return 0;
}
```

Output

```
Enter 6 numbers:
2
3
4
5
3
4
Sum = 21
```

C Call by Reference: Using pointers



When a **pointer** is passed as an argument to a **function**, address of the memory location is passed instead of the value. This is because, pointer stores the location of the memory, and not the value.

Example of Pointer and Functions

Program to swap two number using call by reference.

```
 /* C Program to swap two numbers using pointers and function.
*/
#include <stdio.h>
void swap(int *n1, int *n2);

int main()
{
   int num1 = 5, num2 = 10;

   // address of num1 and num2 is passed to the swap function
   swap( &num1, &num2);
   printf("Number1 = %d\n", num1);
   printf("Number2 = %d", num2);
   return 0;
}

void swap(int * n1, int * n2)
{
   // pointer n1 and n2 points to the address of num1 and num2
respectively
   int temp;
   temp = *n1;
```

```
   *n1 = *n2;
   *n2 = temp;
}
```

Output

```
Number1 = 10
Number2 = 5
```

The address of memory location **num1** and **num2** are passed to the function swap and the pointers ***n1** and ***n2** accept those values. So, now the pointer **n1** and **n2** points to the address of **num1** and **num2** respectively.

When, the value of pointers are changed, the value in the pointed memory location also changes correspondingly. Hence, changes made to ***n1** and ***n2** are reflected in **num1** and **num2** in the main function.

This technique is known **as Call by Reference** in C programming.