**ENIGMA INDUCTION**

* Installation & Base setup of C Compiler
* C Data Types & Operators
* Flow Control (if-else)
* Switch
* Loops
* Arrays
* Strings and String Library
* Functions
* Recursion
* Pointers
* Structures

## **About C**

**C** is a general purpose programming language for software development.By design. C provides constructs that map efficiently to typical machine instructions.

## **Get started (IDE installation)**

IDE stands for integrated development environment (IDE), is a software application that provides comprehensive facilities to computer programmers for software development but for our purpose we will be using to compile the code.

Download link for Code Blocks IDE (offline IDE) is [here](http://www.codeblocks.org/downloads/26)

Link to Coding Blocks IDE (online IDE) is [here](https://ide.codingblocks.com/)

## **C quick start**

### **Tokens:**

In C programs, each word and punctuation is referred to as a token. C Tokens are the smallest building block or smallest unit of a C program.

C Supports Six Types of Tokens:

1. Identifiers
2. Keywords
3. Constants
4. Operators
5. Special Symbols

**Identifiers:** Identifiers are names given to different entities such as constants, variables, structures, functions, etc.

Rules for naming identifiers:

1. An identifier can only have alphanumeric characters (a-z , A-Z , 0-9) (i.e. letters & digits) and underscore( \_ ) symbols.
2. Identifier names must be unique
3. The first character must be an alphabet or underscore.
4. You cannot use a keyword as identifiers.
5. Only the first thirty-one (31) characters are significant.
6. It must not contain white spaces.
7. Identifiers are case-sensitive.

**Keywords:** The C Keywords must be in your information because you can not use them as a variable name.You can't use a keyword as an identifier in your C programs, its reserved words in C library and used to perform an internal operation. The meaning and working of these keywords are already known to the compiler.

Below is the list of keywords in C:

[C Keywords List](https://www.w3schools.in/c-tutorial/keywords/#C_Keywords_List)

Example:

int amount;

double totalBalance;

In the above example, amount and totalBalance are identifiers and int, and double are keywords.

**Constants:** Constants are like a variable, except that their value never changes during execution once defined.

C Constants is the most fundamental and essential part of the C programming language. Constants in C are the fixed values that are used in a program, and its value remains the same during the entire execution of the program.

1. Constants are also called literals
2. Constants can be any of the data types.
3. It is considered best practice to define constants using only upper-case names.

**Operators:** C operators are symbols that are used to perform mathematical or logical manipulations. The C programming language is rich with built-in operators. Operators take part in a program for manipulating data and variables and form a part of the mathematical or logical expressions.

Types of C operators:

1. Arithmetic Operators ( +, -, /, \*, %). For more info refer [here](https://www.w3schools.com/cpp/cpp_operators.asp)
2. Relational Operators ( >, <, >=, <=, ==). For more info refer [here](https://www.w3schools.com/cpp/cpp_operators_comparison.asp)
3. Logical Operators ( &&, ||, ! ). For more info refer [here](https://www.w3schools.com/cpp/cpp_operators_logical.asp)
4. Assignment Operators ( =, +=, -= etc ). For more info refer [here](https://www.w3schools.com/cpp/cpp_operators_assignment.asp)
5. Increment and Decrement Operators (++,--). For more info refer [here](https://www.geeksforgeeks.org/difference-between-increment-and-decrement-operators/)
6. Conditional Operator ( ? : ). For more info refer [here](https://www.geeksforgeeks.org/conditional-or-ternary-operator-in-c-c/)
7. Bitwise Operators ( &, |, !, ~, <<, >> ). For more info refer [here](https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/)

## **Data Types:**

The data-type in a programming language is the collection of data with values having fixed meaning as well as characteristics. Some of them are an integer, floating point, character, etc. Usually, programming languages specify the range values for given data-type.

Used to

1. Identify the type of a variable when it is declared.
2. Identify the type of the return value of a function.
3. Identify the type of a parameter expected by a function.

There are 3 types of data types

1.Primery data type/ Built-in data types i.e. void,int,char,double and float

2. Derived data types i.e. Array, references and Pointer

3. User defined data types i.e. Structure, Union and Enumeration

Examples of Data types with variable name

int age;

char letter;

float height,width;

## **Variables:**

The primary purpose of variables is to store data in memory for later use. Unlike constants which do not change during the program execution, variables value may change during execution. If you declare a variable in C, that means you are asking the operating system to reserve a piece of memory with that variable name.

Syntax:

type variable\_name;

(or)

type variable\_name,variable\_name,variable\_name;

Example:

int age;

char letter;

double d;

age=18; //initialization of variable

## **Basic Input and Output(I/O):**

In C programming you can use scanf() and printf() predefined functions to read and print data.

Example:

#include<stdio.h>

void main()

{

int a,b,c;

printf("Please enter any two numbers: \n");

scanf("%d %d", &a, &b);

c = a + b;

printf("The addition of two number is: %d", c);

}

Output:

Please enter any two numbers:

12

3

The addition of two number is:15

The above program scanf() is used to take input from the user, and respectively printf() is used to display output result on the screen.

## **Format Specifiers:**

Format specifiers start with a percentage % operator and followed by a special character for identifying the type of data.

%d = Integer Format Specifier

%f = Float Format Specifier

%c = Character Format Specifier

%s = String Format Specifier

%u = Unsigned Integer Format Specifier

%ld = Long Int Format Specifier

## **Conditional Statements:**

C conditional statements allow you to make a decision, based upon the result of a condition. These statements are called Decision Making Statements or Conditional Statements.

1. **If else:**

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

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

Use the else statement to specify a block of code to be executed if the 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 = 20;

if (time < 18) {

printf("Good day");

}

else {

printf("Good evening");

}

1. **Switch:**

Use the switch statement to select one of many code blocks to be executed.

## **The break Keyword:**

* When C reaches a break keyword, it breaks out of the switch block.
* This will stop the execution of more code and case testing inside the block.
* When a match is found, and the job is done, it's time for a break.

## **The default Keyword:**

The default keyword specifies some code to run if there is no case match:

Example:-

int day = 4;

switch (day) {

case 6:

printf("Today is Saturday");

break;

case 7:

printf("Today is Sunday");

break;

default:

printf("Looking forward to the Weekend");

}

// Outputs "Looking forward to the Weekend"

1. **Ternary Operator / Conditional Operator**

Syntax:

( Condition ) ? Statement 1 : Statement 2 ;

Example:

#include <stdio.h>

int main()

{

int m = 5, n = 4;

(m > n) ? printf("m is greater than n that is %d > %d",

m, n)

: printf("n is greater than m that is %d > %d",

n, m);

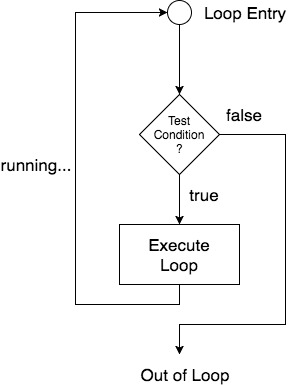
return 0;

}

## Loops in C

In any programming language including C, loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

The below diagram depicts a loop execution.



There are three types of loops in C:

1. **For loop**

Syntax:

for(initialization; condition; increment/decrement)

{

statement-block;

}

In for loop we have exactly two semicolons, one after initialization and second after the condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. But it can have only one **condition**.

The for loop is executed as follows:

1. It first evaluates the initialization code.
2. Then it checks the condition expression.
3. If it is **true**, it executes the for-loop body.
4. Then it evaluates the increment/decrement condition and again follows from step 2.
5. When the condition expression becomes **false**, it exits the loop.

Example:

**#include<stdio.h>**

**void main( )**

**{**

**int i, j;**

**// first for loop //**

**for(i = 1; i < 5; i++)**

**{**

**printf("\n");**

**// second for loop inside the first //**

**for(j = i; j > 0; j--)**

**{**

**printf("%d", j);**

**}**

**}**

**}**

**Output :-**

**1**

**21**

**321**

**4321**

**54321**

## **2. While loop:**

while loop can be addressed as an **entry control** loop. It is completed in 3 steps.

* Variable initialization.(e.g int x = 0;)
* condition(e.g while(x <= 10))
* Variable increment or decrement ( x++ or x-- or x = x + 2 )

**Syntax :**

variable initialization;

while(condition)

{

statements;

variable increment or decrement;

}

Example:

**#include<stdio.h>**

**void main( )**

**{**

**int x;**

**x = 1;**

**while(x <= 10)**

**{**

**printf("%d\t", x);**

**// below statement means, do x = x+1, increment x by 1 //**

**x++;**

**}**

**}**

Output:

1 2 3 4 5 6 7 8 9 10

## **3. do while loop**

In some situations it is necessary to execute the body of the loop before testing the condition. Such situations can be handled with the help of a do-while loop. do statement evaluates the body of the loop first and at the end, the condition is checked using while statement. It means that the body of the loop will be executed at least once, even though the starting condition inside while is initialized to be **false**.

Syntax:

do

{

.....

.....

}

while(condition)

Example:

**#include<stdio.h>**

**void main()**

**{**

**int a, i;**

**a = 5;**

**i = 1;**

**do**

**{**

**printf("%d\t", a\*i);**

**i++;**

**}**

**while(i <= 10);**

**}**

OutPut:

5 10 15 20 25 30 35 40 45 50

## **Array:**

The array is a data structure in C programming, which can store a fixed-size sequential collection of elements of the same data type.

For example, if you want to store ten numbers, it is easier to define an array of 10 lengths, instead of defining ten variables.

In the C programming language, an array can be One-Dimensional, Two-Dimensional, and Multidimensional.

**Syntax:**

datatype arrayName [ size ];

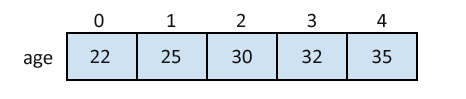
**Example:**

double amount[5];

**Initialize an Array in C**

**int age[5]={22,25,30,32,35};**

**A Pictorial Representation of the Array**



**Accessing Array Elements in C**

**int age[5]={22,25,30,32,35};**

**int myAge = age[1];**

**//assigning 2nd element of array value to integer ‘myAge’**

**Taking Input in an Array and displaying them:**

#include<stdio.h>

void main()

{

int rollNo[10];

// Initializing elements of array separately

for(int n=0;n<sizeof(rollNo)/sizeof(rolNo[0]);n++)

{

rollNo[n] = n;

}

//Displaying the elements

for(int n=0;n<sizeof(rollNo)/sizeof(rolNo[0]);n++)

{

printf(“%d”, rollNo[n]);

}

}

**2D Array:**

An array of arrays is known as 2D array. The two dimensional (2D) array in C programming is also known as matrix. A matrix can be represented as a table of rows and columns.

## **Initialization of 2D Array:**

int disp[2][4] = {

{10, 11, 12, 13},

{14, 15, 16, 17}

}; (recommended)

OR

int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};

## **How to store user input data into 2D array**

We can calculate how many elements a two dimensional array can have by using this formula:

The array arr[n1][n2] can have n1\*n2 elements. The array that we have in the example below is having the dimensions 5 and 4. These dimensions are known as subscripts. So this array has **first subscript** value as 5 and **second subscript** value as 4.

So the array abc[5][4] can have 5\*4 = 20 elements.

#include<stdio.h>

int main(){

/\* 2D array declaration\*/

int abc[5][4];

/\*Counter variables for the loop\*/

int i, j;

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

for(j=0;j<4;j++) {

printf("Enter value for abc[%d][%d]:", i, j);

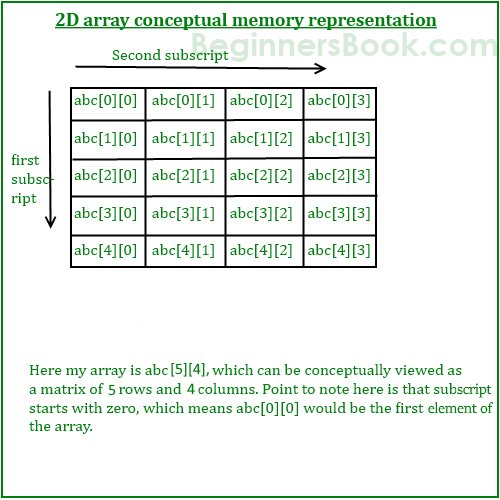
scanf("%d", &abc[i][j]);

}

}

return 0;

}



**Strings:**

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

For handling strings “string.h” header file is used.

The “string.h” header defines one variable type, one macro, and various functions for manipulating arrays of characters.

**Syntax:**

char arrayName [ size ];

**Example:**

char vowels[5];

**Initialize an Array in C**

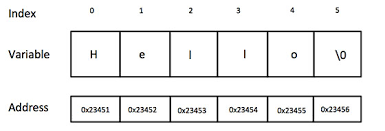
There are different ways to initialize a character array variable.

**char vowels[5]= “abcde”;**

**char vowels[5]={‘a’,’e’,’i’,’o’,’u’,’\0’};**

**char myString[7]={‘A’,’a’,’10’,’99’,’\0’};**

Remember that when you initialize a character array by listing all of its characters separately then you must supply the '\0' character explicitly.

**A Pictorial Representation of the Strings**

**Accessing String Elements in C**

**char alphabet[10]={‘a’,’b’,’c’,’d’,’e’,’f’,’\0’};**

**char myAlpha = alphabet[4];**

**//assigning 4th element of character array value to character ‘myAlpha’**

**Taking Input in an Array and displaying them:**

**#include<stdio.h>**

**#include<string.h> //Header file used for handling of string**

**void main()**

**{**

**char str[20];**

**printf("Enter a string");**

**scanf("%[^\n]", &str);**

**//scanning the whole string,including the white spaces**

**printf("%s", str);**

**}**

### **String Handling Functions**

C language supports a large number of string handling functions that can be used to carry out many of the string manipulations. These functions are packaged in string.h library. Hence, you must include string.h header file in your programs to use these functions.

**The following are the most commonly used string handling functions:-**

|  |  |
| --- | --- |
| **Method** | **Description** |
| **strcat()** | **It is used to concatenate(combine) two strings** |
| **strlen()** | **It is used to show length of a string** |
| **strrev()** | **It is used to show reverse of a string** |
| **strcpy()** | **Copies one string into another** |
| **strcmp()** | **It is used to compare two string** |

#### **strcat() function syntax**

**strcat(“HELLO”,”WORLD”);**

strcat() function will add the string "world" to "hello" i.e it will output HELLO WORLD.

#### **strlen() function syntax**

int j;

j = strlen("studytonight");

printf("%d",j);

strlen() function will return the length of the string passed to it.

Output:

12

#### **strcmp() function syntax**

int j;

j = strcmp("study", "tonight");

printf("%d",j);

strcmp() function will return the ASCII difference between the first unmatching character of two strings.

Output:

0

#### **strcpy() function syntax**

#include<stdio.h>

#include<string.h>

int main()

{

char s1[50];

char s2[50];

strcpy(s1, "StudyTonight"); //copies "studytonight" to string s1

strcpy(s2, s1); //copies string s1 to string s2

printf("%s\n", s2);

return(0);

}

Output:

StudyTonight

#### **strrev() function syntax**

#include<stdio.h>

int main()

{

char s1[50];

printf("Enter your string: ");

gets(s1);

printf("\nYour reverse string is: %s",strrev(s1));

return(0);

}

Output:

Enter your string: Lift-offC

Your reverse string is: Cffo-tfiL

## **FUNCTIONS**

## **DEFINITION**

## Function is basically a set of statements that takes inputs, performs some computation and produces output.

## SYNTAX: return\_type function\_name(set\_of\_inputs);

*set\_of\_inputs* = Inputs provided to the function() – It is not necessary to provide the inputs. Without inputs, it can perform a task and provide an output.

*function\_name* = The name of the function

*return\_type* = The type of output returned by the function( eg:int, char, float)



## 

## 

## **WHY DO WE USE FUNCTIONS?**

## Reusability – Once the function is defined, it can be used again and again.

## Abstraction – If you’re just using the function in your program, you don’t have to worry about how it works inside! Eg: scanf function

## Functions make code modular. Consider a big file having many lines of codes. It becomes really simple to read and use the code if the code is divided into functions.

## **FUNCTION PROTOTYPE**

## A function prototype is a function declaration that specifies the data types of its arguments in the parameter list. The Function prototype serves the following purposes –

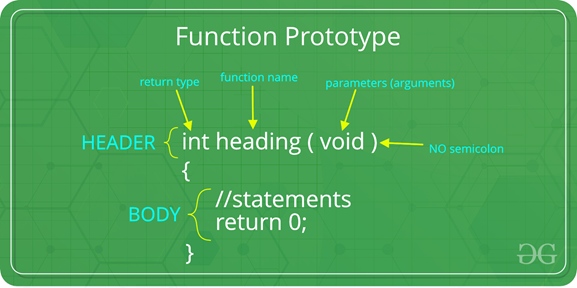
## 1) It tells the return type of the data that the function will return.

## 2) It tells the number of arguments passed to the function.

## 3) It tells the data types of each of the passed arguments.

## 4) Also it tells the order in which the arguments are passed to the function.

## Therefore essentially, function prototype specifies the input/output interface to the function i.e. what to give to the function and what to expect from the function. Prototype of a function is also called the signature of the function. It is always recommended to declare a function before it is used.



* **SCOPE OF THE PROTOTYPE**

Prototypes must be placed appropriately in each compilation unit of a program. The position of the prototype determines its scope. A function prototype, like any function declaration, is considered within the scope of a corresponding function call only if the prototype is specified within the same block as the function call, any enclosing block, or at the outermost level of the source file. The compiler checks all function definitions, declarations, and calls from the position of the prototype to the end of its scope. If you misplace the prototype so that a function definition, declaration, or call occurs outside the scope of the prototype, any calls to that function behave as if there were no prototype. It is an error if the number of arguments in a function definition, declaration, or call does not match the prototype.

## **PASSING PARAMETERS**

## The parameters passed to function are called ***actual parameters***. For example, in the first program 10 and 20 are actual parameters.

## The parameters received by function are called ***formal parameters***. For example, in the first program x and y are formal parameters.

## a. ***Pass by Value:*** In this parameter passing method, values of actual parameters are copied to the function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of the caller.

## b. ***Pass by Reference***: Both actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in actual parameters of the caller. In C, all function parameters are passed by value, so modifying what is passed in callee functions won't affect caller functions' local variables.



* **SOME IMPORTANT POINTS ABOUT FUNCTIONS**

1. Every C program has a function called main() that is called by the operating system when a user runs the program.

**Main Function:**

The main function is a special function. Every C program must contain a function named main. It serves as the entry point for the program. The computer will start running the code from the beginning of the main function.The reason for having the parameter option for the main function is to allow input from the command line.



Since the main function has the return type of int, the programmer must always have a return statement in the code. The number that is returned is used to inform the calling program what the result of the program’s execution was. Returning 0 signals that there were no problems.

1. Every function has a return type. If a function doesn’t return any value, then void is used as return type. Moreover, if the return type of the function is void, we still can use return statement in the body of function definition by not specifying any constant, variable, etc. with it, by only mentioning the ‘return;’ statement which would symbolise the termination of the function as shown below:

## 

## In C, if a function is called before its declaration, the **compiler assumes the return type of the function as int**. **What about parameters?** the compiler assumes nothing about parameters.

## In C, functions can return any type except arrays and functions. We can get around this limitation by returning pointer to array or pointer to function.

## Empty parameter list in C means that the parameter list is not specified and function can be called with any parameters. In C, it is not a good idea to declare a function like fun(). To declare a function that can only be called without any parameter, we should use “void fun(void)”.

## If in a C program, a function is called before its declaration then the C compiler automatically assumes the declaration of that function in the following way:

## 

## And in that case if the return type of that function is different than INT, the compiler would show an error.

## 

## **RECURSION**

Recursion can be defined as the technique of replicating or doing again an activity in a self-similar way calling itself again and again, and the process continues till specific condition reaches. In the world of programming, when your program lets you call that specific function from inside that function with a smaller or different argument, then this concept of calling the function from within itself can be termed as recursion, and the function which makes this possible is called recursive function.

Here's an example of how recursion works in a program:

void rec\_prog(void) {

rec\_prog(); /\* function calls itself \*/}

int main(void) {

rec\_prog();

return 0;

}

C program allows you to do such calling of function within another function, i.e., recursion. But when you implement this recursion concept, you have to be cautious in defining an exit or terminating condition from this recursive function, or else it will continue to an infinite loop, so make sure that the condition is set within your program.

**A Mathematical Interpretation.**

Let us consider a problem : you have to determine the sum of first n natural numbers, there are several ways of doing that but the simplest approach is simply add the numbers starting from 1 to n. So the function simply looks like,

f(n) = 1 + 2 + 3 +……..+ n

but there is another mathematical approach of representing this,

f(n) = 1 ,n=1

f(n) = n + f(n-1) ,n>1

Here the function “ f( ) ” itself is being called inside the function, so this phenomenon is named as recursion and the function containing recursion is called a recursive function.

**Base condition**

Base case is the smallest condition one can think of for a problem.

for the sum of n numbers,

f(n) = n,n=1

is the base case.

In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

int fact(int n)

{

if (n < = 1) // base case

return 1;

else

return n\*fact(n-1);

}

In the above example, the base case for n < = 1 is defined and a larger value of number can be solved by converting to a smaller one till the base case is reached.

**Program to demonstrate recursion.**

int f(int n){

if(n==3)

return 1;

else

return 1+f(n-1);

}

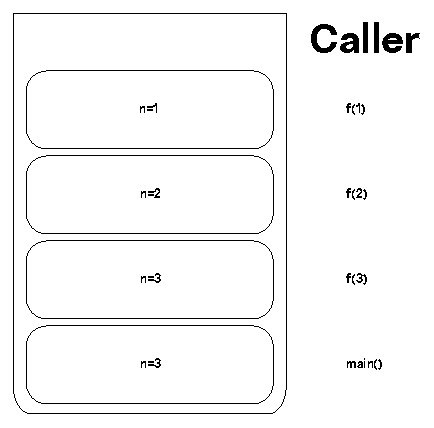
int main(){

int n=3;

printf("%d",f(n));

return 0;

}



here main function calls f(3) then there is a recursive call from

*f(3)→f(2)→f(1).*

after the successful execution of each function, returned values are passed from *f(1)→f(2)→f(1)→main().*

finally the output 3 is printed.

**How a particular problem is solved using recursion?**

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

**Stack Overflow**

If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

int fact(int n)

{

// wrong base case (it may cause

// stack overflow).

if (n == 100)

return 1;

else

return n\*fact(n-1);}

If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

**Direct and Indirect recursion.**

A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun\_new and fun\_new calls fun directly or indirectly.

// An example of direct recursion

void directRecFun()

{

// Some code....

directRecFun();

// Some code...

}

// An example of indirect recursion

void indirectRecFun1()

{

// Some code...

indirectRecFun2();

// Some code...

}

void indirectRecFun2()

{

// Some code...

indirectRecFun1();

// Some code...

}

**What is tail recursion?**

A recursive function is tail recursive when recursive call is the last thing executed by the function. For example the following C function print() is tail recursive.

void print(int n)

{

if (n < 0) return;

printf("%d",n);

// The last executed statement is recursive call

print(n-1);

}

The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by the compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function’s stack frame is of no use.

**Optimising non-tail to tail recursion.**

Consider the following function to calculate factorial of n. It is a non-tail-recursive function. Although it looks like a tail recursive at first look. If we take a closer look, we can see that the value returned by fact(n-1) is used in fact(n), so the call to fact(n-1) is not the last thing done by fact(n) .

unsigned int fact(unsigned int n)

{

if (n == 0) return 1;

return n\*fact(n-1);

}

int main()

{

printf("%d",fact(5)); //output : 120

return 0;

}

The above function can be written as a tail recursive function. The idea is to use one more argument and accumulate the factorial value in the second argument. When n reaches 0, return the accumulated value.

unsigned factTR(unsigned int n, unsigned int a)

{

if (n == 0) return a;

return factTR(n-1, n\*a);

}

unsigned int fact(unsigned int n)

{

return factTR(n, 1);

}

int main()

{

printf("%d",fact(5));

return 0;

}

**How memory is allocated to different function calls in recursion?**

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is deallocated and the process continues.

Let us take the example on how recursion works by taking a simple function.

void printFun(int test)

{

if (test < 1)

return;

else {

printf("%d",test)

printFun(test - 1); // statement 2

printf("%d",test)

return;

}

}

int main()

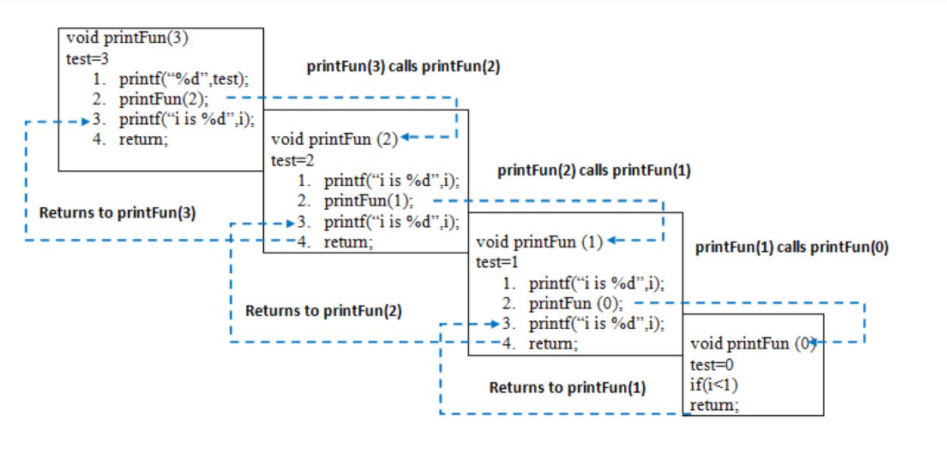
{

int test = 3;

printFun(test);

}

When printFun(3) is called from main(), memory is allocated to printFun(3) and a local variable test is initialized to 3 and statements 1 to 4 are pushed on the stack as shown in the diagram below. It first prints ‘3’. In statement 2, printFun(2) is called and memory is allocated to printFun(2) and a local variable test is initialized to 2 and statement 1 to 4 are pushed in the stack. Similarly, printFun(2) calls printFun(1) and printFun(1) calls printFun(0). printFun(0) goes to if statement and it returns to printFun(1). Remaining statements of printFun(1) are executed and it returns to printFun(2) and so on. In the output, values from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in the diagram below.



**Some Practical problems for better understanding.**

Write a program and recurrence relation to find the Fibonacci series of n where n>2 .

**Mathematical Equation:**

*n if n == 0, n == 1;*

*fib(n) = fib(n-1) + fib(n-2) otherwise;*

**Recurrence Relation:**

*T(n) = T(n-1) + T(n-2) + O(1)*

**Recursive program:**

***Input:*** n = 5

***Output:***

Fibonacci series of 5 numbers is : 0 1 1 2 3

// C code to implement Fibonacci series

#include <stdio.h>

// Function for fibonacci

int fib(int n)

{

// Stop condition

if (n == 0)

return 0;

// Stop condition

if (n == 1 || n == 2)

return 1;

// Recursion function

else

return (fib(n - 1) + fib(n - 2));

}

int main()

{

// Initialize variable n.

int n = 5;

printf("Fibonacci series "

"of %d numbers is: ",

n);

// for loop to print the fibonacci series.

for (int i = 0; i < n; i++) {

printf("%d ", fib(i));

}

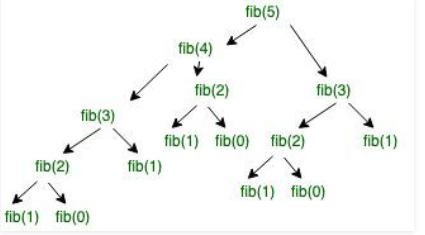
return 0;

}

**Output**

Fibonacci series of 5 numbers is: 0 1 1 2 3

**Working**

****

**2: Write a program and recurrence relation to find the Factorial of n where n>2 .**

**Mathematical Equation:**

1 if n == 0 or n == 1;

f(n) = n\*f(n-1) if n> 1;

**Recurrence Relation:**

***T(n) = 1 for n = 0***

***T(n) = 1 + T(n-1) for n > 0***

**Recursive Program:**

**Input:** n = 5

**Output:**

factorial of 5 is: 120

**Implementation:**

#include <stdio.h>

// Factorial function

int f(int n)

{

// Stop condition

if (n == 0 || n == 1)

return 1;

// Recursive condition

else

return n \* f(n - 1);

}int main()

{

int n = 5;

printf("factorial of %d is: %d", n, f(n));

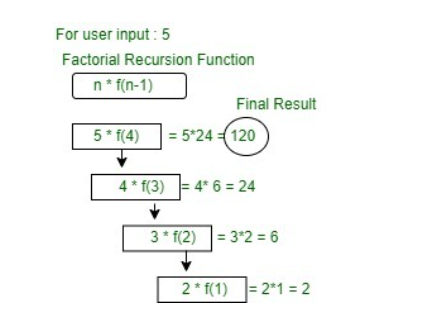
return 0;

}

**Output**

factorial of 5 is: 120

**Working:**



**Disadvantages of recursive programming over iterative programming**

Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

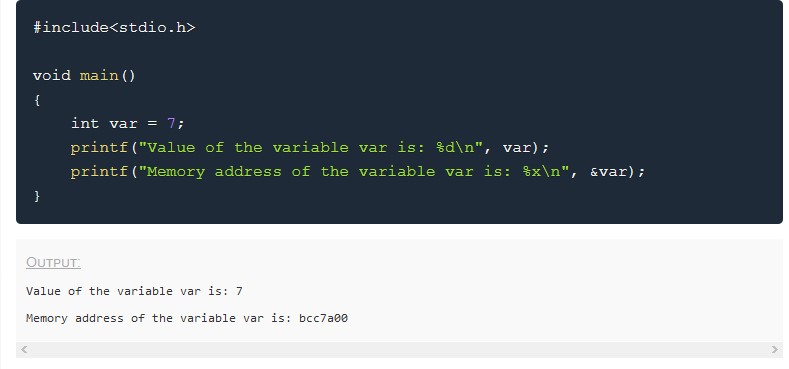
**Advantages of recursive programming over iterative programming.**

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of a stack data structure.

**Addresses in C**

Every variable is stored in the memory on a specific address. To access the variable we use the & symbol. If you have a variable var in your program, &var will give you its address in the memory.

Let's take an example



You must have noticed in the function **scanf()**, we mention &var to take user input for any variable var.

**scanf("%d", &var);**

This is used to store the user inputted value to the address of the variable var.

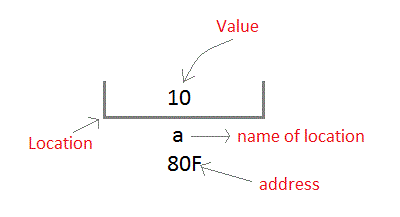
## **Concept of Pointers**

Whenever a **variable** is declared in a program, system allocates a location i.e an address to that variable in the memory, to hold the assigned value. This location has its own address number, which we just saw above.

Let us assume that the system has allocated memory location 80F for a variable a.

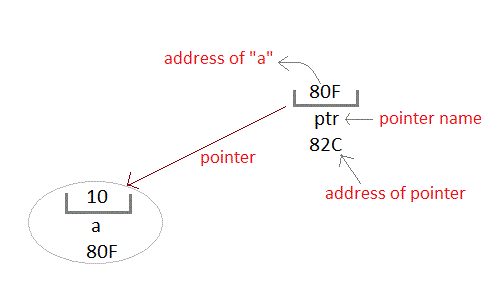
**int a = 10;**

We can access the value 10 either by using the variable name a or by using its address 80F.



The question is how we can access a variable using it's address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses are called **Pointer variables**.

A **pointer** variable is therefore nothing but a variable which holds an address of some other variable. And the value of a **pointer variable** gets stored in another memory location.



### **Benefits of using pointers**

1. Pointers are more efficient in handling Arrays and Structures.
2. Pointers allow references to function and thereby help in passing of function as arguments to other functions.
3. It reduces the length of the program and its execution time as well.
4. It allows C language to support Dynamic Memory management.

### **Declaring a Pointer**

Like variable, pointers in C programming have to be declared before they can be used in your program. Pointers can be named anything you want as long as they obey C's naming rules. A pointer declaration has the following form.

**data\_type \* pointer\_variable\_name;**

Here,

* **data\_type** is the pointer's base type of C's variable types and indicates the type of the variable that the pointer points to.
* The **asterisk** (\*: the same asterisk used for multiplication) which is indirection operator, declares a pointer

Note : The data type of the pointer and the variable to which the pointer variable is pointing must be the same.

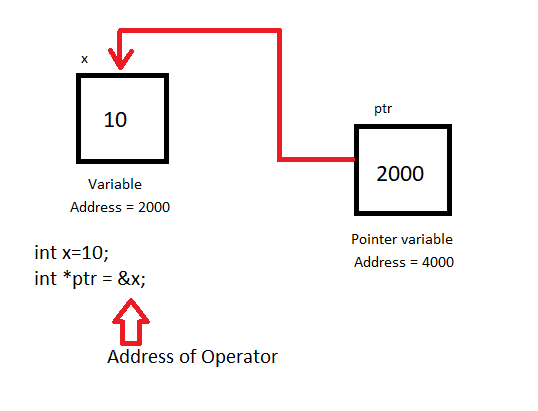
### **Initialization of C Pointer Variables**

Let's take an example.

**int a = 10;**

**int \*ptr; //pointer declaration**

**ptr = &a; //pointer initialization**



While declaring a pointer variable, if it is not assigned to anything then it contains garbage value. Therefore, it is recommended to assign a **NULL** value to it. A pointer that is assigned a NULL value is called a **NULL pointer in C**.

**int \*ptr = NULL;**

### **Few Points to remember while using pointers**

* While declaring/initializing the pointer variable, \* indicates that the variable is a pointer.
* The address of any variable is given by preceding the variable name with Ampersand &.
* The pointer variable stores the address of a variable.
* The declaration int \*a doesn't mean that a is going to contain an integer value. It means that a is going to contain the address of a variable storing integer value.
* To access the value of a certain address stored by a pointer variable \* is used. Here, the \* can be read as 'value at'.

**Let's Look at an Example**

****

Output of the Program

10

10

3795480300

3795480300

3795480304

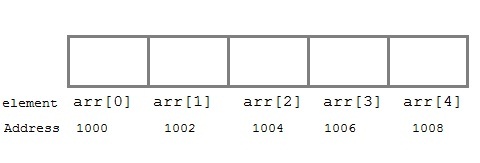
**Note : These address vary on each run**

# **Pointer and Arrays in C**

Let us Declare an array arr

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

Suppose the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:



Variable **arr** will give the base address, which is a **constant pointer** pointing to arr[0]. Hence arr contains the address of arr[0] i.e 1000.

**arr has two purpose -**

* **It is the name of the array**
* **It acts as a pointer pointing towards the first element in the array.**

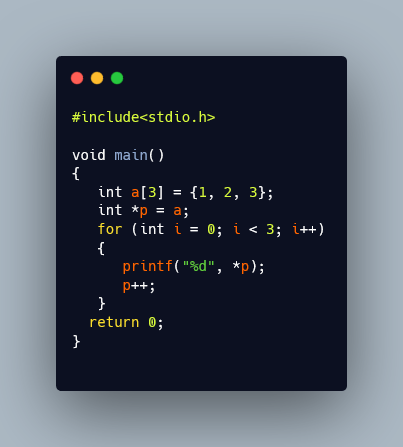
**Note :**

* **arr is equal to &arr[0] by default**
* **You cannot decrement a pointer once incremented i.e p-- won't work.**

**Syntax:**

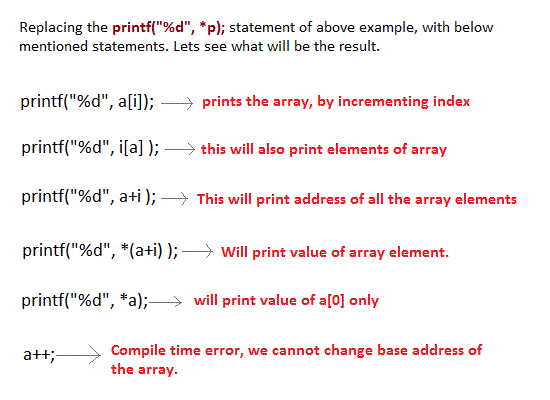
**\*(a+i)** //pointer with an array is same as: **a[i]**

Let's look at an example



OUTPUT

1 2 3



## **Pointer and Character strings**

Pointer is used to create strings. Pointer variables of char type are treated as string.

**char \*str = "Hello";**

The above code creates a string and stores its address in the pointer variable str. The pointer str now points to the first character of the string "Hello".

The string created using a char pointer can be assigned a value at **runtime**.

**char \*str;**

**str = "hello";**

The content of the string can be printed using printf() and puts().

**printf("%s", str);**

**puts(str);**

str is a pointer to the string and also name of the string. Therefore we do not need to use indirection operator \*.

# **Pointers as Function Argument in C**

# Pointer as a function parameter is used to hold addresses of arguments passed during function calls. This is also known as call by reference. When a function is called by reference any change made to the reference variable will affect the original variable.

### **Example Time: Swapping two numbers using Pointer**

OUTPUT

m = 10

n = 20

After Swapping:

m = 20

n = 10

**Other Topics with Pointers(Does it need to be added?)**

* Functions returning Pointer variables
* Pointer to functions
* Pointer Arithmetic in C
* Pointer to Array of Structures in C
* Accessing Structure Members with Pointer
* Pointer to Multidimensional Array
* Array of Pointers
* Pointer to a Pointer in C (Double Pointer)

## **Structure**

***What is a structure?***

A structure is a user defined data type in C. A structure creates a data type that can be used to group items of possibly different types into a single type.

***How to create a structure?***

‘struct’ keyword is used to create a structure. Following is an example.

struct address

{

char name[50];

char street[100];

char city[50];

char state[20];

int pin;

};

***How to declare structure variables?***

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

// A variable declaration with structure declaration.

struct Point

{

int x, y;

} p1; // The variable p1 is declared with 'Point'

// A variable declaration like basic data types

struct Point

{

int x, y;

};

int main()

{

struct Point p1; // The variable p1 is declared like a normal variable

}

***How to initialize structure members?***

Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.

struct Point

{

int x = 0; // COMPILER ERROR: cannot initialize members here

int y = 0; // COMPILER ERROR: cannot initialize members here

};

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members **can be** initialized using curly braces ‘{}’. For example, following is a valid initialization.

struct Point

{

int x, y;

};

int main()

{

// A valid initialization. member x gets value 0 and y

// gets value 1. The order of declaration is followed.

struct Point p1 = {0, 1};

}

***How to access structure elements?***

Structure members are accessed using dot (.) operator.

#include<stdio.h>

struct Point

{

int x, y;

};

int main()

{

struct Point p1 = {0, 1};

// Accessing members of point p1

p1.x = 20;

printf ("x = %d, y = %d", p1.x, p1.y);

return 0;

}

**Output:**

x = 20, y = 1

***What is an array of structures?***

Like other primitive data types, we can create an array of structures.

#include<stdio.h>

struct Point

{

int x, y;

};

int main()

{

// Create an array of structures

struct Point arr[10];

// Access array members

arr[0].x = 10;

arr[0].y = 20;

printf("%d %d", arr[0].x, arr[0].y);

return 0;

}

**Output:**

10 20

***What is a structure pointer?***

Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow ( -> ) operator.

#include<stdio.h>

struct Point

{

int x, y;

};

int main()

{

struct Point p1 = {1, 2};

// p2 is a pointer to structure p1

struct Point \*p2 = &p1;

// Accessing structure members using structure pointer

printf("%d %d", p2->x, p2->y);

return 0;

}

**Output:**

1 2

## **Union**

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.

**Defining a Union**

To define a union, you must use the union statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

**Syntax:**

union [union tag] {

member definition;

member definition;

....

member definition;

} [one or more union variables];

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon.

You can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

union Data {

int i ;

float f ;

char str[20] ;

} data ;

* A variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data.
* You can use any built-in or user defined data types inside a union based on requirement.

The memory occupied by a union will be large enough to hold the largest member of the union.

In the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string.

**Example 1:**

#include<stdio.h>

#include<string.h>

union Data {

int i ;

float f ;

char str[20] ;

} ;

int main( ) {

union Data data;

printf(“Memory size occupied by data : %d\n”, sizeof(data));

return 0;

}

**Output:**

Memory size occupied by data : 20

**Accessing Union Members**

To access any member of a union, we use the **member access operator** (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type.

The following example shows how to use unions in a program −

**Example 2:**

#include<stdio.h>

#include<string.h>

union Data {

int i ;

float f ;

char str[20] ;

} ;

int main( ) {

union Data data;

data.i= 10 ;

data.f= 220.5 ;

strcpy( data.str, “C programming”);

printf( “data.i : %d\n”, data.i);

printf( “data.f : %f\n”, data.f);

printf(“ data.str :%s\n”, data.str);

return 0;

}

**Output:**

data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : C programming

Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions −

**Example 3:**

#include<stdio.h>

#include<string.h>

union Data {

int i ;

float f ;

char str[20] ;

};

int main( )

{

union Data data ;

data.i= 10;

printf(“ data.i : %d\n”,data.i);

data.f= 220.5;

printf(“ data.f : %f\n”,data.f);

strcpy( data.str, ”C Programming”);

printf( “data.str : %s\n” , data.str);

return 0;

}

**Output:**

data.i : 10

data.f : 220.500000

data.str : C Programming

Here, all the members are getting printed very well because one member is being used at a time.

**Applications of Union**

Unions can be useful in many situations where we want to use the same memory for two or more members.

For example, suppose we want to implement a binary tree data structure where each leaf node has a double data value, while each internal node has pointers to two children, but no data. If we declare this as:

struct NODE {

struct NODE\* left;

struct NODE\* right;

double data;

};

then every node requires 16 bytes, with half the bytes wasted for each type of node. On the other hand, if we declare a node as following, then we can save space.

struct NODE {

bool is\_leaf;

union {

struct

{

struct NODE\* left;

struct NODE\* right;

} internal;

double data;

} info;

};

**Pointers to union**

Like structures, we can have pointers to unions and can access members using the arrow operator (->). The following example demonstrates the same.

#include <stdio.h>

union test {

int x;

char y;

};

int main()

{

union test p1;

p1.x = 65;

// p2 is a pointer to union p1

union test\* p2 = &p1;

// Accessing union members using pointer

printf("%d %c", p2->x, p2->y);

return 0;

}

**Output :** 65 A

**Questions To Be Practised :**

**1. Reverse The Number :**

[**https://www.codechef.com/problems/FLOW007**](https://www.codechef.com/problems/FLOW007)

**2. Packaging Cupcakes :**[**https://www.codechef.com/problems/MUFFINS3**](https://www.codechef.com/problems/MUFFINS3)

**3. Mahasena :**

[**https://www.codechef.com/problems/AMR15A**](https://www.codechef.com/problems/AMR15A)

**4. GCD and LCM :** [**https://www.codechef.com/problems/FLOW016**](https://www.codechef.com/problems/FLOW016)

**5. IPL and RCB :**

[**https://www.codechef.com/problems/CLIPLX**](https://www.codechef.com/problems/CLIPLX)

**6. Watermelon :** [**https://www.codechef.com/problems/WATMELON**](https://www.codechef.com/problems/WATMELON)

**7. Lefthanders and Right Handers :** [**https://codeforces.com/problemset/problem/234/A**](https://codeforces.com/problemset/problem/234/A)

**8. Spoilt Permutation :** [**https://codeforces.com/problemset/problem/56/B**](https://codeforces.com/problemset/problem/56/B)

**9. Reconnaissance 2 :** [**https://codeforces.com/problemset/problem/34/A**](https://codeforces.com/problemset/problem/34/A)

**10.Rook, Bishop and King :** [**https://codeforces.com/problemset/problem/370/A**](https://codeforces.com/problemset/problem/370/A)