**CHAPTER-7 ADVANCE FEATURES IN C**

**C Functions**

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as procedure or subroutine in other programming languages.

**Advantage of functions in C**

There are the following advantages of C functions.

* By using functions, we can avoid rewriting same logic/code again and again in a program.
* We can call C functions any number of times in a program and from any place in a program.
* We can track a large C program easily when it is divided into multiple functions.
* Reusability is the main achievement of C functions.
* However, Function calling is always a overhead in a C program.

**Function Aspects**

There are three aspects of a C function.

* **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
* **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
* **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

|  |  |  |
| --- | --- | --- |
| **SN** | **C function aspects** | **Syntax** |
| 1 | Function declaration | return\_type function\_name (argument list); |
| 2 | Function call | function\_name (argument\_list) |
| 3 | Function definition | return\_type function\_name (argument list) {function body;} |

Function declaration consists of 4 parts.

* returntype
* function name
* parameter list
* terminating semicolon

Returntype

When a function is declared to perform some sort of calculation or any operation and is expected to provide with some result at the end, in such cases, a return statement is added at the end of function body. Return type specifies the type of value(int, float, char, double) that function is expected to return to the program which called the function.

Note: In case your function doesn't return any value, the return type would be void.

#### Function Name

Function name is an [identifier](https://www.studytonight.com/keywords-and-identifier.php) and it specifies the name of the function. The function name is any valid C identifier and therefore must follow the same naming rules like other variables in C language.

**Parameter** list

The parameter list declares the type and number of arguments that the function expects when it is called. Also, the parameters in the parameter list receives the argument values when the function is called. They are often referred as formal parameters.

The syntax of creating function in c language is given below:

return\_type function\_name(data\_type parameter...){

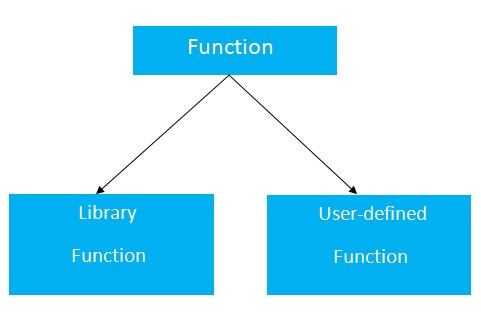
//code to be executed

}

**Types of Functions**

There are two types of functions in C programming:

1. **Library Functions**: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



**Return Value**

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

**Example without return value:**

**void** hello(){

printf("hello c");

}

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

**Example with return value:**

**int** get(){

**return** 10;

}

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

**float** get(){

**return** 10.2;

}

Now, you need to call the function, to get the value of the function.

**Different aspects of function calling**

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

* function without arguments and without return value
* function without arguments and with return value
* function with arguments and without return value
* function with arguments and with return value

### Example for Function without argument and return value

#### Example 1

1. #include<stdio.h>
2. **void** printName();
3. **void** main ()
4. {
5. printf("Hello ");
6. printName();
7. }
8. **void** printName()
9. {
10. printf("oops");
11. }

**Output**

Hello oops

#### Example 2

1. #include<stdio.h>
2. **void** sum();
3. **void** main()
4. {
5. printf("\nGoing to calculate the sum of two numbers:");
6. sum();
7. }
8. **void** sum()
9. {
10. **int** a,b;
11. printf("\nEnter two numbers");
12. scanf("%d %d",&a,&b);
13. printf("The sum is %d",a+b);
14. }

**Output**

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

### Example for Function without argument and with return value

#### Example 1

1. #include<stdio.h>
2. **int** sum();
3. **void** main()
4. {
5. **int** result;
6. printf("\nGoing to calculate the sum of two numbers:");
7. result = sum();
8. printf("%d",result);
9. }
10. **int** sum()
11. {
12. **int** a,b;
13. printf("\nEnter two numbers");
14. scanf("%d %d",&a,&b);
15. **return** a+b;
16. }

**Output**

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

#### Example 2: program to calculate the area of the square

1. #include<stdio.h>
2. **int** sum();
3. **void** main()
4. {
5. printf("Going to calculate the area of the square\n");
6. **float** area = square();
7. printf("The area of the square: %f\n",area);
8. }
9. **int** square()
10. {
11. **float** side;
12. printf("Enter the length of the side in meters: ");
13. scanf("%f",&side);
14. **return** side \* side;
15. }

**Output**

Going to calculate the area of the square

Enter the length of the side in meters: 10

The area of the square: 100.000000

**Example for Function with argument and without return value**

#### Example 1

1. #include<stdio.h>
2. **void** sum(**int**, **int**);
3. **void** main()
4. {
5. **int** a,b,result;
6. printf("\nGoing to calculate the sum of two numbers:");
7. printf("\nEnter two numbers:");
8. scanf("%d %d",&a,&b);
9. sum(a,b);
10. }
11. **void** sum(**int** a, **int** b)
12. {
13. printf("\nThe sum is %d",a+b);
14. }

**Output**

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

#### Example 2: program to calculate the average of five numbers.

1. #include<stdio.h>
2. **void** average(**int**, **int**, **int**, **int**, **int**);
3. **void** main()
4. {
5. **int** a,b,c,d,e;
6. printf("\nGoing to calculate the average of five numbers:");
7. printf("\nEnter five numbers:");
8. scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
9. average(a,b,c,d,e);
10. }
11. **void** average(**int** a, **int** b, **int** c, **int** d, **int** e)
12. {
13. **float** avg;
14. avg = (a+b+c+d+e)/5;
15. printf("The average of given five numbers : %f",avg);
16. }

**Output**

Going to calculate the average of five numbers:

Enter five numbers:10

20

30

40

50

The average of given five numbers : 30.000000

### Example for Function with argument and with return value

#### Example 1

1. #include<stdio.h>
2. **int** sum(**int**, **int**);
3. **void** main()
4. {
5. **int** a,b,result;
6. printf("\nGoing to calculate the sum of two numbers:");
7. printf("\nEnter two numbers:");
8. scanf("%d %d",&a,&b);
9. result = sum(a,b);
10. printf("\nThe sum is : %d",result);
11. }
12. **int** sum(**int** a, **int** b)
13. {
14. **return** a+b;
15. }

**Output**

Going to calculate the sum of two numbers:

Enter two numbers:10

20

The sum is : 30

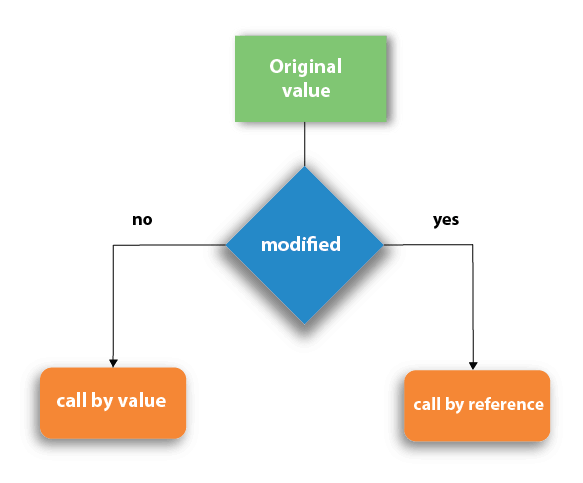
#### Example 2: Program to check whether a number is even or odd

## C Library Functions

## Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension ****.h****. We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as printf/scanf we need to include stdio.h in our program which is a header file that contains all the library functions regarding standard input/output.

# Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., call by value and call by reference.



## Call by value in C

* In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
* In call by value method, we can not modify the value of the actual parameter by the formal parameter.
* In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
* The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

**Concept of call by value in c language by the example given below:**

1. #include<stdio.h>
2. **void** change(**int** num) {
3. printf("Before adding value inside function num=%d \n",num);
4. num=num+100;
5. printf("After adding value inside function num=%d \n", num);
6. }
7. **int** main() {
8. **int** x=100;
9. printf("Before function call x=%d \n", x);
10. change(x);//passing value in function
11. printf("After function call x=%d \n", x);
12. **return** 0;
13. }

#### Output

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=100

#### Example: Swapping the values of the two variables

1. #include <stdio.h>
2. **void** swap(**int** , **int**); //prototype of the function
3. **int** main()
4. {
5. **int** a = 10;
6. **int** b = 20;
7. printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
8. swap(a,b);
9. printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the formal parameters in call by value, a = 10, b = 20
10. }
11. **void** swap (**int** a, **int** b)
12. {
13. **int** temp;
14. temp = a;
15. a=b;
16. b=temp;
17. printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b = 10
18. }

#### Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 10, b = 20

## Call by reference in C

* In call by reference, the address of the variable is passed into the function call as the actual parameter.
* The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
* In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

**Example for the call by reference.**

1. #include<stdio.h>
2. **void** change(**int** \*num) {
3. printf("Before adding value inside function num=%d \n",\*num);
4. (\*num) += 100;
5. printf("After adding value inside function num=%d \n", \*num);
6. }
7. **int** main() {
8. **int** x=100;
9. printf("Before function call x=%d \n", x);
10. change(&x);//passing reference in function
11. printf("After function call x=%d \n", x);
12. **return** 0;
13. }

#### Output

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=200

#### Call by reference Example: Swapping the values of the two variables

1. #include <stdio.h>
2. **void** swap(**int** \*, **int** \*); //prototype of the function
3. **int** main()
4. {
5. **int** a = 10;
6. **int** b = 20;
7. printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
8. swap(&a,&b);
9. printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call by reference, a = 10, b = 20
10. }
11. **void** swap (**int** \*a, **int** \*b)
12. {
13. **int** temp;
14. temp = \*a;
15. \*a=\*b;
16. \*b=temp;
17. printf("After swapping values in function a = %d, b = %d\n",\*a,\*b); // Formal parameters, a = 20, b = 10
18. }

#### Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 20, b = 10

## Difference between call by value and call by reference in c

|  |  |  |
| --- | --- | --- |
| **No.** | **Call by value** | **Call by reference** |
| 1 | A copy of the value is passed into the function | An address of value is passed into the function |
| 2 | Changes made inside the function are limited to the function only. The values of the actual parameters do not change by changing the formal parameters. | Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters. |
| 3 | Actual and formal arguments are created at the different memory location | Actual and formal arguments are created at the same memory location |

**SCOPE OF THE VARIABLE**

# Global variable

## Global variables can be accessed at any point throughout the program, and can be used in any function. There is single copy of the global variable is available.Global variables are defined outside of all the functions, usually on top of the program.

**Local variable**

Variable declared inside a function or block of code are called local variables. They can be accessed only inside that function or block of code. Local variables are not available to outside function.

**Example Program:**

## #include <stdio.h>/\* global variable declaration \*/

int sum;

int main ()

{

/\* local variable declaration \*/ int a, b;

/\* actual initialization \*/ a = 10;

b = 20;

sum = a + b;

printf ("value of a = %d, b = %d and sum of (a and b) is = %d\n", a, b, sum return 0;

}

**Note :-** A program can have same name for local and global variables but local variable overrides the value.

**C storage classes**

In C Programming Language a “Storage Class” refers to the scope or visibility and the life time of the C Variable. Scope of the C Variable defines the availability of the variable in block of the C Program, and by life time of variable means how long variable will persist in the program.

**Functions of storage class**

It tell the location of the C Variable.

Sets the initial or default value of C Variable. It defines the scope of the C Variable.

It defines the life time of C Variable.

**Types of storage class**

C Programming Language Supports following four type of storage classes:

auto register static extern

**auto-storage class**

It is the default storage class for local variables. Variables with auto storage class are declared at the beginning of a code block, and memory is allocated automatically as the program execution enters to a code block and frees up automatically on exiting from the code block. The scope of automatic variables is local to the block where the declared and are not accessible directly in the other block.

**Syntax:**

auto [data\_type] [variable\_name];

**Example :**

auto int a;

**register-storage class**

C Variables with register storage class are same as local variables to the code block but the are stored in CPU register instead of computer memory. Hence it enables the quick access of that variable. Maximum size of the variable is equal to register size and dependent upon the

register size

**Syntax:**

register [data\_type] [variable\_name];

**Example:**

register int a;

**static-storage class**

It is default storage class for global variables.Static storage class can be used only if we want the value of a variable to persist between different function calls. C Variable declared with static storage class will keep its value retained for different function calls.

**Syntax:**

static [data\_type] [variable\_name];

**Example:**

static int a;

**extern-storage class**

C Variable declared with extern storage class is said to be “Global Variables”, it means variables is accessible throughout the program till the end of program execution. External variables are declared outside the functions and can be invoked from and anywhere in a program.External variables can be accessed very fast as compared to any other storage classes.

**Syntax:**

extern [data\_type] [variable\_name];

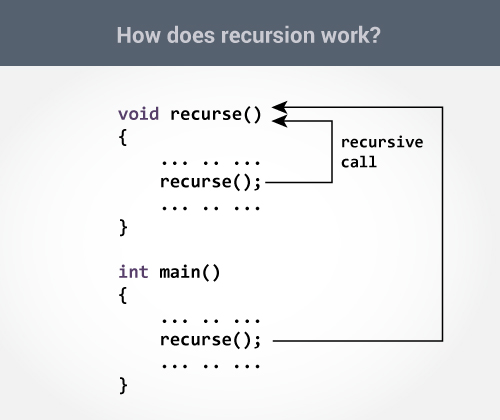
**Example:**

extern int a;

**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.

### How recursion works?

void recurse()

{

... .. ...

recurse();

... .. ...

}

int main()

{

... .. ...

recurse();

……..

}

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 –

Recursion is used to calculate the factorial of a number

1. #include <stdio.h>
2. **int** fact (**int**);
3. **int** main()
4. {
5. **int** n,f;
6. printf("Enter the number whose factorial you want to calculate?");
7. scanf("%d",&n);
8. f = fact(n);
9. printf("factorial = %d",f);
10. }
11. **int** fact(**int** n)
12. {
13. **if** (n==0)
14. {
15. **return** 0;
16. }
17. **else** **if** ( n == 1)
18. {
19. **return** 1;
20. }
21. **else**
22. {
23. **return** n\*fact(n-1);
24. }
25. }

#### Output

Enter the number whose factorial you want to calculate?5

factorial = 120

**Fibonacci Series**

The following example generates the Fibonacci series for a given number using a recursive function −

#include <stdio.h>

int fibonacci(int i) {

if(i == 0) {

return 0;

}

if(i == 1) {

return 1;

}

return fibonacci(i-1) + fibonacci(i-2);

}

int main() {

int i;

for (i = 0; i < 10; i++) {

printf("%d\t\n", fibonacci(i));

}

return 0;

}

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

0

1

1

2

3

5

8

13

21

34

Some more examples:

1. #include<stdio.h>
2. **int** fibonacci(**int**);
3. **void** main ()
4. {
5. **int** n,f;
6. printf("Enter the value of n?");
7. scanf("%d",&n);
8. f = fibonacci(n);
9. printf("%d",f);
10. }
11. **int** fibonacci (**int** n)
12. {
13. **if** (n==0)
14. {
15. **return** 0;
16. }
17. **else** **if** (n == 1)
18. {
19. **return** 1;
20. }
21. **else**
22. {
23. **return** fibonacci(n-1)+fibonacci(n-2);
24. }
25. }

#### Output

Enter the value of n? 12

144

### What are the different types of Recursion in C?

#### 1. Primitive Recursion

It is the types of recursion that can be converted into a loop.

We have already seen the Fibonacci series example which can be programmed with recursion as well as with loop.

#### 2. Tail Recursion

It is a primitive recursion in which the recursive call is present as the last thing in the function.

In the above Fibonacci example, the recursive function is executed as the last statement of the ‘fibo’ function.

#### 3. Single Recursion

It is the types of recursion when there is only one recursive call in the function.

#### 4. Multiple Recursion

As by its name, it is the types of recursion when there are multiple recursive calls in the function.

It is just the opposite of a single recursion. Recursion can be either single or multiple type.

#### 5. Mutual Recursion or Indirect Recursion)

There are two or more functions involved in this type of recursion.

In this type of recursion, a function calls another function, which eventually calls the original function.

#### 6. General Recursion

If there is a function which cannot be defined without recursion, is called as general recursion.

It is the opposite of primitive type recursion.

These are the different types of recursion in C.

**More about it:**

**Types of Recursion**  
Recursive functions can be classified  on the basis of :  
a.) If the functions calls itself directly or indirectly. – Direct / Indirect  
b.) If an operation is pending at each recursive call. – Tail Recursive/ Not  
c.) based on the structure of function calling pattern. – Linear / Tree  
   
a.) **Direct Recursion:**

If a function explicitly calls itself it is called directly recursive.

When the method invokes itself it is direct.

Example:

int testfunc( int num)  
{  
if (num == 0)  
return 0;  
else  
return (testfunc(num-1));  
}  
Here, the function ‘testfunc’ calls itself for all positive values of num.  
   
b.) Indirect Recursion:

This occurs when the function invokes other method which again causes the original function to be called again.

If a method ‘X’ , calls method ‘Y’, which calls method ‘Z’ which again leads to ‘X’ being invoked is called indirect recursive or mutually recursive as well.

Example:

int testfunc1( int num)  
{  
if (num == 0)  
return 0;  
else  
return (testfunc2(num-1));  
}  
int testfunc2(int num2)  
{ return testfunc1( num2-1); }

Tail / Bottom Recursion:

A function is said to be tail recursive, if no operations are pending when the recursive function returns to its caller.

Such functions, immediately return the return value from the calling function.

It is an efficient method as compared to others, as the stack space required is less and even compute overhead will get reduced.

Recollect the previously discussed example, factorial of a number. We had written it in non tail recursive way, as after call operation is still pending.

int fact (int n)

{  
if (n==1)  
return 1;  
else  
return (n\*fact(n-1));  
}  
In order to make it tail recursive, information about pending tasks has to be tracked.  
   
Tail recursion:

int fact (int n)

{return (n\*fact2(n-1));}  
   
int fact2(int n, int result)  
{  
if (n==1)  
return result;  
return fact2(n-1, n\*result);}  
   
If you observe, the ‘fact2’ has similar syntax to the original fact. Now, ‘fact’ in tail recursion case does not have pending calculations / operations to perform on return from recursive function calls.  
The value computed by fact2 is simply returned. Thus, amount of space required on stack reduces considerably . ( just for value of n and result , space required.)

Linear and Tree Recursion:

Depending on the structure the recursive function calls take, or grows it can be either linear or non linear.  
It is linearly recursive when, the pending operations do not involve another recursive call to the function. Our Factorial recursive function is linearly recursive as it only involves multiplying the returned values and no further calls to function.  
Tree recursion is when, pending operations involve another recursive call to function.  
For example –  Fibonacci series ; the pending operations have recursive call to the fib() recursive function to compute the results.

**When not to use recursion ?**  
Essentially, any problem that can be solved by recursion can be also solved by iteration. Although for certain applications like Tower of Hanoi, certain artificial intelligence problems etc recursion is used since it defines the problem naturally, still there are certain times when recursion should not be used.

Iterations are faster than their recursive counterparts. So, for speed we would normally use iteration.

 If the stack limit is constraining then we will prefer iteration over recursion.

Some problems are naturally programmed recursively, here recursion would suit better.

The overhead involved in recursion in terms of time and space consumed both discourages to use recursion often.

**Array in c**

An array is a series of elements of the similar type placed in contiguous memory locations that can be accessed individually by adding an index or subscript to a unique identifier. An array is made up of a key and a value pair, and the key points to the value. Arrays may be of any variable type.

**Types of an Array**

One / Single Dimensional Array

Multi dimensional array ( Example : Two dimensional array )

**Single / One Dimensional Array :**

A one-dimensional array is a structured collection of array elements that can be accessed individually by specifying the position of a element with a single index value.

**Declaring Single Dimensional Array**

<data-type> <array\_name> [size];

//total size = length of array \* size of data type

**Example:**

int a[100];

**Initializing an Array**

int arr[5] = {10,20,30,40,50};

**Accessing Array’s Elements**

An array elements can be accessed easily by index or subscript like array\_name[index]. Indexing for an array starts from zero (0) to the last element of an array is array\_name[size- 1] where size is the number of elements of the array.

**Example program for one/single dimension C array:**

#include<stdio.h> int main()

{

int i;

int arr[5] = {10,20,30,40,50};

// declaring and Initializing array

for (i=0;i<5;i++)

{

// Accessing each variable

printf("value of arr[%d] is %d \n", i, arr[i]);

}

}

**Output:**

value of arr[0] is 10 value of arr[1] is 20 value of arr[2] is 30 value of arr[3] is 40 value of arr[4] is 50

**Multi dimensional array ( Example : Two dimensional array )**

Multidimensional arrays is nothing just an array of arrays. We can declare a multidimensional array, as below:

<data-type> <array\_name>[size][size2][size3]…

**Example program for two/multi dimension C array:**

#include<stdio.h> int main()

{

int i,j;

// declaring and Initializing array int arr[2][2] = {10,20,30,40};

for (i=0;i<2;i++)

{

for (j=0;j<2;j++)

{

// Accessing variables

printf("value of arr[%d][%d] is %d\n",i,j,arr[i][j]);

}

}

}

**Output:**

value of arr[0][0] is 10

value of arr[0][1] is 20

value of arr[1][0] is 30

value of arr[1][1] is 40

C Programming Strings

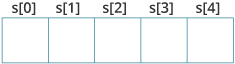
In C programming, a string is a sequence of characters terminated with a null character \0. For example:

1. char c[] = "c string";
2. When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.

Memory diagram of strings in C programming

**How to declare a string?**

Here's how you can declare strings:

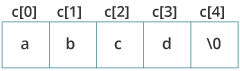
1. char s[5];
2. 

Here, we have declared a string of 5 characters.

**How to initialize strings?**

You can initialize strings in a number of ways.

1. char c[] = "abcd";
2. char c[50] = "abcd";
3. char c[] = {'a', 'b', 'c', 'd', '\0'};
4. char c[5] = {'a', 'b', 'c', 'd', '\0'};



Let's take another example:

1. char c[5] = "abcde";
2. Here, we are trying to assign 6 characters (the last character is '\0') to a char array having 5 characters. This is bad and you should never do this.

**Read String from the user**

You can use the scanf() function to read a string.

The scanf() function reads the sequence of characters until it encounters [whitespace](https://stackoverflow.com/questions/30033582/what-is-the-symbol-for-whitespace-in-c) (space, newline, tab etc.).

**Example 1: scanf() to read a string**

1. #include <stdio.h>
2. int main()
3. {
4. char name[20];
5. printf("Enter name: ");
6. scanf("%s", name);
7. printf("Your name is %s.", name);
8. return 0;
9. }

**Output**

Enter name: Dennis Ritchie

Your name is Dennis.

Even though Dennis Ritchie was entered in the above program, only "Ritchie" was stored in the name string. It's because there was a space after Dennis.

**How to read a line of text?**

You can use the fgets() function to read a line of string. And, you can use puts() to display the string.

**Example 2: fgets() and puts()**

1. #include <stdio.h>
2. int main()
3. {
4. char name[30];
5. printf("Enter name: ");
6. fgets(name, sizeof(name), stdin); // read string
7. printf("Name: ");
8. puts(name); // display string
9. return 0;
10. }

**Output**

Enter name: Tom Hanks

Name: Tom Hanks

Here, we have used fgets() function to read a string from the user.

fgets(name, sizeof(name), stdlin); // read string

The sizeof(name) results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the name string.

To print the string, we have used puts(name);.

**Note:** The gets() function can also be to take input from the user. However, it is removed from the C standard.  
  
It's because gets() allows you to input any length of characters. Hence, there might be a buffer overflow.

**Passing Strings to Functions**

Strings can be passed to a function in a similar way as arrays. Learn more about [passing arrays to a function](https://www.programiz.com/c-programming/c-arrays-functions).

**Example 3: Passing string to a Function**

1. #include <stdio.h>
2. void displayString(char str[]);
3. int main()
4. {
5. char str[50];
6. printf("Enter string: ");
7. fgets(str, sizeof(str), stdin);
8. displayString(str); // Passing string to a function.
9. return 0;
10. }
11. void displayString(char str[])
12. {
13. printf("String Output: ");
14. puts(str);
15. }

**Strings and Pointers**

Similar like arrays, string names are "decayed" to pointers. Hence, you can use pointers to manipulate elements of the string. We recommended you to check [C Arrays and Pointers](https://www.programiz.com/c-programming/c-pointers-arrays) before you check this example.

**Example 4: Strings and Pointers**

1. #include <stdio.h>
2. int main(void) {
3. char name[] = "Harry Potter";
4. printf("%c", \*name); // Output: H
5. printf("%c", \*(name+1)); // Output: a
6. printf("%c", \*(name+7)); // Output: o
7. char \*namePtr;
8. namePtr = name;
9. printf("%c", \*namePtr); // Output: H
10. printf("%c", \*(namePtr+1)); // Output: a
11. printf("%c", \*(namePtr+7)); // Output: o
12. }

**Commonly Used String Functions**

* [**strlen()** - calculates the length of a string](https://www.programiz.com/c-programming/library-function/string.h/strlen)
* [**strcpy()** - copies a string to another](https://www.programiz.com/c-programming/library-function/string.h/strcpy)
* [**strcmp()** - compares two strings](https://www.programiz.com/c-programming/library-function/string.h/strcmp)
* [**strcat()** - concatenates two strings](https://www.programiz.com/c-programming/library-function/string.h/strcat)

**C pointer**

Pointers is one of the most powerful feature available in C Programming Language . Pointer is a special type of variable that refers to the address of other data object or variable.Variable can be of any data type i.e int, float, char, double, short etc.Pointer is used for dynamic memory allocation.

**Syntax to declare pointer variable:**

<data\_type> \*pointer\_name;

data\_type specifies the type of pointer, then asterisk (\*) followed by the pointer name.

**Example:**

**Example Program to demonstrate pointer**

int \*ptr;

#include <stdio.h>

int main()

{

int \*ptr; int var1; var1 = 20;

/\* address of var1 is assigned to ptr\*/

ptr = &var1;

/\* display var1's value using ptr variable \*/

printf("%d", \*ptr);

return 0;

}

**Output:**

20

Pointer Expressions

We can use pointer variables in expression.

**Example:**  
int x = 10, y = 20, z;  
int \*ptr1 = &x;  
int \*ptr2 = &y;  
z = \*ptr1 \* \*ptr2 ;  
Will assign 200 to variable z.

We can perform addition and subtraction of integer constant from pointer variable.

**Example:**  
ptr1 = ptr1 + 2;  
ptr2 = ptr2 – 2;

We can not perform addition, multiplication and division operations on two pointer variables.

For Example:  
ptr1 + ptr2 is not valid

However we can subtract one pointer variable from another pointer variable.  
We can use increment and decrement operator along with pointer variable to increment or decrement the address contained in pointer variable.

**Example:**  
ptr1++;  
ptr2--;

We can use relational operators to compare pointer variables if both pointer variable points to the variables of same data type.

# Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

* Increment
* Decrement
* Addition
* Subtraction
* Comparison

## Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

1. new\_address= current\_address + i \* size\_of(data type)

Where i is the number by which the pointer get increased.

### 32-bit

For 32-bit int variable, it will be incremented by 2 bytes.

### 64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

Let's see the example of incrementing pointer variable on 64-bit architecture.

1. #include<stdio.h>
2. **int** main(){
3. **int** number=50;
4. **int** \*p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+1;
8. printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incremented by 4 bytes.
9. **return** 0;
10. }

**Output**

Address of p variable is 3214864300

After increment: Address of p variable is 3214864304

### Traversing an array by using pointer

1. #include<stdio.h>
2. **void** main ()
3. {
4. **int** arr[5] = {1, 2, 3, 4, 5};
5. **int** \*p = arr;
6. **int** i;
7. printf("printing array elements...\n");
8. **for**(i = 0; i< 5; i++)
9. {
10. printf("%d  ",\*(p+i));
11. }
12. }

**Output**

printing array elements...

1 2 3 4 5

## Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

1. new\_address= current\_address - i \* size\_of(data type)

### 32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

### 64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

1. #include <stdio.h>
2. **void** main(){
3. **int** number=50;
4. **int** \*p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p-1;
8. printf("After decrement: Address of p variable is %u \n",p); // P will now point to the immidiate previous location.
9. }

**Output**

Address of p variable is 3214864300

After decrement: Address of p variable is 3214864296

## C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

1. new\_address= current\_address + (number \* size\_of(data type))

### 32-bit

For 32-bit int variable, it will add 2 \* number.

### 64-bit

For 64-bit int variable, it will add 4 \* number.

Let's see the example of adding value to pointer variable on 64-bit architecture.

1. #include<stdio.h>
2. **int** main(){
3. **int** number=50;
4. **int** \*p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+3;   //adding 3 to pointer variable
8. printf("After adding 3: Address of p variable is %u \n",p);
9. **return** 0;
10. }

**Output**

Address of p variable is 3214864300

After adding 3: Address of p variable is 3214864312

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., 4\*3=12 increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., 2\*3=6. As integer value occupies 2-byte memory in 32-bit OS.

## C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

1. new\_address= current\_address - (number \* size\_of(data type))

### 32-bit

For 32-bit int variable, it will subtract 2 \* number.

### 64-bit

For 64-bit int variable, it will subtract 4 \* number.

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

1. #include<stdio.h>
2. **int** main(){
3. **int** number=50;
4. **int** \*p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p-3; //subtracting 3 from pointer variable
8. printf("After subtracting 3: Address of p variable is %u \n",p);
9. **return** 0;
10. }

**Output**

Address of p variable is 3214864300

After subtracting 3: Address of p variable is 3214864288

You can see after subtracting 3 from the pointer variable, it is 12 (4\*3) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

1. Address2 - Address1 = (Subtraction of two addresses)/size of data type which pointer points

Consider the following example to subtract one pointer from an another.

1. #include<stdio.h>
2. **void** main ()
3. {
4. **int** i = 100;
5. **int** \*p = &i;
6. **int** \*temp;
7. temp = p;
8. p = p + 3;
9. printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);
10. }

**Output**

Pointer Subtraction: 1030585080 - 1030585068 = 3

## Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

* Address + Address = illegal
* Address \* Address = illegal
* Address % Address = illegal
* Address / Address = illegal
* Address & Address = illegal
* Address ^ Address = illegal
* Address | Address = illegal
* ~Address = illegal

## Pointer to function in C

As we discussed in the previous chapter, a pointer can point to a function in C. However, the declaration of the pointer variable must be the same as the function. Consider the following example to make a pointer pointing to the function.

1. #include<stdio.h>
2. **int** addition ();
3. **int** main ()
4. {
5. **int** result;
6. **int** (\*ptr)();
7. ptr = &addition;
8. result = (\*ptr)();
9. printf("The sum is %d",result);
10. }
11. **int** addition()
12. {
13. **int** a, b;
14. printf("Enter two numbers?");
15. scanf("%d %d",&a,&b);
16. **return** a+b;
17. }

**Output**

Enter two numbers?10 15

The sum is 25

## Pointer to Array of functions in C

To understand the concept of an array of functions, we must understand the array of function. Basically, an array of the function is an array which contains the addresses of functions. In other words, the pointer to an array of functions is a pointer pointing to an array which contains the pointers to the functions. Consider the following example.

1. #include<stdio.h>
2. **int** show();
3. **int** showadd(**int**);
4. **int** (\*arr[3])();
5. **int** (\*(\*ptr)[3])();
7. **int** main ()
8. {
9. **int** result1;
10. arr[0] = show;
11. arr[1] = showadd;
12. ptr = &arr;
13. result1 = (\*\*ptr)();
14. printf("printing the value returned by show : %d",result1);
15. (\*(\*ptr+1))(result1);
16. }
17. **int** show()
18. {
19. **int** a = 65;
20. **return** a++;
21. }
22. **int** showadd(**int** b)
23. {
24. printf("\nAdding 90 to the value returned by show: %d",b+90);
25. }

**Output**

printing the value returned by show : 65

Adding 90 to the value returned by show: 155

**Structure in C**

In C Programing Language we can use arrays when we want hold multiple element of the homogeneous data type in single variable, but what if we want to hold heterogeneous data type element, using structure you can wrap one or more variables that may be in different data types into one.

**Syntax:** struct struct\_name{ structure\_member };

To define a structure, you use the struct keyword followed by structure name. The variables which are declared inside the structure are called as ‘members of structure’.

**Example:**

struct student

{

int id;

char name[20]; float percentage;

};

The student structure contains id as an positive integer, name as a string, percentage code as a float.

**Declaring structure:**

The above example only defines an student structure without creating any structure variable. To declare a structure variable, you can do it in two ways:

**Declaring structure with variable together:**

struct struct\_name {

structure\_member;

...

}

instance\_1,instance\_2 instance\_n;

**Declaring the structure variable after you define the structure:**

struct struct\_name instance\_1,instance\_2 instance\_n;

**Note :**– When declaring structure following points need to be keep in mind –

Structure is always terminated with semicolon (;).

Structure name as struct\_name can be later used to declare structure variables of its type in a program.

# Accessing structure members

## Member operator ‘.’ is used to access the individual structure members. It is also known as ‘dot operator’ or ‘period operator’.

**Syntax:**

structure\_var.member;

addr[100];

**Example Program:**

#include <stdio.h>

#include<conio.h>

struct student

{

char name[20];

char addr[100];

} student;

void main()

{

clrscr();

printf("\n Enter Student Name : ");

gets (student.name);

printf("\n Enter student address: ");

gets(student.addr);

printf (“\n\n student name=%s”,student.name);

printf (“\n\n student addr=%s”,student.addr);

getch();

}

UNIONS

A **union** is a special data type available in **C** that allows to store different data types in the same memory location. You can define a **union** with many members, but only one member can contain a value at any given time. **Unions** provide an efficient way of using the same memory location for multiple-purpose.A union is a user- defined type similar to structs in C programming. We recommend you to learn [C structs](https://www.programiz.com/c-programming/c-structures) before you check this tutorial.

# How to define a union?

We use the union keyword to define unions. Here's an example:

1.

2.

3.

4.

5.

union car

{

char name[50]; int price;

};

The above code defines a derived type union car.

# Create union variables

When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Here's how we create union variables.

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

11.

union car

{

char name[50]; int price;

};

int main()

{

union car car1, car2, \*car3; return 0;

}

Another way of creating union variables is:

1.

2.

3.

4.

5.

6.

union car

{

char name[50]; int price;

} car1, car2, \*car3;

In both cases, union variables car1, car2, and a union pointer car3 of union car type are created.

# Access members of a union

We use the . operator to access members of a union. To access pointer variables, we use also use the -

> operator.

In the above example,

* To access price for car1, car1.price is used.
* To access price using car3, either (\*car3).price or car3->price can be used.

# Difference between unions and structures

Let's take an example to demonstrate the difference between unions and structures:

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

11.

12.

13.

14.

15.

16.

17.

18.

19.

20.

21.

22.

#include <stdio.h>

union unionJob

{

//defining a union char name[32]; float salary;

int workerNo;

} uJob;

struct structJob

{

char name[32]; float salary;

int workerNo;

} sJob;

int main()

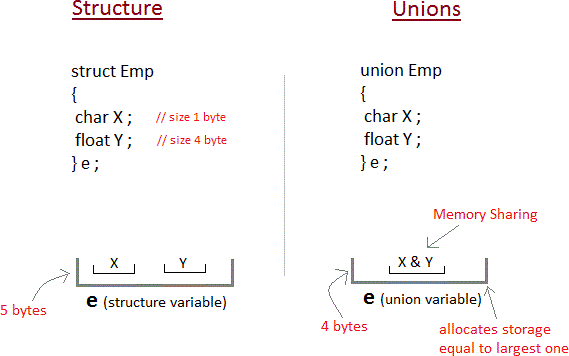
{

printf("size of union = %d bytes", sizeof(uJob)); printf("\nsize of structure = %d bytes", sizeof(sJob)); return 0;

}

# Output

size of union = 32 size of structure = 40



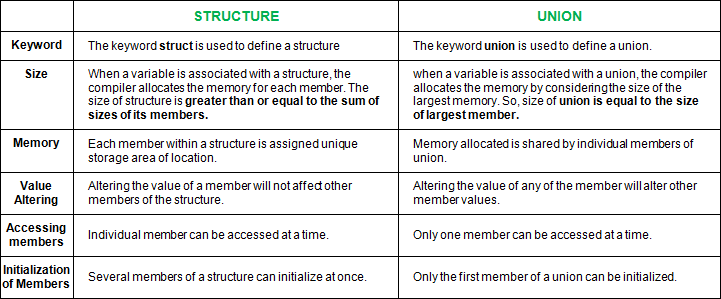
# Why this difference in the size of union and structure variables?

Here, the size of sJob is 40 bytes because

* the size of name[32] is 32 bytes
* the size of salary is 4 bytes
* the size of workerNo is 4 bytes

However, the size of uJob is 32 bytes. It's because the size of a union variable will always be the size of its largest element. In the above example, the size of its largest element, (name[32]), is 32 bytes.

**Differences**



# Only one union member can be accessed at a time

You can access all members of a structure at once as sufficient memory is allocated for all members. However, it's not the case in unions. You can only access a single member of a union at one time. Let's see an example.

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

11.

12.

13.

14.

15.

16.

#include <stdio.h>

union Job

{

float salary; int workerNo;

} j;

int main()

{

j.salary = 12.3;

j.workerNo = 100;

printf("Salary = %.1f\n", j.salary);

printf("Number of workers = %d", j.workerNo); return 0;

}

# Output

Salary = 0.0

Number of workers = 100

Notice that 12.3 was not stored in j.salary.