Programming in C

Module 1 :
Recall basic programming concepts – C program structure, selection structure and repetition structures.Function – Declarations, prototype, definition, function call, storage class, lifetime and visibility of variables.Preprocessor – file inclusion – macro substitution Recursion – Recursive definition of a problem, Implementation of programs using recursion.

C program structure

Hello World Example

A C program basically consists of the following parts −

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World" −

```
#include <stdio.h>

int main() {
   /* my first program in C */
   printf("Hello, World! \n");

   return 0;
}
```

Let us take a look at the various parts of the above program −

- The first line of the program *#include <stdio.h>* is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.
- The next line *int main()* is the main function where the program execution begins.
- The next line /*...*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
- The next line *printf(...)* is another function available in C which causes the message "Hello, World!" to be displayed on the screen.
- The next line return 0; terminates the main() function and returns the value 0.

Compile and Execute C Program

Let us see how to save the source code in a file, and how to compile and run it. Following are the simple steps −

- Open a text editor and add the above-mentioned code.
- Save the file as *hello.c*
- Open a command prompt and go to the directory where you have saved the file.
- Type *gcc hello.c* and press enter to compile your code.
- If there are no errors in your code, the command prompt will take you to the next line and would generate *a.out* executable file.
- Now, type *a.out* to execute your program.
- You will see the output *"Hello World"* printed on the screen.

```
$ gcc hello.c
$ ./a.out
Hello, World!
```

Make sure the gcc compiler is in your path and that you are running it in the directory containing the source file hello.c.

You have seen the basic structure of a C program, so it will be easy to understand other basic building blocks of the C programming language.

Tokens in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens −

```
printf("Hello, World! \n");
```

The individual tokens are −

```
printf
(
   "Hello, World! \n"
)
;
```

Semicolons

In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

Given below are two different statements −

```
printf("Hello, World! \n");
return 0;
```

Comments

Comments are like helping text in your C program and they are ignored by the compiler. They start with /* and terminate with the characters */ as shown below −

```
/* my first program in C */
```

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, $, and % within identifiers. C is a case-sensitive programming language. Thus, *Manpower* and *manpower* are two different identifiers in C. Here are some examples of acceptable identifiers −

```
mohd       zara    abc   move_name  a_123
myname50   _temp   j     a23b9      retVal
```

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void

continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows −

```
type variable_list;
```

Here, type must be a valid C data type including char, w_char, int, float, double, bool, or any user-defined object; and variable_list may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

C Data Types

In this tutorial, you will learn about basic data types such as int, float, char etc. in C programming.

Basic types

Here's a table containing commonly used types in C programming for quick access.

Type	Size (bytes)	Format Specifier

int	at least 2, usually 4	%d, %i
char	1	%c
float	4	%f
double	8	%lf
short int	2 usually	%hd

Variable Declaration in C

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only, the compiler needs actual variable definition at the time of linking the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use the keyword extern to declare a variable at any place. Though you can declare a variable multiple times in your C program, it can be defined only once in a file, a function, or a block of code.

Example

Try the following example, where variables have been declared at the top, but they have been defined and initialized inside the main function −

```
#include <stdio.h>

// Variable declaration:

extern int a, b;

extern int c;

extern float f;
```

```
int main () {

   /* variable definition: */
   int a, b;
   int c;
   float f;

   /* actual initialization */
   a = 10;
   b = 20;

   c = a + b;
   printf("value of c : %d \n", c);

   f = 70.0/3.0;
   printf("value of f : %f \n", f);

   return 0;
}
```

Conditions and If Statements

You learned from the operators comparison chapter, that C supports the usual logical conditions from mathematics:

- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b
- Equal to a == b
- Not Equal to: a != b

You can use these conditions to perform different actions for different decisions.

C has the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false
- Use switch to specify many alternative blocks of code to be executed

---

The if Statement

Use the if statement to specify a block of C code to be executed if a condition is true.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

Example

```
if (20 > 18) {
  printf("20 is greater than 18");
}
```

The else Statement

Use the else statement to specify a block of code to be executed if the condition is false.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

Example

```
int time = 20;

if (time < 18) {

  printf("Good day.");

} else {

  printf("Good evening.");

}

// Outputs "Good evening."
```

The else if Statement

Use the else if statement to specify a new condition if the first condition is false.

Syntax

```
if (condition1) {

  // block of code to be executed if condition1 is true

} else if (condition2) {

  // block of code to be executed if the condition1 is false and condition2 is true

} else {

  // block of code to be executed if the condition1 is false and condition2 is false

}
```

Example

```
int time = 22;

if (time < 10) {

  printf("Good morning.");

} else if (time < 20) {

  printf("Good day.");

} else {

  printf("Good evening.");

}

// Outputs "Good evening."
```

Control Structures - Repetition

Repetition Statements

- Repetition statements are called *loops*, and are used to repeat the same code mulitple times in succession.
- The number of repetitions is based on criteria defined in the loop structure, usually a true/false expression
- The three loop structures in C++ are:
    - **while** loops
    - **do-while** loops
    - **for** loops
- Three types of loops are not actually needed, but having the different forms is convenient

while and do-while loops

Format of while loop:

```
while (expression)
  statement
```

Format of do/while loop:

do
  statement
while (*expression*);

- The *expression* in these formats is handled the same as in the if/else statements discussed previously (0 means false, anything else means true)
- The "statement" portion is also as in if/else. It can be a single statement or a compund statement (a block { }).

We could also write the formats as follows (illustrating more visually what they look like when a compound statement makes up the loop "body"):

while loop format

```
while (expression)
{
  statement1;
  statement2;
  // ...

  statementN;
}
```

do-while loop format

```
do
{
  statement1;
  statement2;
  // ...

  statementN;
} while (expression);
```

- **HOW THEY WORK**
  - The expression is a test condition that is evaluated to decide whether the loop should repeat or not.
    - **true** means run the loop body again.
    - **false** means quit.
  - The while and do/while loops both follow the same basic flowchart -- the only exception is that:
    - In a while loop, the expression is tested first
    - In a do/while loop, the loop "body" is executed first

The for loop

- The **for** loop is most convenient with *counting loops* -- i.e. loops that are based on a counting variable, usually a known number of iterations

Format of for loop:

for (*initialCondition*; *testExpression*; *iterativeStatement*)

statement

Remember that the statement can be a single statement or a compound statement (block), so an alternate way to write the format might be:

```
for (initialCondition; testExpression; iterativeStatement)
{
  statement1;
  statement2;
  // ...

  statementN;
}
```

- How it works
  - The *initialCondition* runs once, at the start of the loop
  - The *testExpression* is checked. (This is just like the expression in a while loop). If it's false, quit. If it's true, then:
    - Run the loop body
    - Run the *iterativeStatement*
    - Go back to the *testExpression* step and repeat

Function

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, strcat() to concatenate two strings, memcpy() to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

The general form of a function definition in C programming language is as follows −

```
return_type function_name( parameter list )
{
  body of the function
}
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

- Return Type − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
- Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- Function Body − The function body contains a collection of statements that define what the function does.

Function Declarations

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

```
return_type function_name( parameter list );
```

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A

called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

```
function_name( parameter list );
```

Example:

```
#include <stdio.h>

/* function declaration */

int max(int num1, int num2);

 int main () {

   /* local variable definition */

   int a = 100;

   int b = 200;

   int ret;

   /* calling a function to get max value */

   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;

}

/* function returning the max between two numbers */

int max(int num1, int num2) {

   /* local variable declaration */

   int result;

   if (num1 > num2)

      result = num1;

   else
```

```
    result = num2;

  return result;

}
```

C - Storage Classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program −

- auto
- register
- static
- extern

The auto Storage Class

The auto storage class is the default storage class for all local variables.

```
{
  int mount;
  auto int month;
}
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class

The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when static is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

```
static int count = 5; /* global variable */
```

The extern Storage Class

The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

```
int count ;
extern void write_extern();
```

C - Preprocessors

The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives −

Sr.No.	Directive & Description

1	#define Substitutes a preprocessor macro.
2	#include Inserts a particular header from another file.
3	#undef Undefines a preprocessor macro.
4	#ifdef Returns true if this macro is defined.
5	#ifndef Returns true if this macro is not defined.
6	#if Tests if a compile time condition is true.

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

C - Recursion

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {
```

```
  recursion(); /* function calls itself */
}

int main() {
  recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Number Factorial

The following example calculates the factorial of a given number using a recursive function

```
#include <stdio.h>
unsigned long long int factorial(unsigned int i) {
  if(i <= 1) {
    return 1;
  }
  return i * factorial(i - 1);
}
int  main() {
  int i = 12;
  printf("Factorial of %d is %d\n", i, factorial(i));
  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Factorial of 12 is 479001600
```

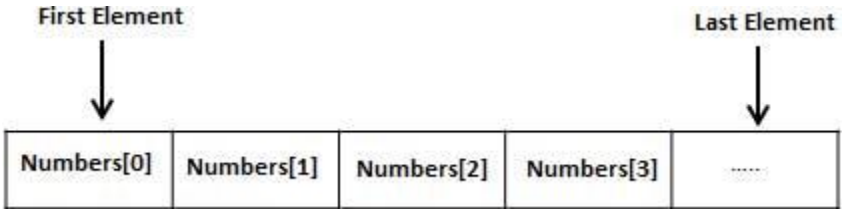
Module 2 :
Arrays – definition, initialization and processing of arrays – Searching algorithms – Linear search, Binary Search, Sorting algorithms – Selection sort, Quick sort,Passing arrays to functions - Strings – Representation of strings in C – String input and output - String processing – copy, concentrate, length,comparison, pattern searching etc - builtin String functions – Implementation of string functions.

C - 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 [ ].

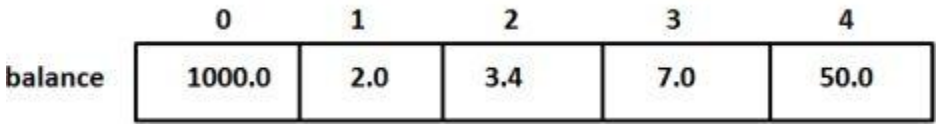
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

```
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 −



Searching Algorithms

The searching algorithms are used to search or find one or more than one element from a dataset. These type of algorithms are used to find elements from a specific data structures.

Searching may be sequential or not. If the data in the dataset are random, then we need to use sequential searching. Otherwise we can use other different techniques to reduce the complexity.

In this Section We are going to cover −

- Linear Search
- Binary Search

Linear Search Algorithm

In this article, we will discuss the Linear Search Algorithm. Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.

Two popular search methods are Linear Search and Binary Search. So, here we will discuss the popular searching technique, i.e., Linear Search Algorithm.

Linear search is also called as **sequential search algorithm.** It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

The steps used in the implementation of Linear Search are listed as follows -

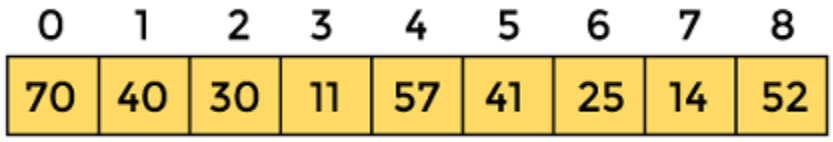
- First, we have to traverse the array elements using a **for** loop.
- In each iteration of **for loop,** compare the search element with the current array element, and -
    - If the element matches, then return the index of the corresponding array element.
    - If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return **-1.**

Working of Linear search

Now, let's see the working of the linear search Algorithm.

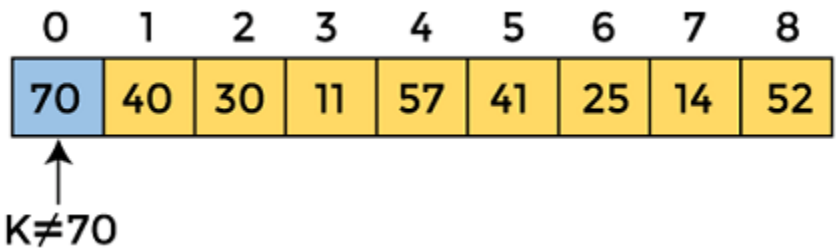
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

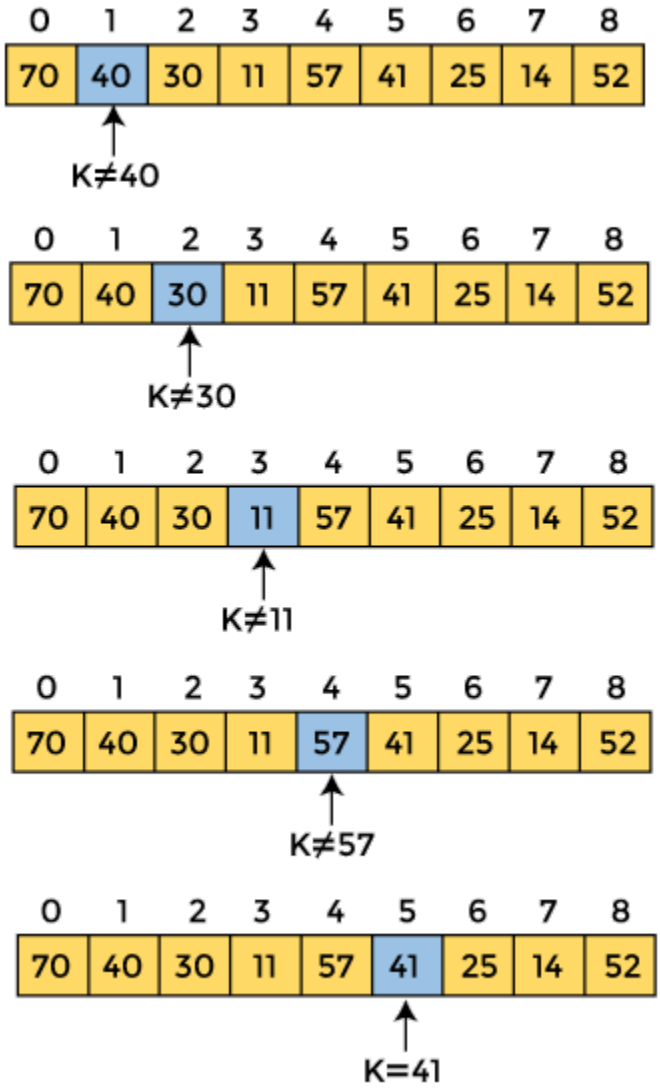


Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.



The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



Binary Search Algorithm

In this article, we will discuss the Binary Search Algorithm. Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element. Otherwise, the search is called unsuccessful.

Linear Search and Binary Search are the two popular searching techniques. Here we will discuss the Binary Search Algorithm.

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Working of Binary search

Now, let's see the working of the Binary Search Algorithm.

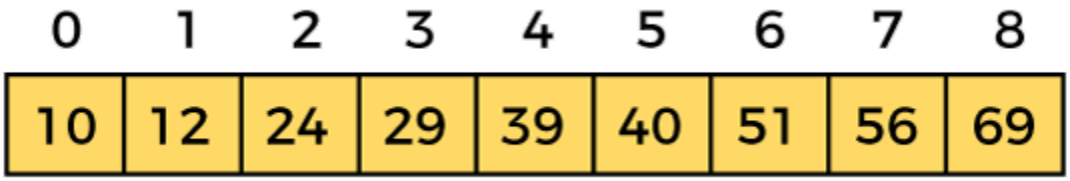
To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.

There are two methods to implement the binary search algorithm -

- Iterative method
- Recursive method

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -



Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

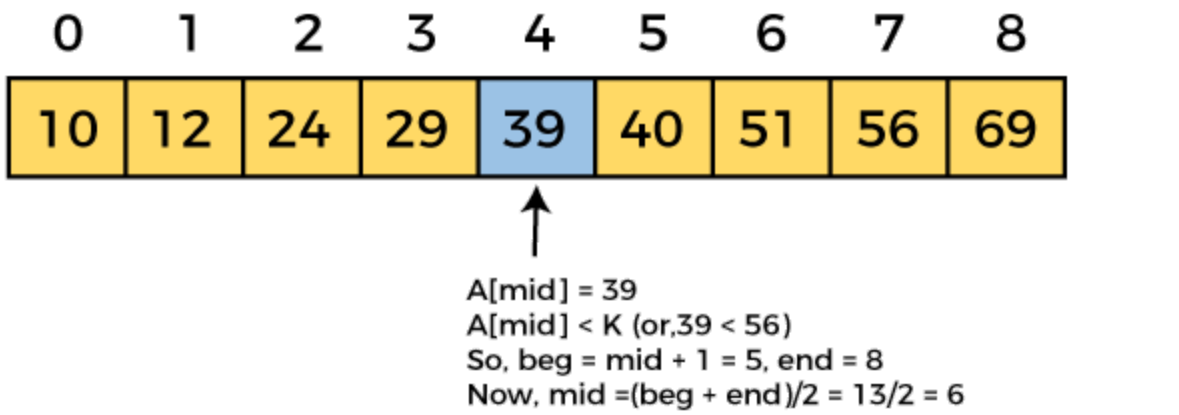
1. mid = (beg + end)/2

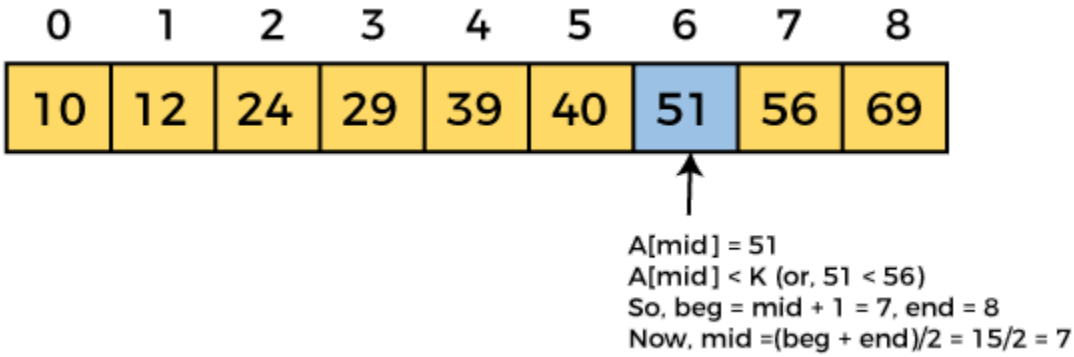
So, in the given array -

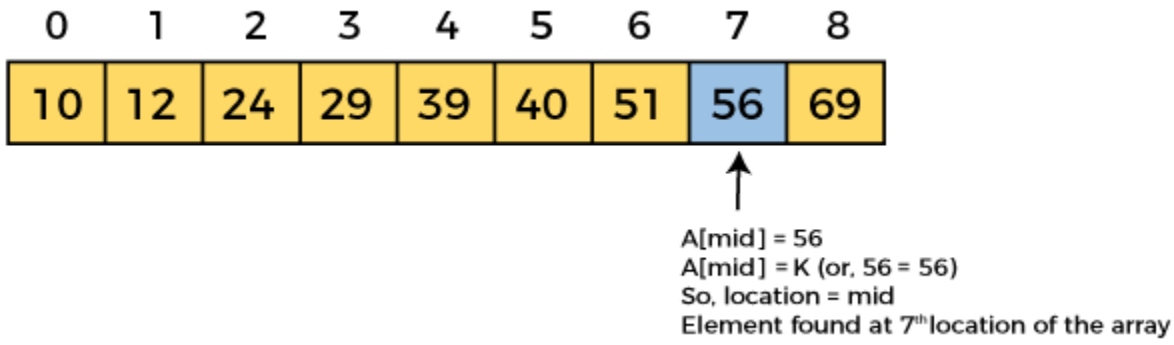
beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.







Sorting algorithms in C language

C language provides five sorting techniques, which are as follows −

- Bubble sort (or) Exchange Sort.
- Selection sort.

- Insertion sort (or) Linear sort.
- Quick sort (or) Partition exchange sort.
- Merge Sort (or) External sort.

Selection Sort Algorithm

In this article, we will discuss the Selection sort Algorithm. The working procedure of selection sort is also simple. This article will be very helpful and interesting to students as they might face selection sort as a question in their examinations. So, it is important to discuss the topic.

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

Working of Selection sort Algorithm

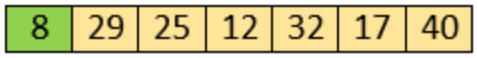
Now, let's see the working of the Selection sort Algorithm.To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are -

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

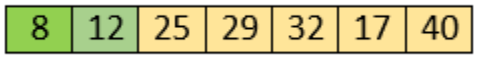
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

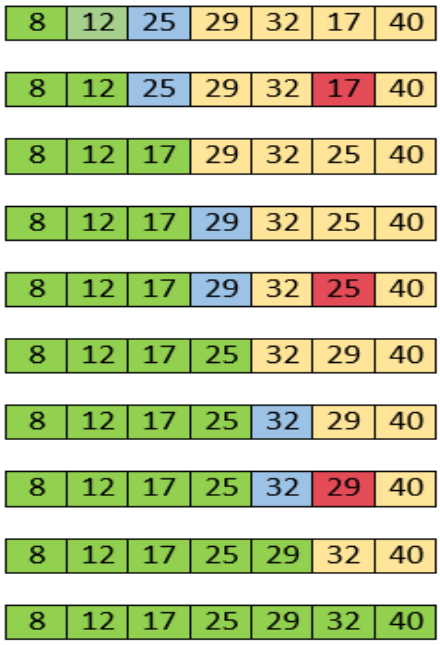


For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.



The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Quick Sort Algorithm

In this article, we will discuss the Quicksort Algorithm. The working procedure of Quicksort is also simple. This article will be very helpful and interesting to students as they might face quicksort as a question in their examinations. So, it is important to discuss the topic.

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

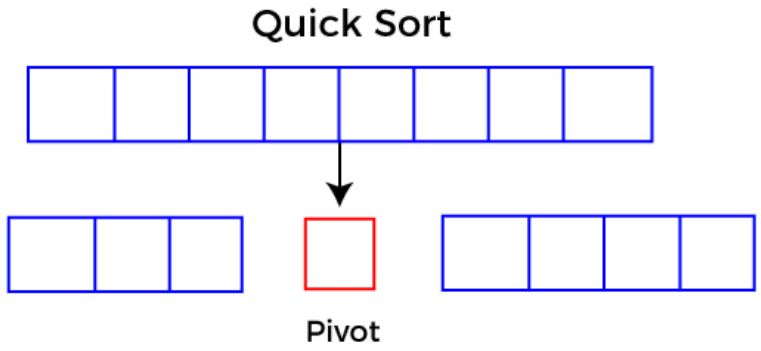
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.

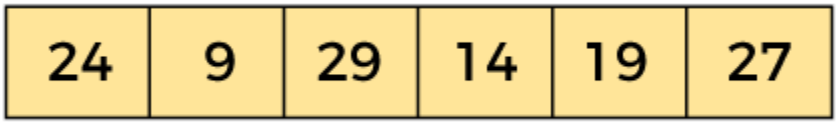
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

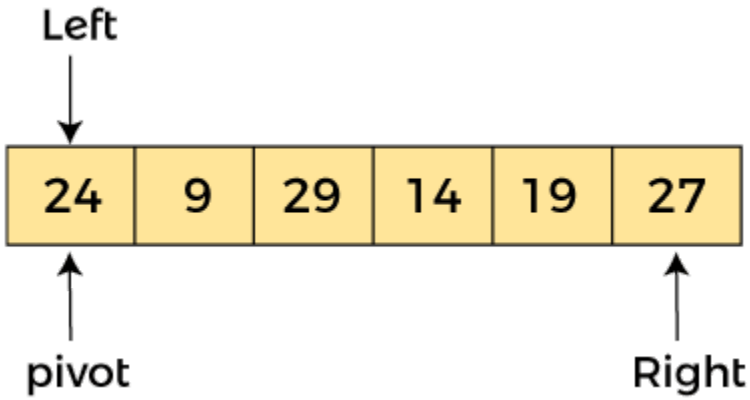
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

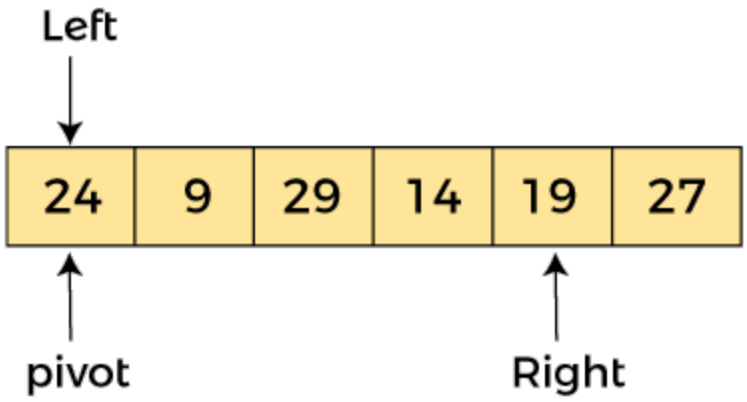


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

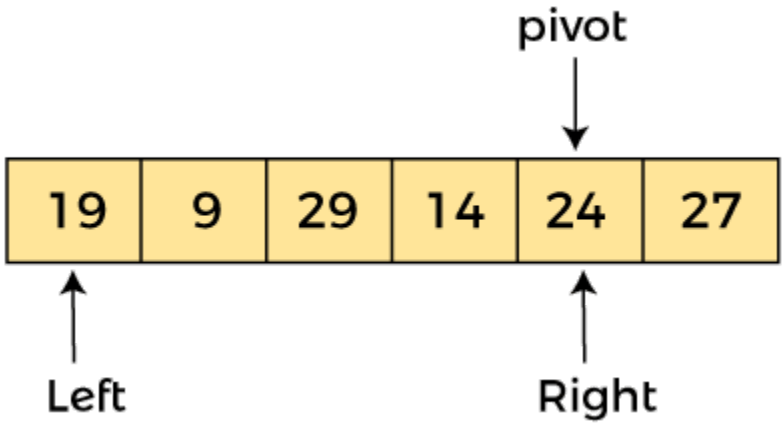


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



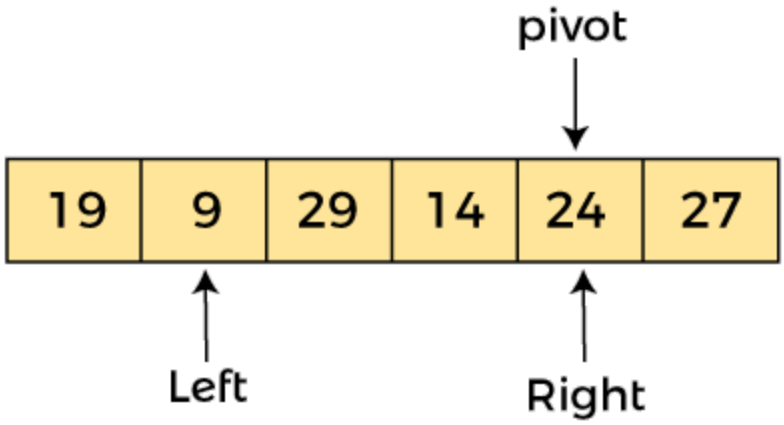
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

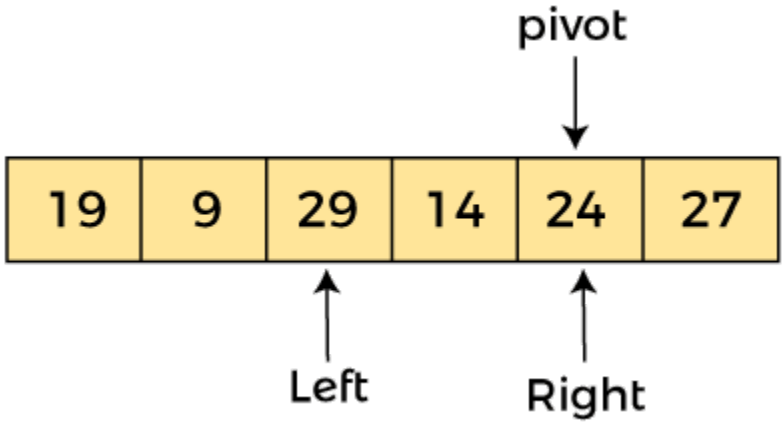


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

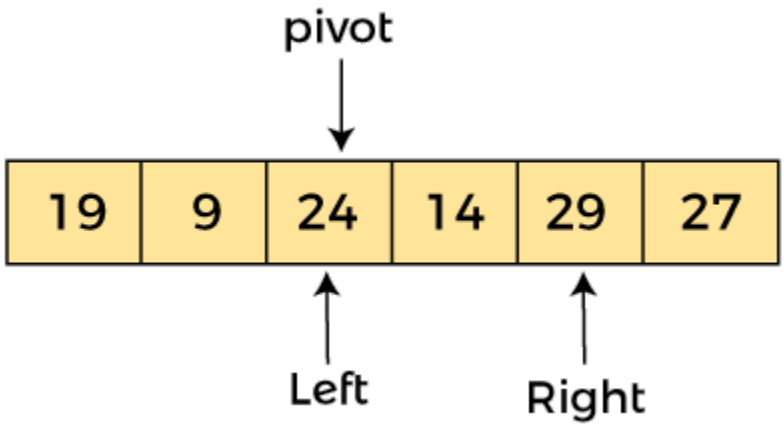
As a[pivot] > a[left], so algorithm moves one position to right as -



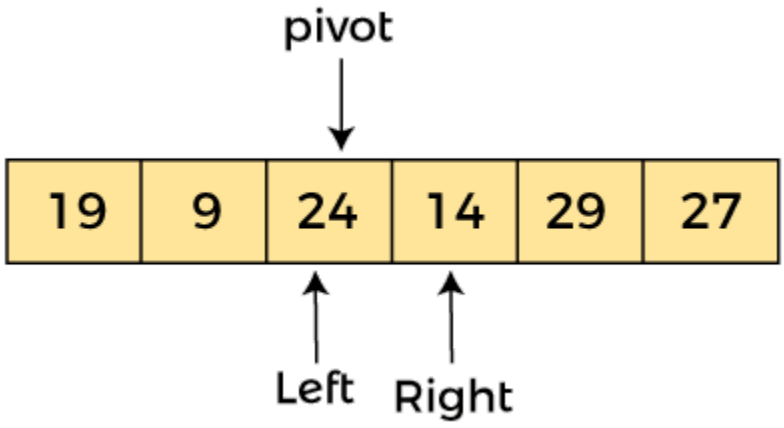
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



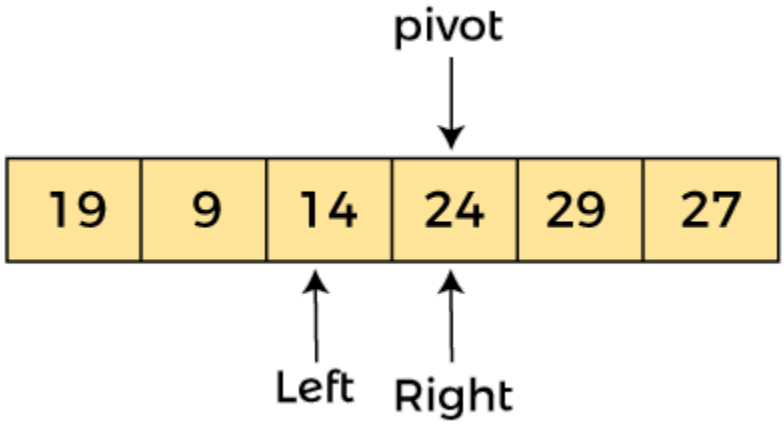
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



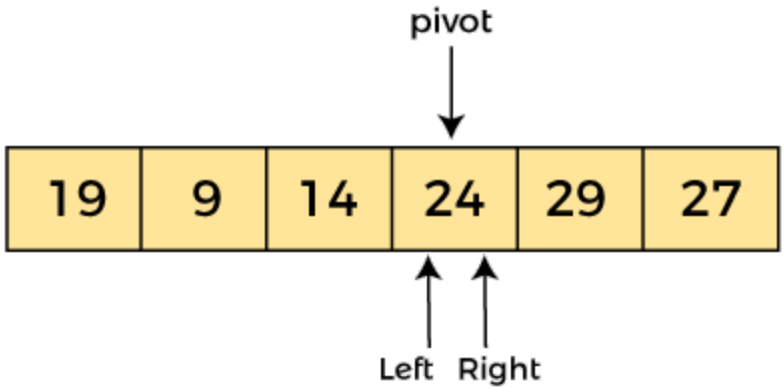
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



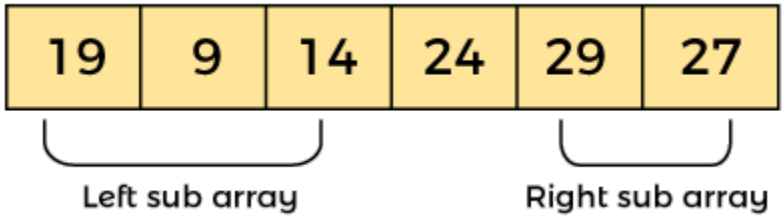
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



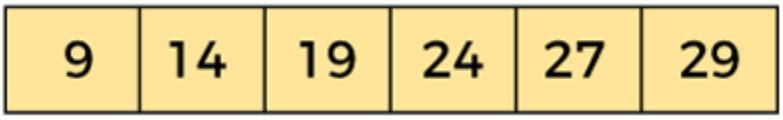
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



C - Strings

Strings are actually one-dimensional array of characters terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

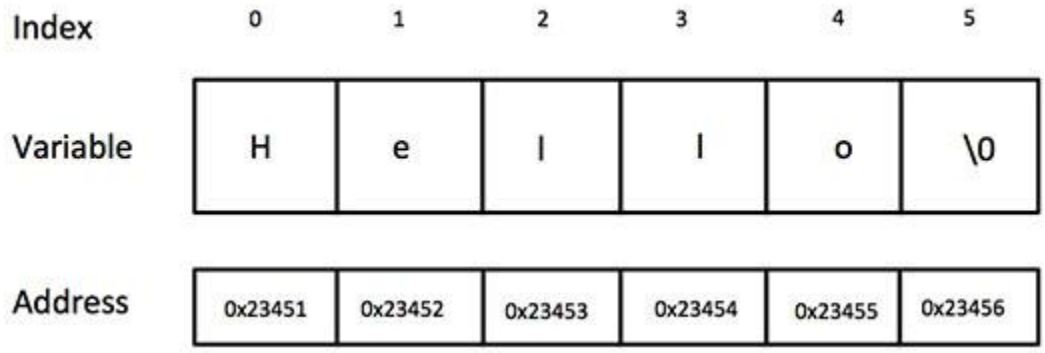
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

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

If you follow the rule of array initialization then you can write the above statement as follows −

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++ −



Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string −

```
#include <stdio.h>

int main () {
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   printf("Greeting message: %s\n", greeting );
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null-terminated strings −

Sr.No.	Function & Purpose
1	strcpy(s1, s2); Copies string s2 into string s1.
2	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
3	strlen(s1); Returns the length of string s1.
4	strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions −

```
#include <stdio.h>

#include <string.h>

int main () {

  char str1[12] = "Hello";

  char str2[12] = "World";
```

```
   char str3[12];
   int  len ;
   /* copy str1 into str3 */
   strcpy(str3, str1);
   printf("strcpy( str3, str1) :  %s\n", str3 );
   /* concatenates str1 and str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):   %s\n", str1 );
   /* total lenghth of str1 after concatenation */
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

Module - 03
Pointers – Fundamentals – declaration, Initialization, accessing of pointer variables -Pointer arithmetic – Passing pointers to Functions – dynamic memory allocation- Arrays and Pointers - Strings and Pointers – Array of Pointers.

C Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

1. **int** n = 10;
2. **int*** p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.

Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as an indirection pointer used to dereference a pointer.

1. **int** *a;//pointer to int
2. **char** *c;//pointer to char

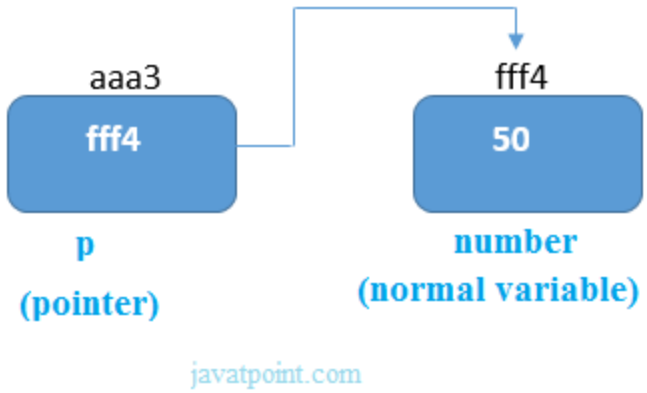
Pointer Example

An example of using pointers to print the address and value is given below.

67.3M

1.1K

Hello Java Program for Beginners



As you can see in the above figure, the pointer variable stores the address of the number variable, i.e., fff4. The value of the number variable is 50. But the address of the pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of the pointer variable p.

Let's see the pointer example as explained in the above figure.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.
7. printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.
8. return 0;
9. }
```

Output

Address of number variable is fff4

Address of p variable is fff4

Value of p variable is 50

Pointer to array

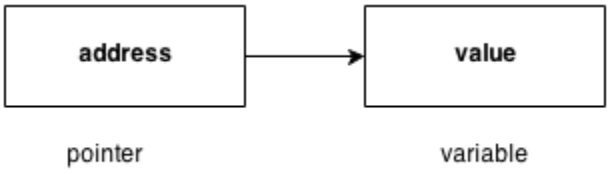
```
1. int arr[10];
2. int *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.
```

Pointer to a function

```
1. void show (int);
2. void(*p)(int) = &display; // Pointer p is pointing to the address of a function
```

Pointer to structure

```
1. struct st {
2.     int i;
3.     float f;
4. }ref;
5. struct st *p = &ref;
```



Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2) We can **return multiple values from a function** using the pointer.

3) It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many applications of pointers in the C language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. printf("value of number is %d, address of number is %u",number,&number);
5. return 0;
6. }
```

Output

value of number is 50, address of number is fff4

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

int *p=NULL;

In most libraries, the value of the pointer is 0 (zero).

C - Pointer arithmetic

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer −

ptr++

After the above operation, the ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If ptr points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array −

```
#include <stdio.h>

const int MAX = 3;

int main () {

  int  var[] = {10, 100, 200};

  int  i, *ptr;

  /* let us have array address in pointer */

  ptr = var;

  for ( i = 0; i < MAX; i++) {

    printf("Address of var[%d] = %x\n", i, ptr );

    printf("Value of var[%d] = %d\n", i, *ptr );

    /* move to the next location */

    ptr++;

  }

  return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var[0] = bf882b30

Value of var[0] = 10

Address of var[1] = bf882b34

Value of var[1] = 100

Address of var[2] = bf882b38

Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below −

```
#include <stdio.h>

const int MAX = 3;

int main () {

  int  var[] = {10, 100, 200};

  int  i, *ptr;

  /* let us have array address in pointer */

  ptr = &var[MAX-1];

  for ( i = MAX; i > 0; i--) {

    printf("Address of var[%d] = %x\n", i-1, ptr );

    printf("Value of var[%d] = %d\n", i-1, *ptr );

    /* move to the previous location */

    ptr--;

  }

  return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var[2] = bfedbcd8

Value of var[2] = 200

Address of var[1] = bfedbcd4

Value of var[1] = 100

Address of var[0] = bfedbcd0

Value of var[0] = 10
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example − one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1] −

```
#include <stdio.h>

const int MAX = 3;

int main () {

  int  var[] = {10, 100, 200};

  int  i, *ptr;

  /* let us have address of the first element in pointer */

  ptr = var;

  i = 0;

  while ( ptr <= &var[MAX - 1] ) {

    printf("Address of var[%d] = %x\n", i, ptr );

    printf("Value of var[%d] = %d\n", i, *ptr );

    /* point to the next location */

    ptr++;

    i++;

  }

  return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var[0] = bfdbcb20

Value of var[0] = 10

Address of var[1] = bfdbcb24
```

Value of var[1] = 100

Address of var[2] = bfdbcb28

Value of var[2] = 200

Passing pointers to functions in C

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function −

```
#include <stdio.h>

#include <time.h>

void getSeconds(unsigned long *par);

int main () {

   unsigned long sec;

   getSeconds( &sec );

   /* print the actual value */

   printf("Number of seconds: %ld\n", sec );

   return 0;

}

void getSeconds(unsigned long *par) {

   /* get the current number of seconds */

   *par = time( NULL );

   return;

}
```

When the above code is compiled and executed, it produces the following result −

Number of seconds :1294450468

The function, which can accept a pointer, can also accept an array as shown in the following example −

```
#include <stdio.h>

/* function declaration */

double getAverage(int *arr, int size);

int main () {

   /* an int array with 5 elements */

   int balance[5] = {1000, 2, 3, 17, 50};

   double avg;

   /* pass pointer to the array as an argument */

   avg = getAverage( balance, 5 ) ;

   /* output the returned value  */

   printf("Average value is: %f\n", avg );

   return 0;

}

double getAverage(int *arr, int size) {

   int  i, sum = 0;

   double avg;

   for (i = 0; i < size; ++i) {

      sum += arr[i];

   }

   avg = (double)sum / size;

   return avg;

}
```

When the above code is compiled together and executed, it produces the following result −

```
Average value is: 214.40000
```

Dynamic memory allocation in C

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

static memory allocation	**dynamic memory allocation**
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.

free()	frees the dynamically allocated memory.

malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

```
1. ptr=(cast-type*)malloc(byte-size)
```

Let's see the example of malloc() function.

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. int main(){
4.   int n,i,*ptr,sum=0;
5.     printf("Enter number of elements: ");
6.     scanf("%d",&n);
7.     ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
8.     if(ptr==NULL)
9.     {
10.        printf("Sorry! unable to allocate memory");
11.        exit(0);
12.    }
13.    printf("Enter elements of array: ");
14.    for(i=0;i<n;++i)
15.    {
16.        scanf("%d",ptr+i);
17.        sum+=*(ptr+i);
18.    }
19.    printf("Sum=%d",sum);
20.    free(ptr);
21. return 0;
22. }
```

Output

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

1. ptr=(cast-type*)calloc(number, byte-size)

Let's see the example of calloc() function.

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. int main(){
4.  int n,i,*ptr,sum=0;
5.     printf("Enter number of elements: ");
6.     scanf("%d",&n);
7.     ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
8.     if(ptr==NULL)
9.     {
10.       printf("Sorry! unable to allocate memory");
11.       exit(0);
12.    }
13.    printf("Enter elements of array: ");
14.    for(i=0;i<n;++i)
15.    {
16.       scanf("%d",ptr+i);
17.       sum+=*(ptr+i);
18.    }
19.    printf("Sum=%d",sum);
20.    free(ptr);
21. return 0;
```

22. }

Output

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

1. ptr=realloc(ptr, **new**-size)

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

```
free(ptr)
```

Relationship Between Arrays and Pointers

In this tutorial, you'll learn about the relationship between arrays and pointers in C programming. You will also learn to access array elements using pointers.
Before you learn about the relationship between arrays and pointers, be sure to check these two topics:

- C Arrays
- C Pointers

Relationship Between Arrays and Pointers

An array is a block of sequential data. Let's write a program to print addresses of array elements.

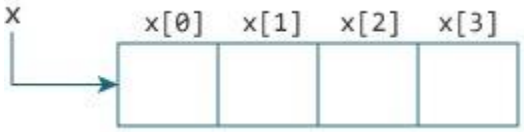
```
#include <stdio.h>
int main() {
  int x[4];
  int i;

  for(i = 0; i < 4; ++i) {
    printf("&x[%d] = %p\n", i, &x[i]);
  }

  printf("Address of array x: %p", x);

  return 0;
}
```

Output

```
&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448
```

There is a difference of 4 bytes between two consecutive elements of array x. It is because the size of int is 4 bytes (on our compiler).

Notice that, the address of &x[0] and x is the same. It's because the variable name x points to the first element of the array.



Relation between Arrays and Pointers

From the above example, it is clear that &x[0] is equivalent to x. And, x[0] is equivalent to *x.

Similarly,

- &x[1] is equivalent to x+1 and x[1] is equivalent to *(x+1).
- &x[2] is equivalent to x+2 and x[2] is equivalent to *(x+2).
- ...
- Basically, &x[i] is equivalent to x+i and x[i] is equivalent to *(x+i).

Example 1: Pointers and Arrays

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

```
 int i, x[6], sum = 0;

 printf("Enter 6 numbers: ");

 for(i = 0; i < 6; ++i) {
 // Equivalent to scanf("%d", &x[i]);
    scanf("%d", x+i);

 // Equivalent to sum += x[i]
    sum += *(x+i);
 }

 printf("Sum = %d", sum);

 return 0;
}
```

When you run the program, the output will be:

```
Enter 6 numbers: 2
3
4
4
12
4
Sum = 29
```

C - Array of pointers

Before we understand the concept of arrays of pointers, let us consider the following example, which uses an array of 3 integers −

```
#include <stdio.h>
const int MAX = 3;
int main () {

  int  var[] = {10, 100, 200};
  int i;

  for (i = 0; i < MAX; i++) {
    printf("Value of var[%d] = %d\n", i, var[i] );
  }

  return 0;
```

```
}
```
When the above code is compiled and executed, it produces the following result −
```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer −
```
int *ptr[MAX];
```

It declares ptr as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows −
Live Demo
```
#include <stdio.h>
const int MAX = 3;
int main () {

  int  var[] = {10, 100, 200};
  int i, *ptr[MAX];

  for ( i = 0; i < MAX; i++) {
    ptr[i] = &var[i]; /* assign the address of integer. */
  }

  for ( i = 0; i < MAX; i++) {
    printf("Value of var[%d] = %d\n", i, *ptr[i] );
  }

  return 0;
}
```
When the above code is compiled and executed, it produces the following result −
```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

You can also use an array of pointers to character to store a list of strings as follows −
Live Demo
```
#include <stdio.h>

const int MAX = 4;

int main () {

  char *names[] = {
```

```
      "Zara Ali",
      "Hina Ali",
      "Nuha Ali",
      "Sara Ali"
   };

   int i = 0;

   for ( i = 0; i < MAX; i++) {
      printf("Value of names[%d] = %s\n", i, names[i] );
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

Module 04

Structure – declaration, definition and initialization of structure variables, Accessing of structure elements – Array of structure – Structure and Pointer – Structure and Function – Union - enumerations.

File – Defining, opening, closing a file - input and output operations on sequential files - Command Line arguments.

C Structure

Why use structure?

In C, there are cases where we need to store multiple attributes of an entity. It is not necessary that an entity has all the information of one type only. It can have different attributes of different data types. For example, an entity Student may have its name (string), roll number (int), marks (float). To store such type of information regarding an entity student, we have the following approaches:

- Construct individual arrays for storing names, roll numbers, and marks.
- Use a special data structure to store the collection of different data types.

Let's look at the first approach in detail.

```
1.  #include<stdio.h>
2.  void main ()
3.  {
4.    char names[2][10],dummy; // 2-dimensioanal character array names is used to store the
    names of the students
5.    int roll_numbers[2],i;
6.    float marks[2];
7.    for (i=0;i<3;i++)
8.    {
9.
10.     printf("Enter the name, roll number, and marks of the student %d",i+1);
11.     scanf("%s %d %f",&names[i],&roll_numbers[i],&marks[i]);
12.     scanf("%c",&dummy); // enter will be stored into dummy character at each iteration
13.   }
14.   printf("Printing the Student details ...\n");
15.   for (i=0;i<3;i++)
16.   {
17.     printf("%s %d %f\n",names[i],roll_numbers[i],marks[i]);
18.   }
19. }
```

Output

Enter the name, roll number, and marks of the student 1Arun 90 91
Enter the name, roll number, and marks of the student 2Varun 91 56
Enter the name, roll number, and marks of the student 3Sham 89 69

Printing the Student details...
Arun 90 91.000000
Varun 91 56.000000
Sham 89 69.000000

The above program may fulfill our requirement of storing the information of an entity student. However, the program is very complex, and the complexity increase with the amount of the input. The elements of each of the array are stored contiguously, but all the arrays may not be stored contiguously in the memory. C provides you with an additional and simpler approach where you can use a special data structure, i.e., structure, in which, you can group all the information of different data type regarding an entity.

What is Structure

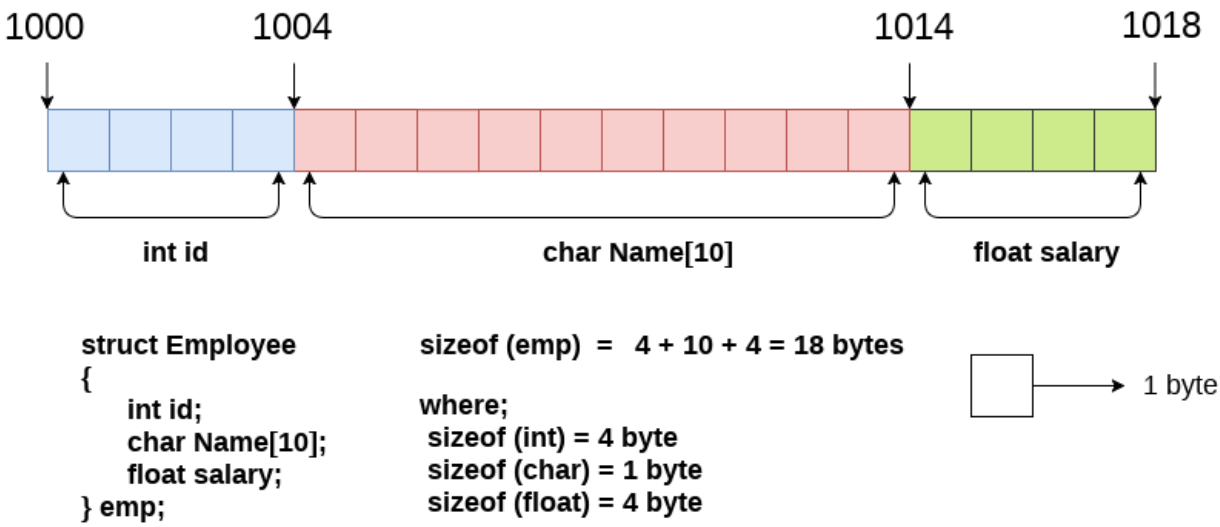
Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. Structures ca; simulate the use of classes and templates as it can store various information
The ,struct keyword is used to define the structure. Let's see the syntax to define the structure in c.

```
1.  struct structure_name
2.  {
3.      data_type member1;
4.      data_type member2;
5.      .
6.      .
7.      data_type memeberN;
8.  };
```

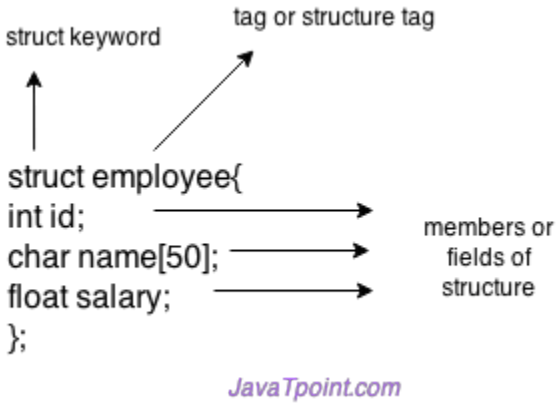
Let's see the example to define a structure for an entity employee in c.

```
1.  struct employee
2.  {   int id;
3.      char name[20];
4.      float salary;
5.  };
```

The following image shows the memory allocation of the structure employee that is defined in the above example.



Here, struct is the keyword; employee is the name of the structure; id, name, and salary are the members or fields of the structure. Let's understand it by the diagram given below:



Declaring structure variable

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

1st way:

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

```
1. struct employee
2. {  int id;
3.     char name[50];
4.     float salary;
5. };
```

Now write given code inside the main() function.

```
1. struct employee e1, e2;
```

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 can be treated in the same way as the objects in C++ and Java.

2nd way:

Let's see another way to declare variable at the time of defining the structure.

```
1. struct employee
2. {   int id;
3.     char name[50];
4.     float salary;
5. }e1,e2;
```

Which approach is good

If number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.

If no. of variables are fixed, use 2nd approach. It saves your code to declare a variable in main() function.

Accessing members of the structure

There are two ways to access structure members:

1. By . (member or dot operator)
2. By -> (structure pointer operator)

Let's see the code to access the id member of p1 variable by. (member) operator.

```
1. p1.id
```

C Structure example

Let's see a simple example of structure in C language.

```
1. #include<stdio.h>
2. #include <string.h>
3. struct employee
4. {   int id;
5.     char name[50];
6. }e1;  //declaring e1 variable for structure
7. int main( )
8. {
9.    //store first employee information
10.   e1.id=101;
11.   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
12.   //printing first employee information
13.   printf( "employee 1 id : %d\n", e1.id);
14.   printf( "employee 1 name : %s\n", e1.name);
15. return 0;
16. }
```

Output:
employee 1 id : 101
employee 1 name : Sonoo Jaiswal

C Array of Structures

Why use an array of structures?

Consider a case, where we need to store the data of 5 students. We can store it by using the structure as given below.

```
1. #include<stdio.h>
2. struct student
3. {
4.     char name[20];
5.     int id;
6.     float marks;
7. };
8. void main()
9. {
10.    struct student s1,s2,s3;
11.    int dummy;
12.    printf("Enter the name, id, and marks of student 1 ");
13.    scanf("%s %d %f",s1.name,&s1.id,&s1.marks);
14.    scanf("%c",&dummy);
15.    printf("Enter the name, id, and marks of student 2 ");
16.    scanf("%s %d %f",s2.name,&s2.id,&s2.marks);
17.    scanf("%c",&dummy);
18.    printf("Enter the name, id, and marks of student 3 ");
19.    scanf("%s %d %f",s3.name,&s3.id,&s3.marks);
20.    scanf("%c",&dummy);
21.    printf("Printing the details....\n");
22.    printf("%s %d %f\n",s1.name,s1.id,s1.marks);
23.    printf("%s %d %f\n",s2.name,s2.id,s2.marks);
24.    printf("%s %d %f\n",s3.name,s3.id,s3.marks);
25. }
```

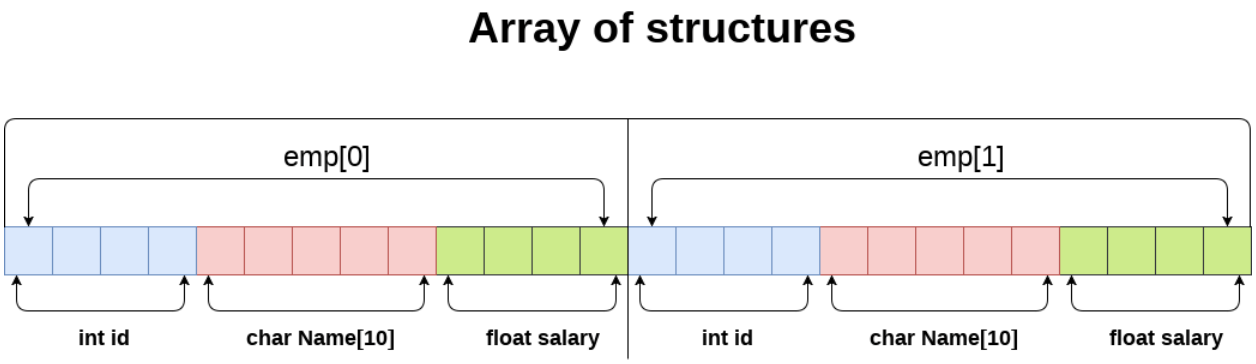
Output
Enter the name, id, and marks of student 1 James 90 90
Enter the name, id, and marks of student 2 Adoms 90 90
Enter the name, id, and marks of student 3 Nick 90 90
Printing the details....
James 90 90.000000
Adoms 90 90.000000

Nick 90 90.000000

In the above program, we have stored data of 3 students in the structure. However, the complexity of the program will be increased if there are 20 students. In that case, we will have to declare 20 different structure variables and store them one by one. This will always be tough since we will have to declare a variable every time we add a student. Remembering the name of all the variables is also a very tricky task. However, c enables us to declare an array of structures by using which, we can avoid declaring the different structure variables; instead we can make a collection containing all the structures that store the information of different entities.

Array of Structures in C

An array of structres in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.




```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

sizeof (emp) = 4 + 5 + 4 = 13 bytes

sizeof (emp[2]) = 26 bytes

Let's see an example of an array of structures that stores information of 5 students and prints it.

```
1. #include<stdio.h>
2. #include <string.h>
3. struct student{
4. int rollno;
5. char name[10];
6. };
7. int main(){
8. int i;
9. struct student st[5];
10. printf("Enter Records of 5 students");
11. for(i=0;i<5;i++){
```

```
12. printf("\nEnter Rollno:");
13. scanf("%d",&st[i].rollno);
14. printf("\nEnter Name:");
15. scanf("%s",&st[i].name);
16. }
17. printf("\nStudent Information List:");
18. for(i=0;i<5;i++){
19. printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
20. }
21.    return 0;
22. }
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz

Student Information List:
Rollno:1, Name:Sonoo
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
Rollno:5, Name:Sarfraz
```

File Handling in C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file

- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

---

Opening File: fopen()

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

1. FILE *fopen(const char * filename, const char * mode);

The fopen() function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like "c://some_folder/some_file.ext".
- The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

The fopen function works in the following way.

- Firstly, It searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

Consider the following example which opens a file in write mode.

```
1. #include<stdio.h>
2. void main( )
3. {
4. FILE *fp ;
5. char ch ;
6. fp = fopen("file_handle.c","r") ;
7. while ( 1 )
8. {
9. ch = fgetc ( fp ) ;
10. if ( ch == EOF )
11. break ;
12. printf("%c",ch) ;
13. }
14. fclose (fp ) ;
15. }
```

Output

The content of the file will be printed.

```
#include;
void main( )
{
FILE *fp; // file pointer
char ch;
fp = fopen("file_handle.c","r");
while ( 1 )
{
ch = fgetc ( fp ); //Each character of the file is read and stored in the character file.
if ( ch == EOF )
break;
printf("%c",ch);
}
fclose (fp );
}
```

Closing File: fclose()

The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

```
int fclose( FILE *fp );
```

Command Line Arguments in C

The arguments passed from command line are called command line arguments. These arguments are handled by main() function.
To support command line argument, you need to change the structure of main() function as given below.

```
1. int main(int argc, char *argv[] )
```

Here, argc counts the number of arguments. It counts the file name as the first argument.
The argv[] contains the total number of arguments. The first argument is the file name always.

Example

Let's see the example of command line arguments where we are passing one argument with file name.

```
1.  #include <stdio.h>
2.  void main(int argc, char *argv[] )  {
3.
4.     printf("Program name is: %s\n", argv[0]);
5.
6.     if(argc < 2){
7.        printf("No argument passed through command line.\n");
8.     }
9.     else{
10.       printf("First argument is: %s\n", argv[1]);
11.    }
12. }
```

Run this program as follows in Linux:

```
1.  ./program hello
```

Run this program as follows in Windows from command line:

```
1.  program.exe hello
```

Output:
Program name is: program
First argument is: hello

If you pass many arguments, it will print only one.

```
1.  ./program hello c how r u
```

Output:
Program name is: program
First argument is: hello

But if you pass many arguments within double quote, all arguments will be treated as a single argument only.

```
1.  ./program "hello c how r u"
```

Output:

Program name is: program
First argument is: hello c how r u

You can write your program to print all the arguments. In this program, we are printing only argv[1], that is why it is printing only one argument.