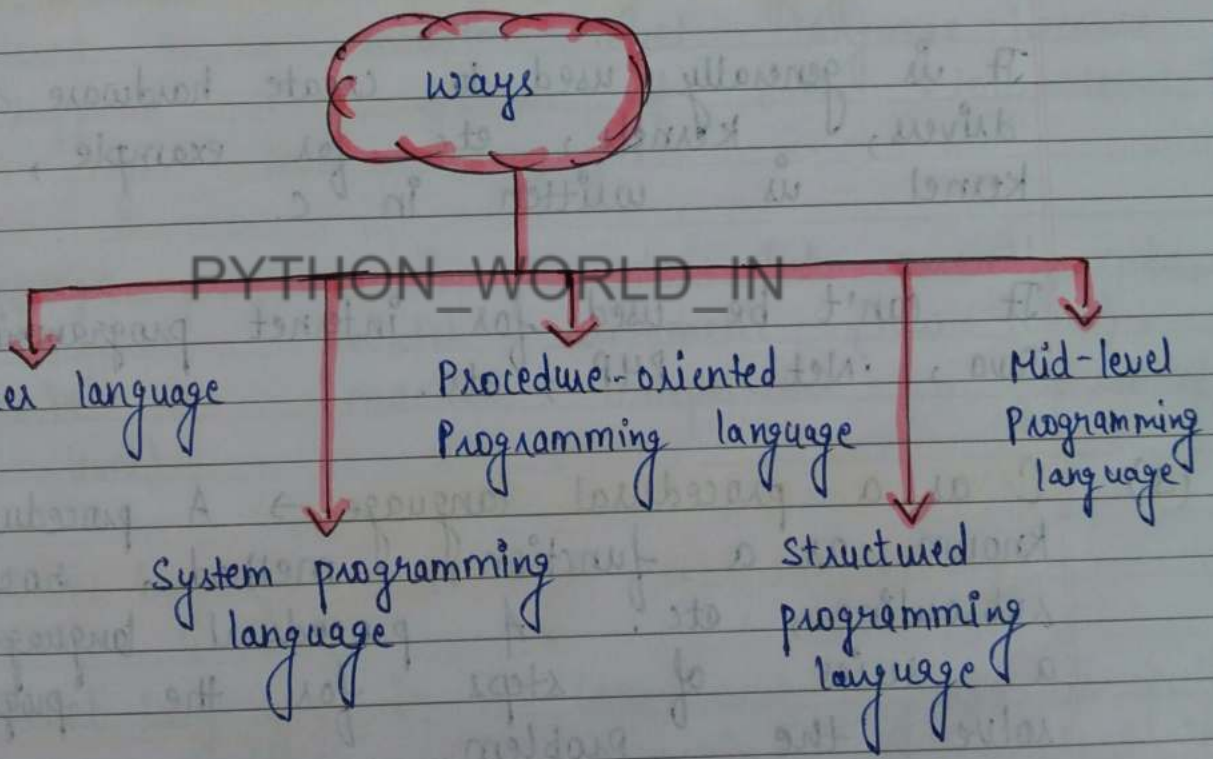
C Tutorial

- What is C Language
- History of C
- Features of C
- How to install C
- First C program
- Flow of C program
- Print scanf
- Variables in C
- Data types and keywords in C
- C operators
- C comments
- C Escape Sequence
- Constants in C

Introduction of C :-

The C Language is developed by Dennis Ritchie for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.

C programming is considered as the base for other programming languages, that is why it is known as mother language.
It can be defined by the following ways:



(1). C as a mother language → C language is considered as the mother language of all the modern programming languages because most of the compilers, JVMs, kernels, etc. are written in C language, and most of the programming languages follow C syntax for example, C++, Java, C#, etc.

It provides the core concepts like the array, strings, functions, file handling, etc. that are being used in many languages like C++, Java, C#, etc.

(2) C as a system programming language → A system programming language is used to create system software. C language is a system programming language because it can be used to do low level programming (for example driver and kernel).

It is generally used to create hardware devices, OS, drivers, kernels, etc. for example, Linux kernel is written in C.

It can't be used for internet programming like Java, .Net, PHP, etc.

(3) C as a procedural language → A procedure is known as a function, method, routine, subroutine, etc. A procedural language specifies a series of steps for the program to solve the problem.

A procedural language breaks the program into functions, data structures, etc.

C is a procedural language. In C, variables and function prototypes must be declared before being used.

(4) C as a structural programming language → A structured programming language is a subset of the procedural languages. Structure means to break a program into parts or blocks so that it may be easy to understand.

In the C language, we break the program into parts using functions. It makes the program easier to understand and modify.

(5) C as a mid-level programming language → C is considered as a middle-level language because it supports the feature of both low-level and high-level languages.

C language program is converted into assembly code, it supports pointer arithmetic (low-level), but it is machine independent (a feature of high-level).

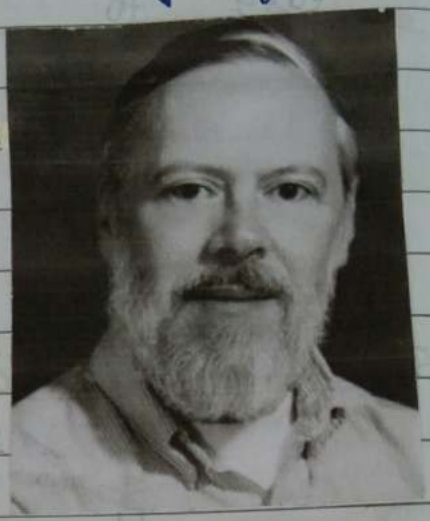
Low level language and High level language ↴

Low level language :- A low level language is specific to one machine, i.e., machine dependent. it it is machine dependent, fast to run. But it is not easy to understand.

High level language :- A High level language is not specific to one machine, i.e., machine independent. It is easy to understand.

History of C language :-

History of C language is interesting to know. Here we are going to discuss a brief history of the C language.

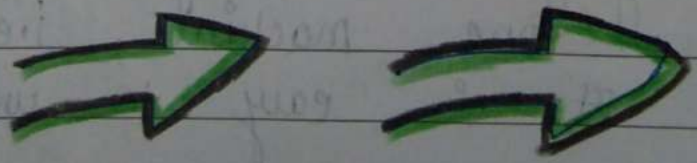


C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T. (which stands for American Telephone & Telegraph), located in the U.S.A.

Dennis Ritchie is known as the founder of the C language. It was developed to overcome the problems of previous languages such as B, BCPL, etc.

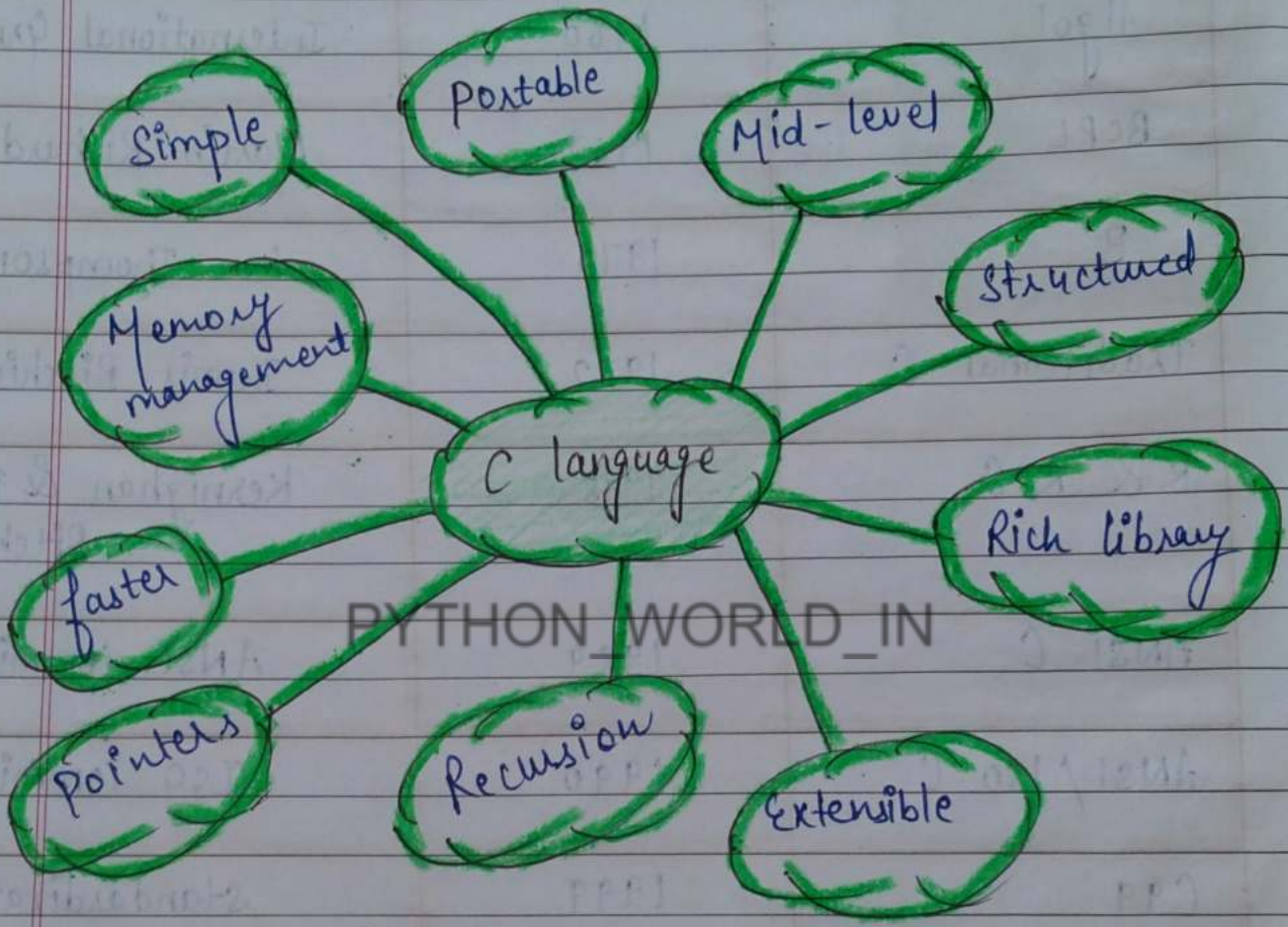
Initially, C language was developed to be used in UNIX operating system. It inherits many features of previous language such as B and BCPL.

Let's see the programming languages that were developed before C language.



Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

Features of C language :



C is the widely used language. It provides many features that are given below:

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language

4. Structured programming language
5. Rich library.
6. Memory Management
7. Fast Speed
8. pointers
9. Recursion
10. Extensible

(1) C is a simple language in the sense that it provides a structured approach (to break the problem into parts), the rich set of library functions, data types, etc.

(2) Machine Independent or Portable unlike assembly language, C programs can be executed on different machines with some machine specific changes. Therefore, C is a machine independent language.

(3) Mid-level programming language Although, C is intended to do low-level language programming. It is used to develop system applications such as kernel, driver,

etc. It also supports the features of a high level language. That is why it is known as mid-level language.

(4) Structured programming language: C is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify. functions also provide code reusability.

(5) Rich library :- C provides a lot of inbuilt functions that make the development fast.

(6) Memory Management :- It supports the feature of dynamic memory allocations. In C language, we can free the allocated memory at any time by calling the free() function.

(7) Speed :- The compilation and execution time of C language is fast since there are lesser inbuilt functions and hence the lesser overhead.

(8) Pointer :- C provides the features of pointers, we can directly interact with the

Memory by using the pointers. we can can use pointers for memory, structures, functions, array, etc.

(9) Recursion :- In C, we can call the function within the function. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

(10) Extensible :- C language is extensible because it can easily adopt new features.

How to install C

There are many compilers available for c and c++. you need to download any one. Here, we are going to use Turbo c++. It will work for both c and c++. To install the Turbo C software, you need to follow following steps.

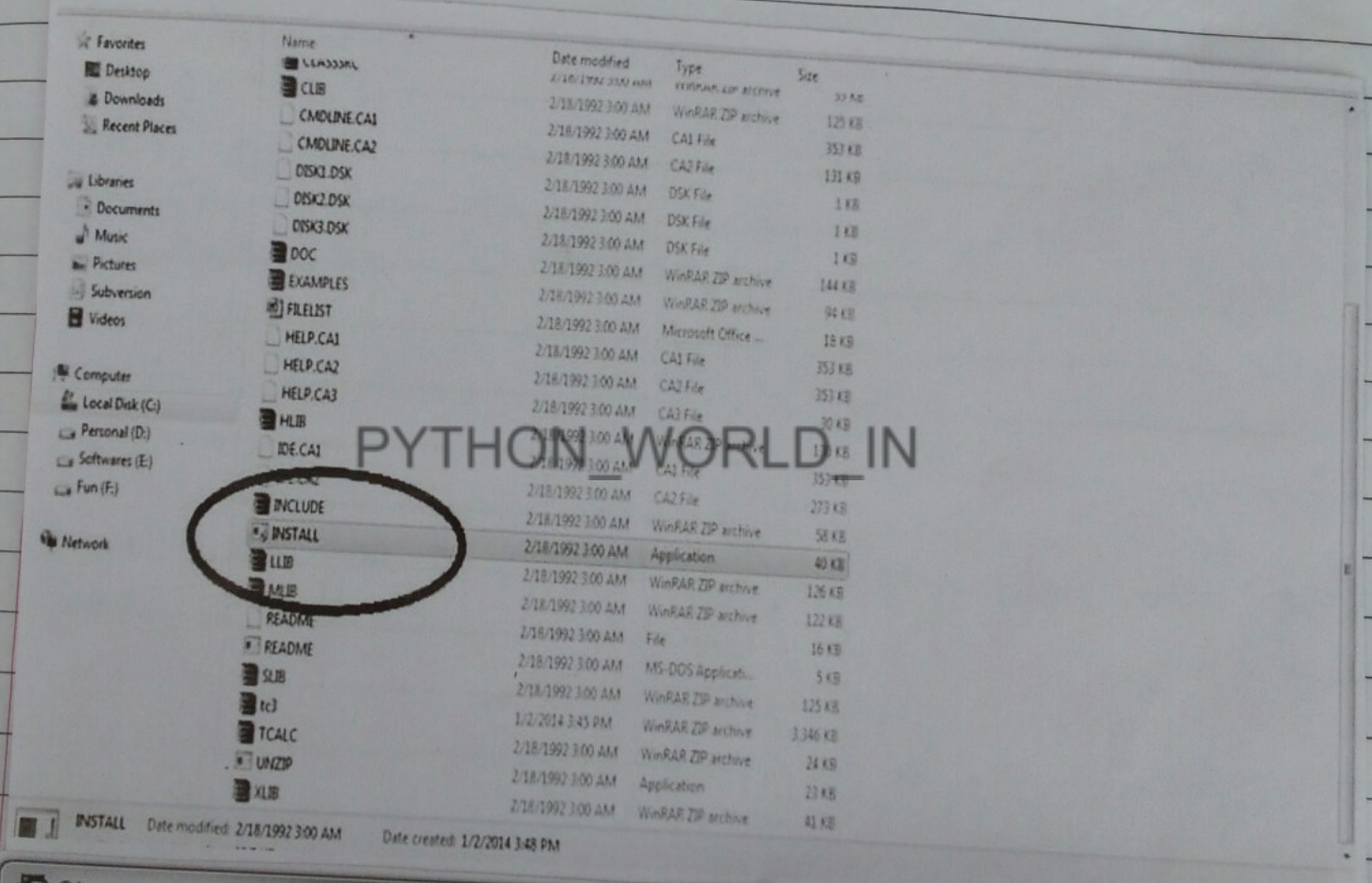
1. Download Turbo c++
2. create turboc directory inside c drive and extract the tc3.zip inside c:\turboc
3. Double click on install.exe file.
4. click on the tc application file located inside c:\TC\BIN to write the c program.

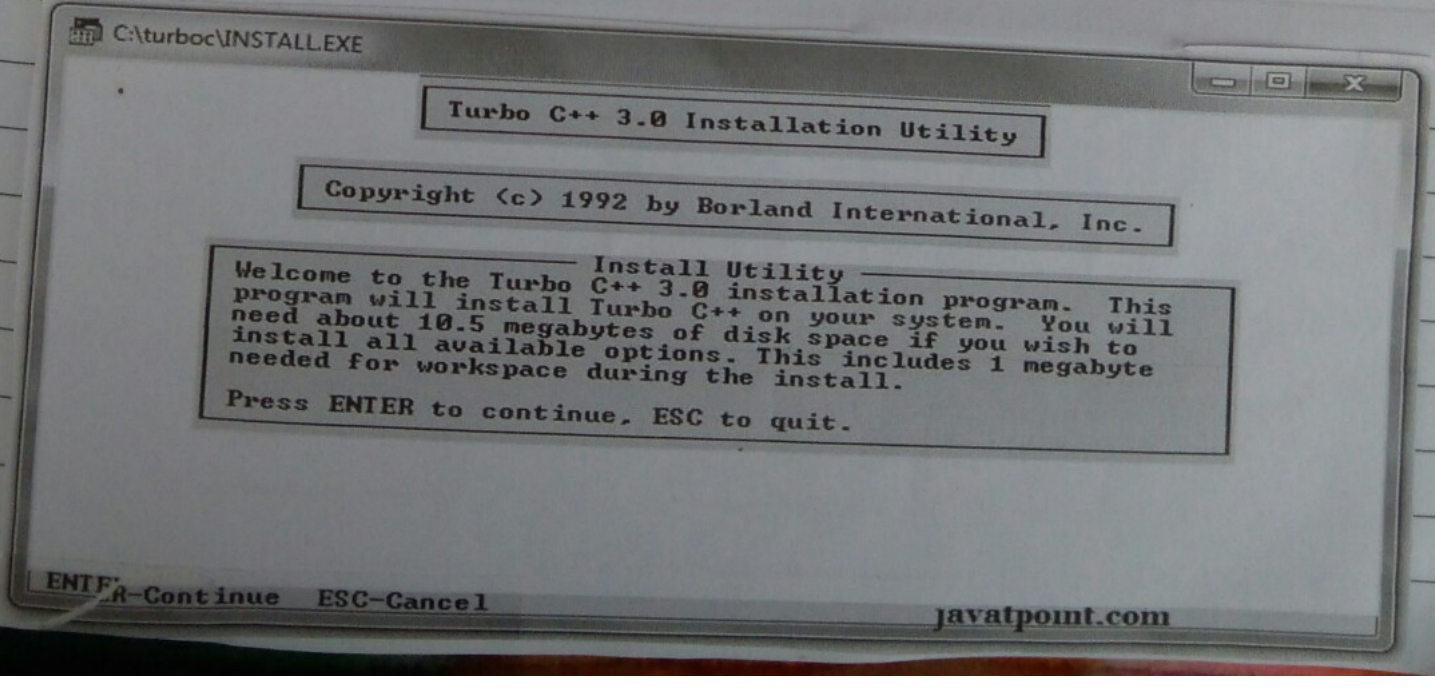
1) Download Turbo C++ Software
you can download turbo c++ from many sites. download Turbo c++

2) Create turboc directory in c drive and extract the tc3.zip

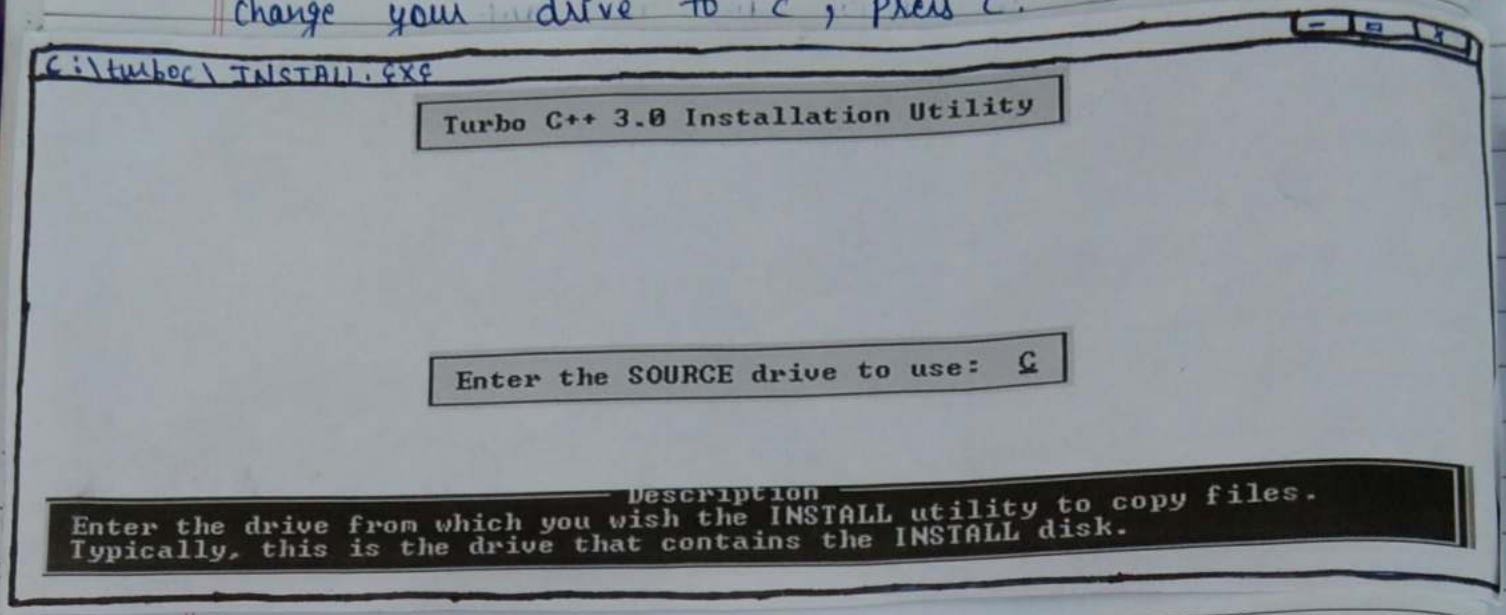
Now, you need to create a new directory turboc inside the c: dive. Now extract the tc3.zip file in c:\turboc directory.

3) Double click on the install.exe file and follow steps Now, click on the install icon located inside the c:\turboc

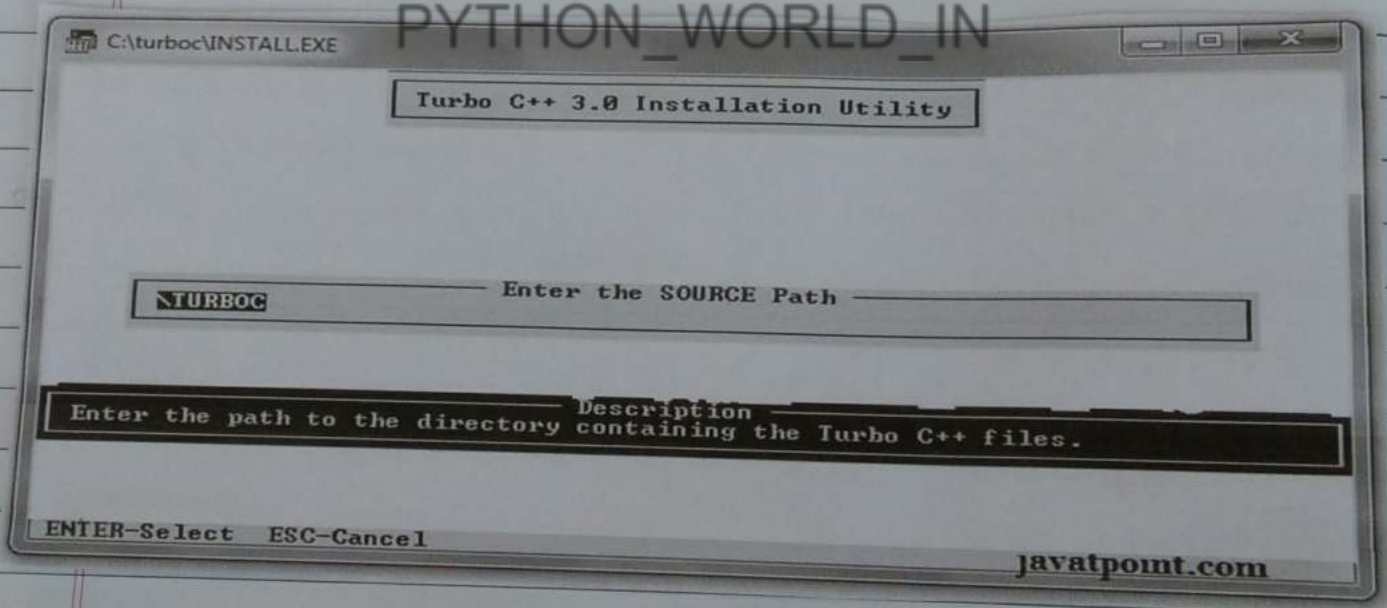




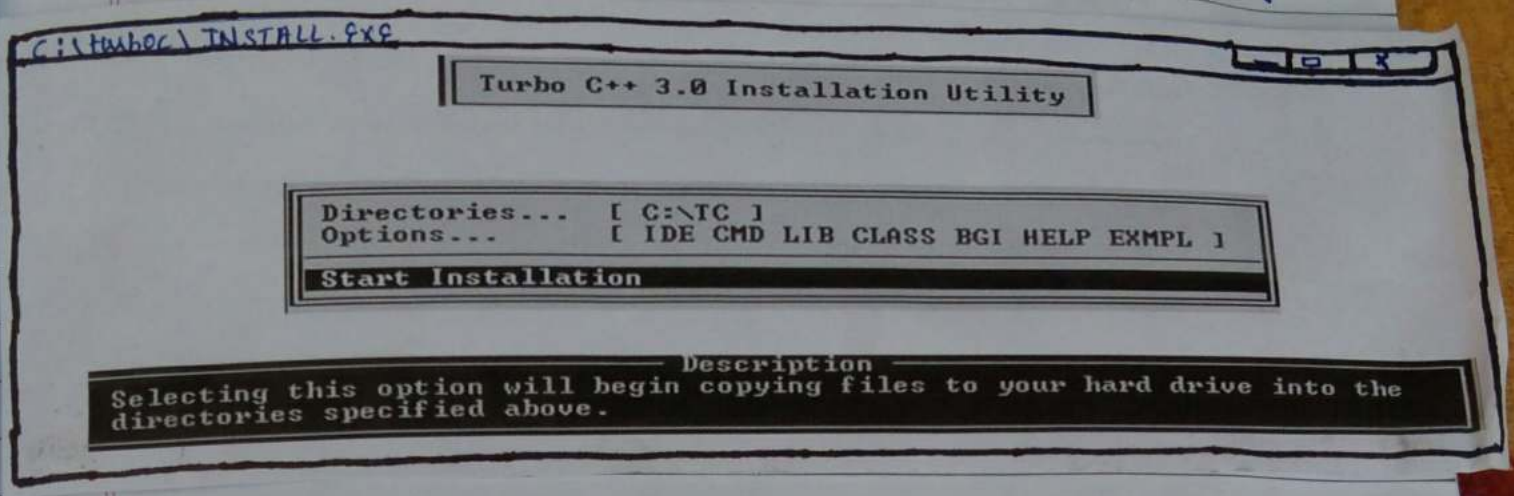
change your drive to c, press c.



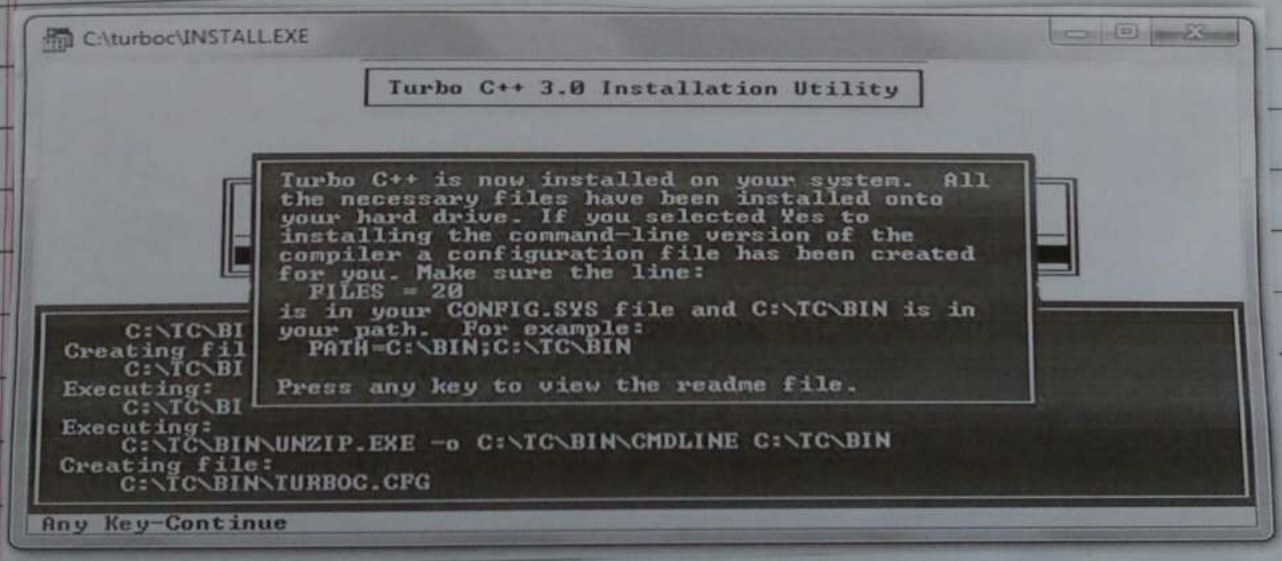
Press enter, it will took inside the c:\turboc directory for the required files.



Select start installation by the down arrow key then press enter.

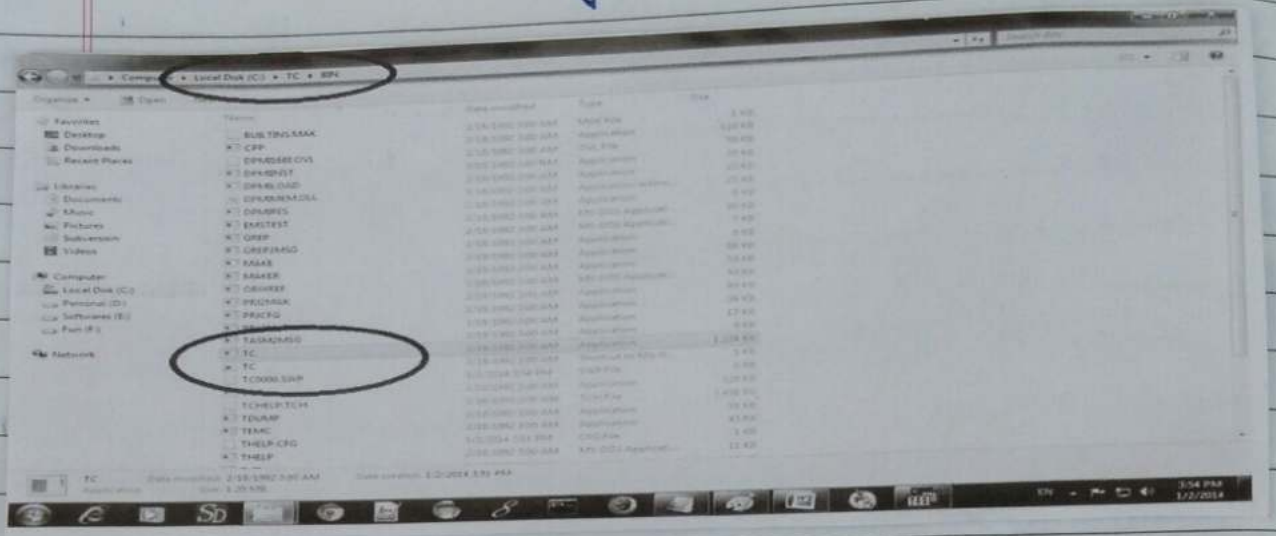


Now C is installed, press enter to read documentation or close the software.



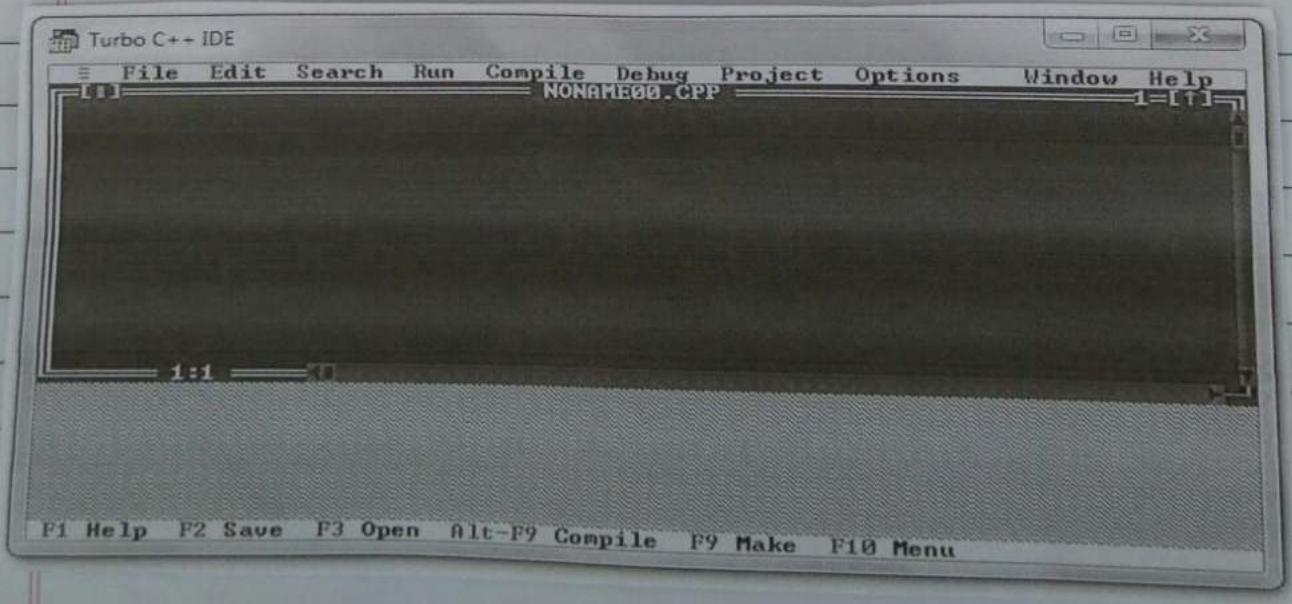
4) click on the tc application located inside the C:\TC\BIN.

Now double click on the tc icon located in C:\TC\BIN directory to write the C program.



In windows 7 or window 8, it will show a dialog block to ignore and close the application because full screen mode is not supported.
click on ignore button.

Now it will showing following console :



First C Program

Before starting the abcd of C language, you need to learn how to write, compile and run the first C program.

To write the first c program, open the C console and write the following code :

Source code

```
#include <stdio.h>
int main(){
printf("Hello C Language");
return 0;
}
```


#include <stdio.h> includes the standard input output library functions. The printf() function is defined in stdio.h

int main() The main() function is the entry point of every program in c language.

printf() The printF() function is used to print data on the console.

return 0 The return 0 statement, returns execution status to the OS. The 0 value is used for succesful execution and 1 for unsuccesful execution.

How to compile and run the C program
There are 2 ways to compile and run the C program, by menu and by shortcut.

By menu
Now click on the compile menu then compile sub menu to compile the C program.
Then click on the run menu then run sub menu to run the C program.

By shortcut
Or, press ctrl+F9 keys compile and run the program directly.
You will see the following output on user screen.

you can view the user screen any time by pressing the alt + F5 keys.

Now press Esc to return to the turbo c++ console.

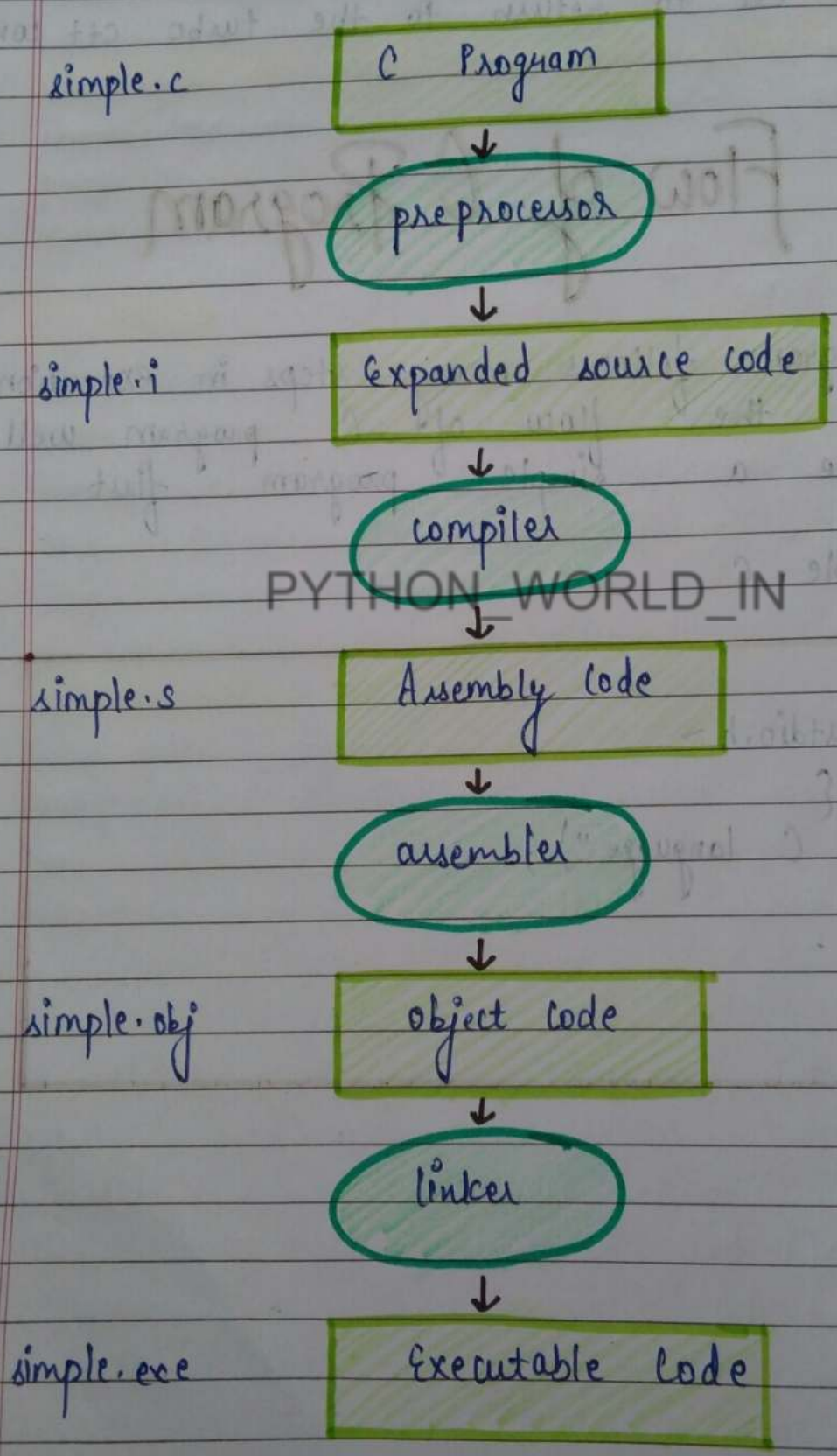
Flow of C Program

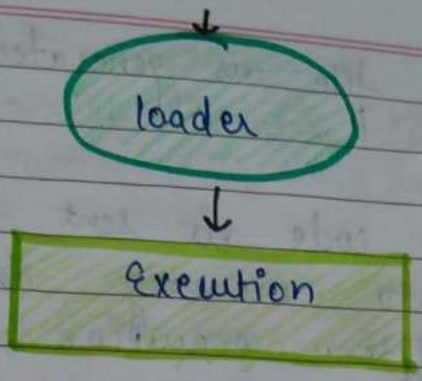
The C program follows many steps in execution. To understand the flow of C program well, let us see a simple program first.

File : simple.c


```
#include <stdio.h>
int main() {
printf("Hello C language");
return 0;
}
```

Execution Flow





Let's try to understand the flow of above program by the figure given below.

1) C program (source code) is sent to preprocessor first. The preprocessor is responsible to convert the preprocessor directives into their respective values. The preprocessor generates an expanded source code.

2) Expanded source code is sent to compiler which compiles the code and converts it into assembly code.

3) The assembly code is sent to assembler which assembles the code and converts it into object code. Now a simple.obj file is generated.

4) The object code is sent to linker which links it to the library such as header files. Then it is converted into executable code.

A simple .exe file is generated.

5) The executable code is sent to loader which loads it into memory and then it is executed. After execution, output is sent to console.

PrintF() and ScanF() in C

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

printf() function

The printf() function is used for output. It prints the given statement to the console.
The syntax of printf() function is given below:

```
printf("format string", argument_list);
```

The format string can be %d (integer), %c (character), %s (string), %f (float) etc.

scanf() function

The scanf() function is used for input. It reads the input data from the console.

```
scanf("format string", argument_list);
```

Program :- To print cube of given number

Let's see a simple example of c language that gets input from the user and the prints the cube of the given number.

Source code


```
#include <stdio.h>
int main() {
int number;
printf("enter a number:");
scanf("%d", &number);
printf("cube of number is: %d", number * number * number);
return 0;
}
```

Output

```
enter a number : 5

Cube of number is : 125
```

The scanf("%d", &number) statement reads integer number from the console and stores the given value in number variable.

The printf("cube of number is : %d", number* number* number) statement prints the cube of number on the console.

Program :- To print sum of 2 numbers

Let's see a simple example of input and output in C language that prints addition of 2 numbers.

Source code

```
#include <stdio.h>
int main() {
int x=0, y=0, result=0;

printf("enter first number:");
scanf("%d", &y);

result = x+y
printf("sum of 2 numbers : %d", result);

return 0;
}
```

Output

```
enter first number : 9
enter second number : 9
sum of 2 numbers : 18
```

Variables in C

A variable is a name of the memory location. It is used to store data. its value can be changed, and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

```
type variable_list ;
```

The example of declaring the variable is given below:

```
int a ;
Float b ;
char c ;
```

Here, a, b, c are variables. The int, float, char are the data types.
We can also provide values while declaring the variables as given below:

```
int a = 10, b = 20; // declaring 2 variable of integer type
float f = 20.8;
char c = 'A';
```

Rules for defining variables

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

valid variables names :

```
int a;
int _ab;
int a30;
```

Invalid variables names :

```
int 2;
int a_b;
int long;
```


Types of Variables in C

There are many types of variables in c :

Local Variables

A variable that is declared inside the function or block is called a local variable. It must be declared at the start of the block.

```
void function1() {
int x = 10; // local variable
}
```

you must have to initialize the local variable before it is used.

Global Variables

A variable that is declared outside the function or block is called a global variable. Any function can change the value of the global variable. It is available to all the functions.

It must be declared at the start of the block.

```
int value = 20; // global variable
void function1() {
int x = 10; // local variable
}
```

Static Variable

A variable that is declared with the static keyword is called static variable. It retains its value between multiple function calls.

Source code

```
void function1 () {
int x = 10; // local variable
static int y = 10; // static variable
x = x+1;
y = y+1;
printf("%d, %d", x, y);
}
```


If you call this function many times, the local variables will print the same value for each function call, e.g, 11, 11, 11 and so on. But the static variable will print the incremented value in each function call, e.g, 11, 12, 13 and so on.

Automatic Variable

All variables in C that are declared inside the block, are automatic variables by default. We can explicitly declare an automatic

variable using auto keyword.

```
void main () {
int x = 10 ; // local variable (also automatic)
auto int y = 20 ; // automatic variables
}
```

External Variable

We can share a variable in multiple C source files by using an external variable. To declare an external variable, you need to use extern keyword.

myfile.h

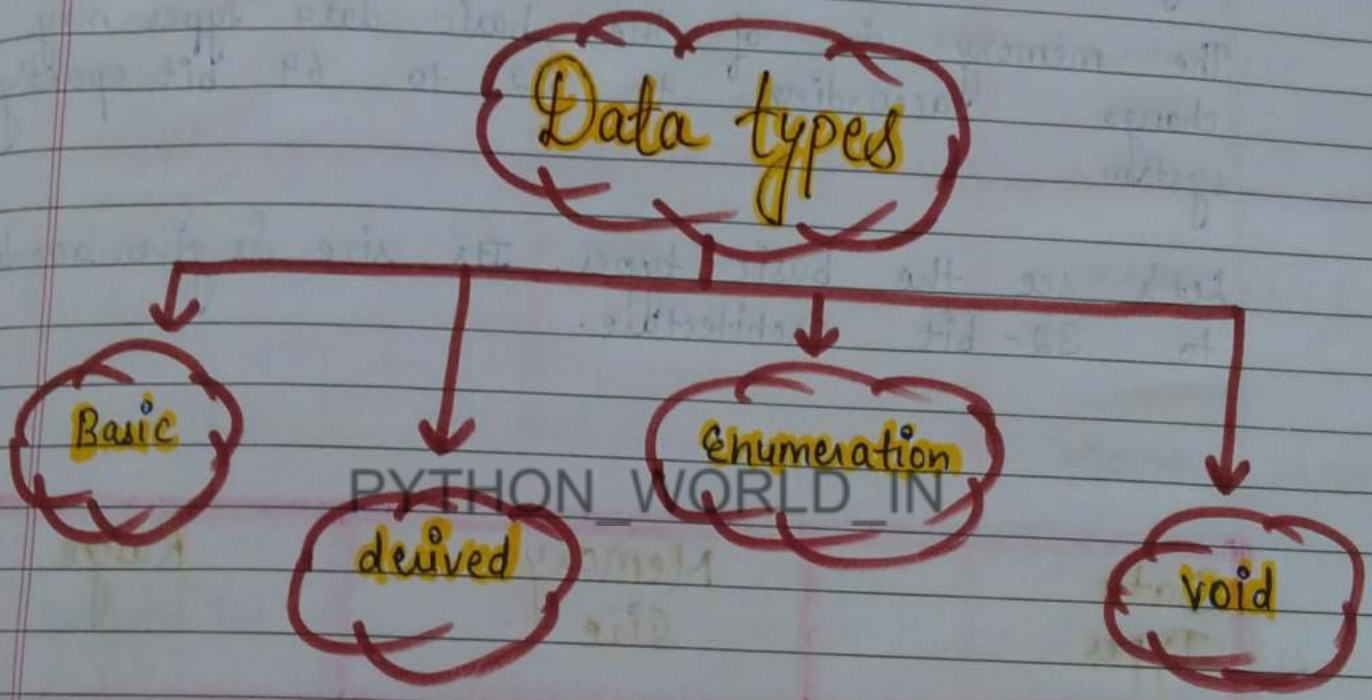
```
extern int x = 10 ; // external variable (also global)
```

program1.c

```
#include "myfile.h"
#include <stdio.h>
void printValue () {
    printf ("Global variable : %d ", global_variable);
}
```

Data types in C

A data type specific the type of data that a variable can store such as integer, floating, character, etc.



There are the following data types in C languages.

Types	Data Types
Basic data type	int, char, float, double
derived data type	array, pointer, structure, union
Enumeration Data type	enum
void Data type	void

short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	2 byte	-32,768 to 32,767
signed int	2 byte	-32,768 to 32,767
unsigned int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long in	4 byte	0 to 4,294,967,295

short int	2 byte	-32,768 to 32,767
signed short int	2 byte	-32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
float	4 byte	
double	8 byte	
long double	10 byte	

Keywords in C

A keyword is a reserved word. you cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:

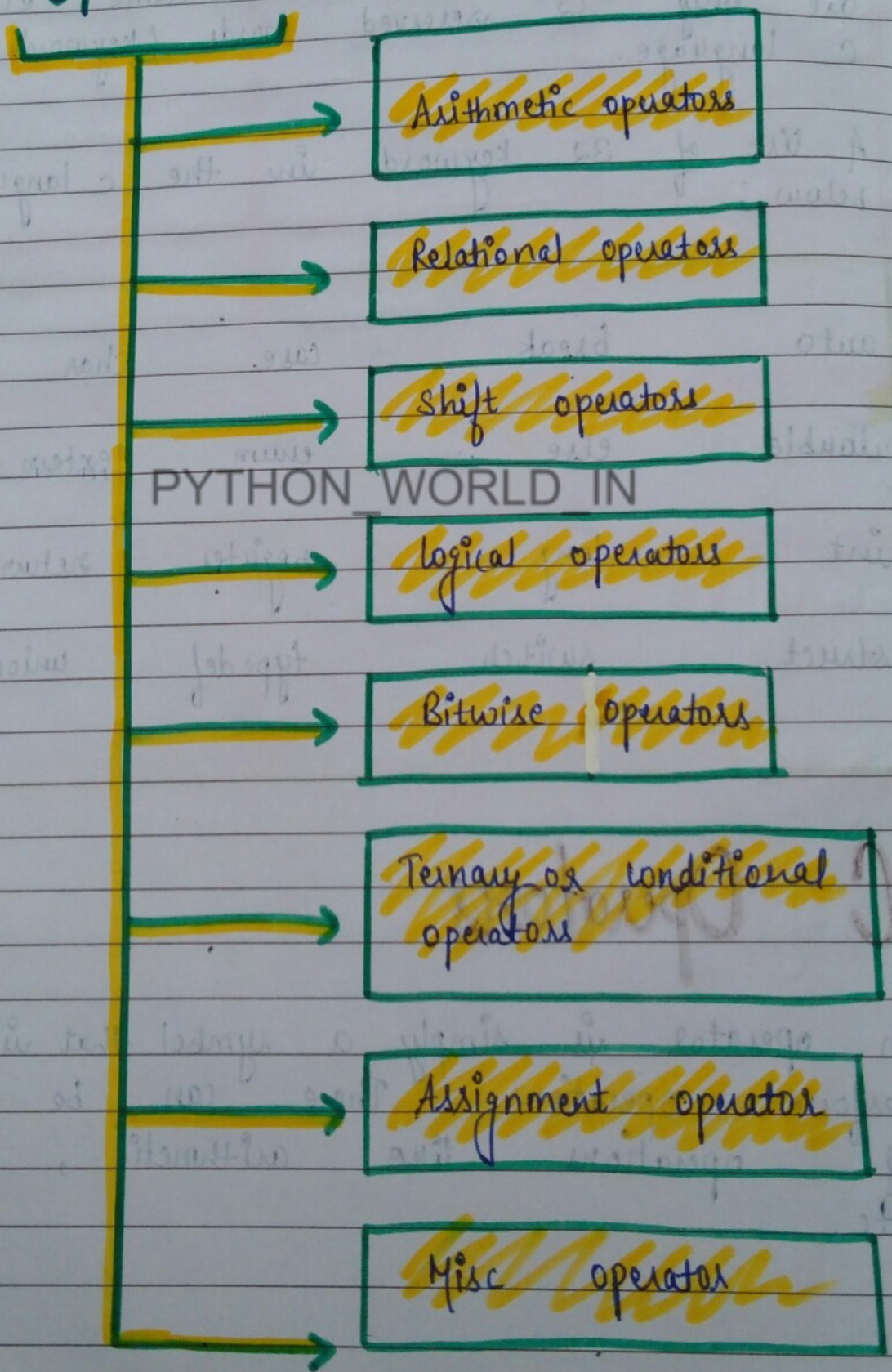
auto	break	case	char	const
double	else	enum	extern	float
int	long	register	return	short
struct	switch	typedef	union	unsigned

C Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc.

There are following types of operators to perform different types of operations in c language.

>>> Operators



The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operator direction to be evaluated; it may be left to right or right to left.

Let's understand the precedence by the example given below:


```
int value = 10 + 20 * 10;
```

The value variable will contain 210 because * (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C operator is given below:

Category	Operator	Associativity
Postfix	() [] -> . ++ --	left to right
unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	left to right

Bitwise XOR	^	left to right
Bitwise OR	|	left to right
Logical AND	&&	left to right
Logical OR	||	left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Comments in C

Comments in c language are used to provide the information about lines of code. It is widely used for documenting code. There are two types of comments in the c language.

1. Single line comments
2. Multi- line comments

Single Line Comments

Single line comments are represented by double slash //. Let's see an example of a single line comment in c.

Source code

```
#include <stdio.h>
int main() {
   // Printing information
   print ("Hello C");
return 0;
}
```

Output :-

```
Hello C
```

Even you can place the comment after the statement. for example :

```
printf ("Hello C") ; // printing information
```

Multi line Comments ↓

Multi-line comments are represented by slash asterisk /* ... */ It can occupy in many lines of code, but it can't be nested. Syntax:

```
/*
code
to be commented
*/
```


Let's see an example of a multi-line comment in C.

source code

```
#include <stdio.h>
int main() {
    /* printing information
     Multi-line comment */
    printf("Hello C");
return 0;
}
```

Output :-

```
Hello C
```

Escape Sequence in C

An escape sequence in C language is a sequence of characters that doesn't represent itself when used inside string literal or character.

It is composed of two or more characters starting with backslash \. for example: \n represents new line.

List of Escape Sequences in C

Escape Sequence	Meaning
\a	Alarm or Beep
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	single Quote
\"	Double Quote
\?	Question Mark

\nnn	octal number
\xhh	hexadecimal number
\0	Null

Escape Sequence Example

source code

```
#include <stdio.h>
int main () {
    int number = 50;
    printf ("you\nare\nlearning\n'c'\nlanguage\n\"Do you
        know C language\"");
return 0;
}
```


Output :-

```
You
are
learning
'c' language
"Do you know C language"
```

Constants in C

A constant is a value or variable that can't be changed in the program, for example: 10, 20, 'a', 3, 3.4, "C programming" etc.

There are different types of constants in C programming.

List of Constants in C

Constant	Example
Decimal Constant	10, 20, 450 etc.
Real or Floating-point constant	10.3, 20.2, 450.6 etc
Octal constant	021, 033, 046 etc.
Hexadecimal constant	0x2a, 0x7b, 0xaa etc.
character Constant	'a', 'b', 'x' etc.
String Constant	"C", "C program", "C in javatpoint" etc.

Two Ways to define Constant in C

There are two ways to define constant in C programming.

1) const keyword
2) #define preprocessor

1) Const keyword in C

The const keyword is used to define constant in C programming.


```
const float PI = 3.14;
```

Now, the value of PI variable can't be changed.

source code

```
# include <stdio.h>
int main () {
   const float PI = 3.14
   printf ("The value of PI is : %f", PI);
   return 0;
}
```

Output :-

```
The value of PI is : 3.140000
```

If you try to change the value of PI, it will render compile time error.

source code

```
#include <stdio.h>
int main() {
const float PI = 3.14;
PI = 4.5;
printf("The value of PI is : %f", PI);
  return 0;
}
```


Output :-

```
Compile Time Error : cannot modify a const object
```

2) #define preprocessor in C

The #define preprocessor is also used to define constant.

C #define

The #define preprocessor directive is used to define constant or micro substitution. It can be use any basic data type.

Syntax :-

```
#define token value
```

Let's see an example of #define to define a constant.

Source code

```
#include <stdio.h>
#define PI 3.14
main() {
   print{"%f", PI);
}
```

Output :-

```
3.140000
```

Let's see an example of #define to create a macro.

Source code

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))
void main() {
  printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
}
```


Output :-

```
Minimum between 10 and 20 is: 10
```

C Fundamental Test 1

1) which of the following is the first operating system developed using C programming language?

a. windows

b. DOS

c. Mac

d. UNIX

2) The C compiler used for UNIX operating system is

a. cc

b. gcc

c. Vc++

d. Borland

3) which of the following is a logical AND operator?

a. ||

b. !

c. &&

d. None of the Above.

4) Which format specifier is used for printing double value?

a. %LF

b. %L

c. %lf

d. None of the above.

5) which of the following statement is used to free the allocated memory space for a program?

a. vanish (var-name);
b. remove (var-name);
c. erase (var-name);
d. free (var-name);

C control Statements

- if-else
- switch and loops
- do while loop in c
- for and while loop
- break and continue
- goto in c
- Type casting
- C control statement Test

if else Statement in C

The if- else statement in c is used to perform the operations based on some specific condition. The operations specified in if block are executed if and only if the given condition is true.

There are the following variants of if statement in c language.

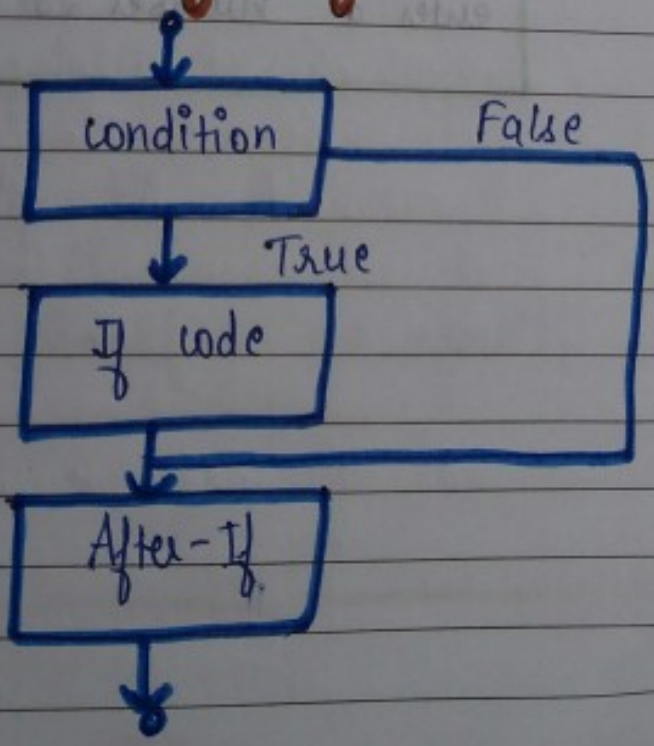
- if statement
- if- else statement
- if else - if ladder
- Nested if

If Statement

The if statement is used to check some given condition and perform some operations depending upon the correctness of that condition. It is mostly used in the scenario where we need to perform the different operations for the different conditions. The syntax of the if statement is given below.

```
if (expression) {
// code to be executed
}
```


Flowchart of if statement in C



Let's see a simple example of c language if statement.

Source code

```
#include <stdio.h>
int main () {
int number = 0;
printf ("Enter a number:");
scanf ("%d", &number);
if (number % 2 == 0) {
printf ("%d is even number", number);
}
return 0;
}
```


Output :

```
Enter a number : 4
4 is even number
enter a number : 5
```

Program :- To find the largest number of the three

Source code

```
#include <stdio.h>
int main()
{
    int a, b, c;
    printf("Enter three numbers?");
    scanf("%d %d %d", &a, &b, &c);
    if (a>b && a>c)
    {
        printf("%d is largest", a);
    }
    if (b>a && b<c)
    {
        printf("%d is largest", b);
    }
    if (c>a && c>b)
    {
        printf("%d is largest", c);
    }
    if (a==b && a==c)
    {
        printf("All are equal");
    }
}
```

Output 1

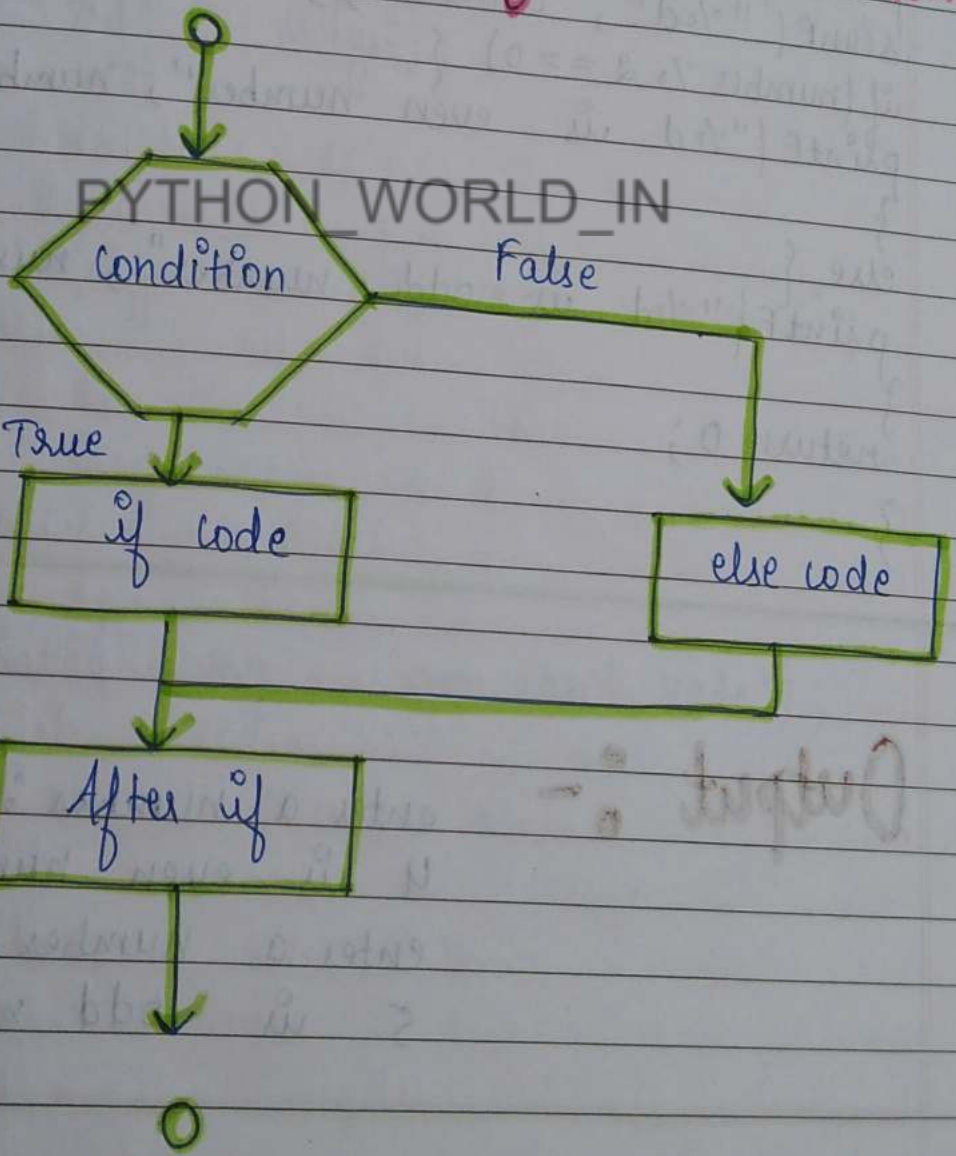
```
Enter the numbers?
12 23 24
30 is largest.
```

If - else Statement

The if-else statement is used to perform two operations for a single condition. The if-else statement is an extension to the if statement using which we can perform two different operations, i.e., one is for the correctness of that condition, and the other is for the correctness of the condition. Here, we must notice that if and else block cannot be executed simultaneously. Using if-else statement is always preferable since it always invokes an otherwise case with every if condition. The syntax of the if-else statement is given below.

```
if (expression){
// code to be executed if condition is true
}else{
// code to be executed if condition is false.
}
```

Flowchart of the if-else Statement in c



Let's see the simple example to check whether a number is even or odd using if-else statement in C language.

Source code

```
#include <stdio.h>
int main() {
int number = 0;
printf("enter a number:");
scanf("%d", &number);
if(number % 2 == 0) {
printF("%d is even number", number);
}
else {
printF("%d is odd number", number);
}
return 0;
}
```


Output :-

```
enter a number : 4
4 is even number
enter a number : 5
5 is odd number
```

Program : To check whether a person is eligible to vote or not.

Source code

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

   int age;
   printf("enter your age?");
   scanf("%d", &age);
   if (age >= 18)
   {

      printf("You are eligible to vote ...");
   }

   else
   {

      printf("Sorry ... you can't vote");
   }

}
```

Output :-

```
Enter your age? 18
you are eligible to vote...
Enter your age? 13
sorry ... you can't vote
```

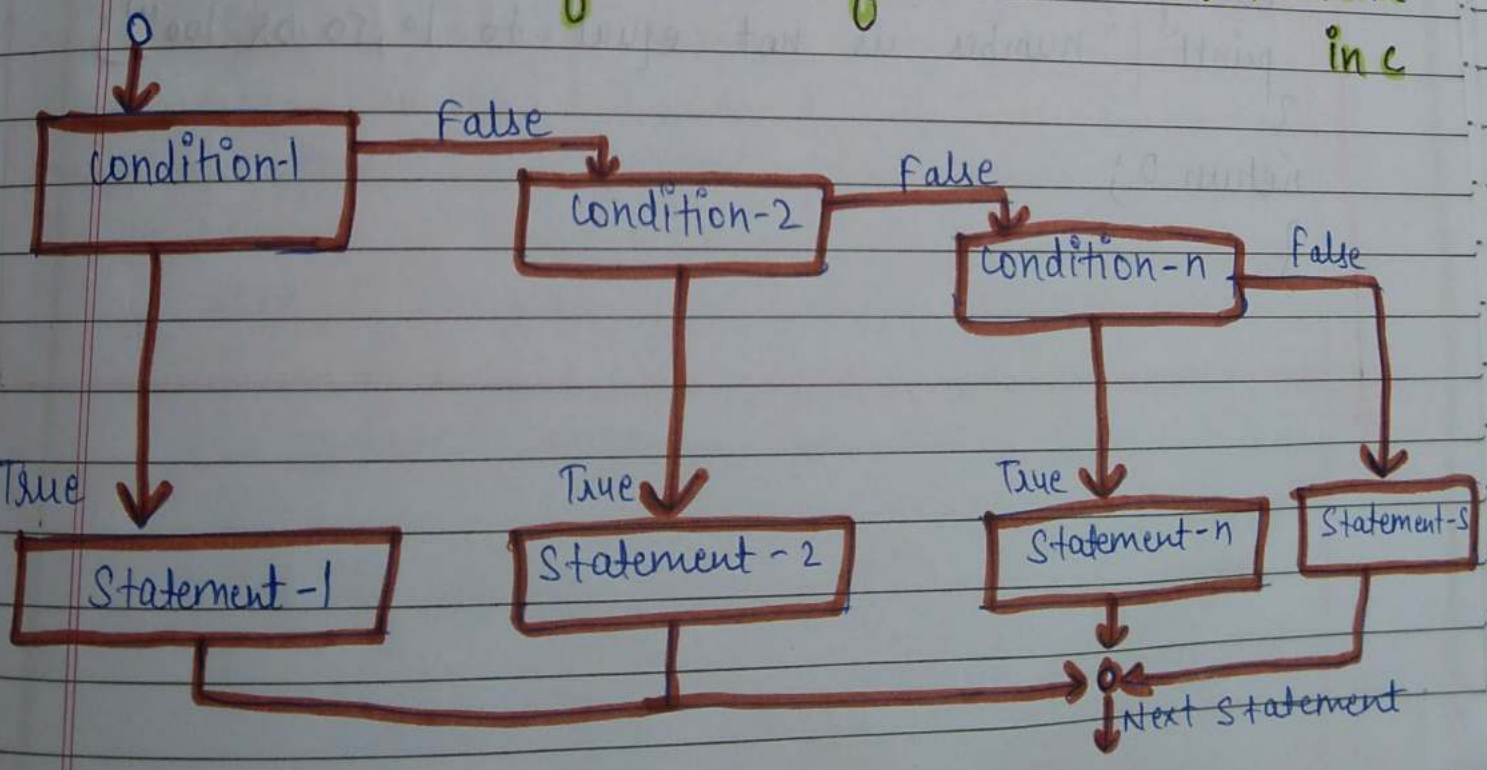
If-else-if ladder Statement

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed. There are multiple else-if blocks possible. It is similar to the switch case statement where the default is executed instead of else block if none of the

cases is matched.

```
if (condition1) {
// code to be executed if condition 1 is true
} else if (condition 2) {
// code to be executed if condition 2 is true
}
 else if (condition 3) {
// code to be executed if condition3 is true
}
...
else {
// code to be executed if all the conditions are false
}
```


Flowchart of else-if ladder Statement in c



The example of an if-else-if statement in C language is given below.

Source code

```
#include <stdio.h>
int main() {
int number = 0;
print("enter a number:");
scanf("%d", &number);
if (number == 10) {
printf("number is equals to 10");
}
else if (number == 50) {
printf("number is equal to 50");
}
else if (number == 100) {
printf("number is equal to 100");
}
else {
printf("number is not equal to 10, 50 OR 100");
}
return 0;
}
```

Output :-

```
enter a number : 4
number is not equal to 10, 50 or 100
enter a number : 50
number is equal to 50
```

Program :-

To calculate the grade of the student according to the specified marks.

Source code

```
#include <stdio.h>
int main()
{
    int marks;
    printf("Enter your marks?");
    scanf("%d", &marks);
    if (marks > 85 && marks <= 100)
    {

        printf("Congrats ! you scored grade A ...");
    }
    else if (marks > 60 && marks <= 85)
    {
        printf("you scored grade B+ ...");
    }
    else if (marks > 40 && marks <= 60)
    {
        printf("you scored grade B ...");
    }
```

```
else if (marks > 30 && marks <= 40)
{
    printf("you stored grade C ...");
}
else
{
    printf("Sorry you are fail...");
}
}
```

Output ↓

```
Enter your marks? 10
Sorry you are fail...
Enter your marks? 40
you stored grade C ...
Enter your marks? 90
you stored grade A ...
```

C Switch Statement

The switch statement in c is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possibles values of a single variable.

The syntax of switch statement in c language is given below:

```
switch(expression){
case value1:
// code to be executed;
break; // optional
case value2:
// code to be executed;
break; // optional
.....

default:
code to be executed if all cases are not matched;
}
```

Rules for Switch statement in C language ...

1) The switch expression must be of an integer or character type.
2) The case value must be an integer or character constant.
3) The case value can be used only inside the switch statement.
4) The break statement in switch case is not must. It is optional. If there is no break statement found in the cases, all the cases will be executed present after the matched case. It is known as fall through the state of C switch statement.

Let's try to understand it by the examples. We are assuming that there are following variables:

```
int x, y, z;
char a, b;
float f;
```

Valid Switch	Invalid Switch	valid case	Invalid case
switch (x)	switch(f)	case 3;	case 2.5;
switch (x>y)	switch(x+2.5)	case 'a';	case x;
switch(a+b-2)		case 1+2;	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

Functioning of Switch case statement

First, the integer expression specified in the switch statement is evaluated. The value is then matched one by one with the constant values given in the different cases. If a match is found, then all the statements specified in that case are executed along with the all the cases present after that case including the default statement. No two cases can have similar values. If the matched case contains a break statement, then

all the cases present after that will be skipped and the control comes out of the switch. otherwise, all the cases following the matched case will be executed.

Let's see a simple example of c language switch statement.

Source code

```
#include <stdio.h>
int main() {
int number = 0;
printf("enter a number:");
scanf("%d", &number);
switch(number){
case 10:
printf("number is equals to 10");
break;
case 50:
printf("number is equal to 50");
break;
case 100:
printf("number is equal to 100");
break;
default:
printf("number is not equal to 10, 50, or 100");
}
return 0;
}
```

Output :

```
enter a number : 4
number is not equal to 10, 50, or 100

enter a number : 50
number is equal to 50.
```

Switch case example 2

Source code

```
#include <stdio.h>
int main()
{
  int x = 10, y = 5;
  switch (x>y && x+y>0)
  {
    case 1:
    printf("hi");
    break;
    case 0:
    printf("bye");
    break;
    default:
    printf("Hello bye");
  }
}
```

Output :- hi

Flowchart of Switch Statement in C

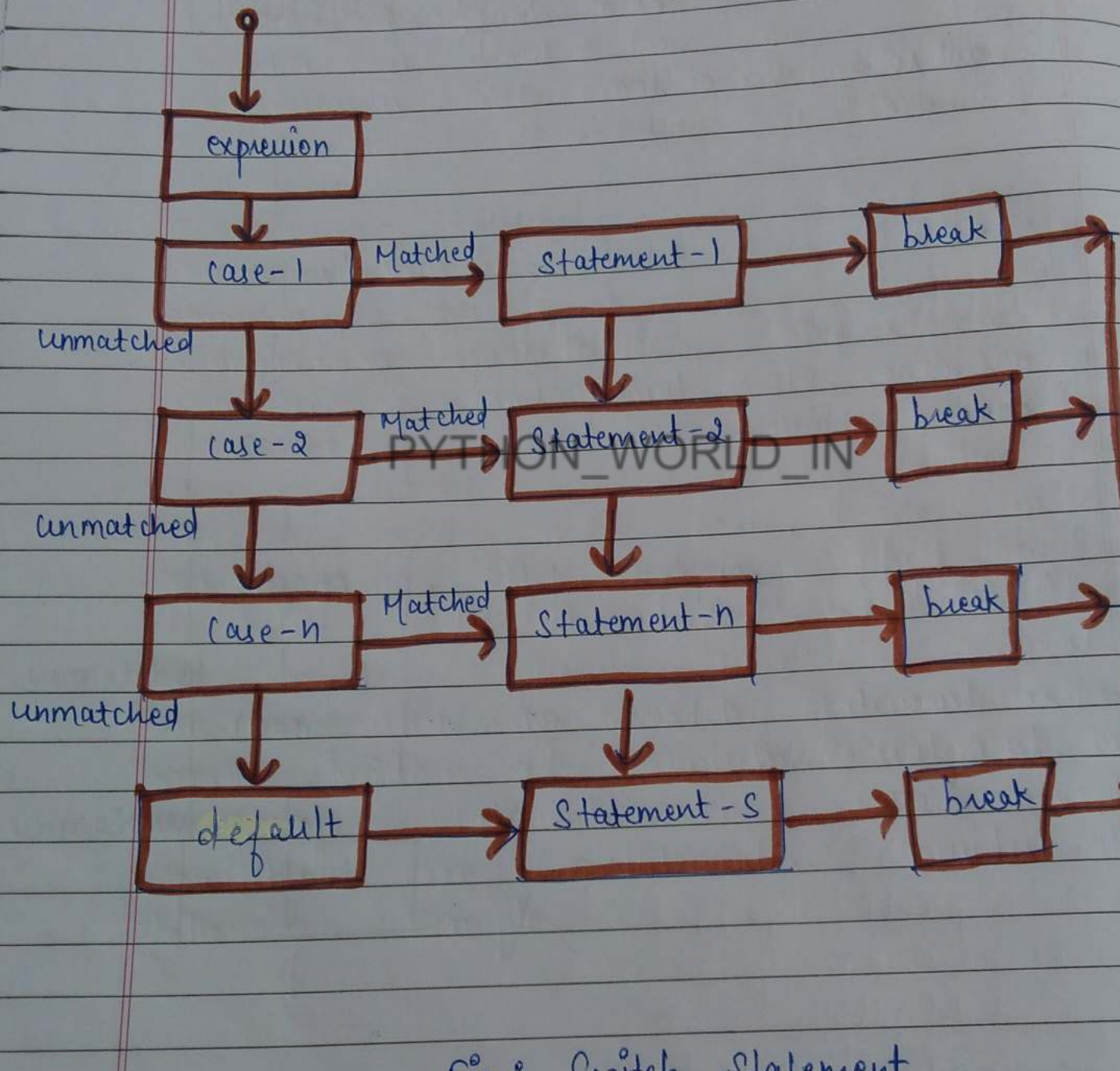


Fig : Switch Statement

C Switch statement is fall-through

In C language, the switch statement is fall through it means if you don't use a break statement in the switch case, all the cases after the matching case will be executed.

Let's try to understand the fall through state of switch statement by the example given below.

Source code

```
#include <stdio.h>
int main(){
int number = 0;
printf("enter a number:");
scanf("%d", &number);
switch(number){
case 10:
printf("number is equal to 10\n");
case 50:
printf("number is equal to 50\n");
case 100:
printf("number is equal to 100\n");
default:
printf("number is not equal to 10,50 or 100");
}
return 0;
}
```

Output :

```
enter a number : 10
number is equal to 10
number is equal to 50
number is equal to 100
number is not equal to 10, 50 or 100
```

```
enter a number : 50
number is equal to 50
number is equal to 100
number is not equal to 10, 50 or 100
```


Nested switch case Statement

We can use many switch statement as we want inside a switch statement. such type of switch statements is called nested case statements. Consider the following example.

```
int main() {
    int i = 10;
    int j = 20;

    switch(i) {
        case 10:
            printf("the value of i evaluated in outer switch: %
                    d\n", i);
        case 20:
            switch (j) {
                case 20:
                    printf("the value of j evaluated in nested
                            switch : %d\n", j);
            }

    }

    printF("Exact value of i is : %d\n", i);
    printF("Exact value of j is : %d\n", j);

    return 0;
}
```


Output :-

```
the value of i evaluated in outer switch: 10
the value of j evaluated in nested switch: 20
Exact value of i is : 10
Exact value of j is : 20
```

C loops

The looping can be defined as repeating the same process multiple times until a specific condition satisfies. There are three types of loops used in the C language. In this part we are discussing looping and looping means repeatation.

Why use loops in C language?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. for examples, if we need to print the first 10 natural numbers then, instead of using the printf statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantage of loops in C

1) It provides code reusability.
2) using loops, we do not need to write the same code again and again.
3) using loops, we can traverse over the elements of data structures (array or linked lists).

Types of C loops

There are three types of loops in c language that is given below:

1) do while
2) while
3) for

do-while loop in C

The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

The syntax of do-while loop in c language is

given below:

```
do {
// code to be executed
} while(condition);
```

While loop in C

The while loop in c is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

The syntax of while loop in c language is given below:

```
while(condition){
// code to be executed
}
```

For loop in C

The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a pre-tested loop. It is better to use for loop if the number of iteration is known in advance.

The syntax of for loop in c language is given below:

```
PYTHON_WORLD_IN
for (initialization ; condition ; incr/decr){
// code to be executed
}
```

do-while loop [Example 1]

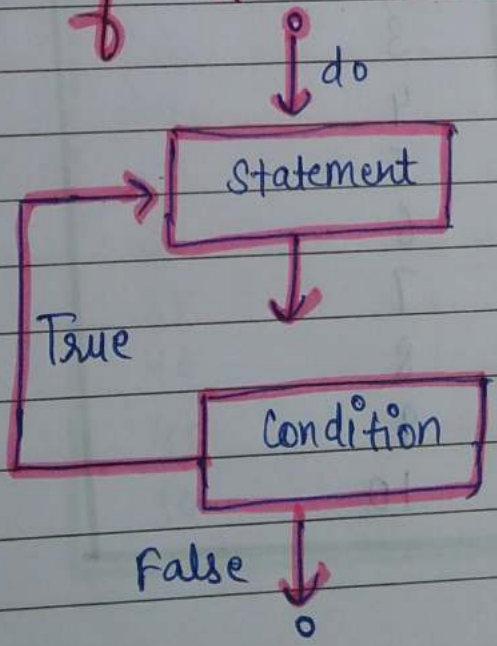
```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    char c;
    int choice, dummy;
    do{
    printf("\n1. Print Hello\n2. Print Javatpoint\n3. Exit\n");
    scanf("%d", &choice);
    switch(choice)
    {
        case 1:
        printf("Hello");
        break;
        case 2:
        printf("Javatpoint");
        break;
        case 3:
        exit(0);
        break;
        default:
        printf("please enter valid choice");
    }
    printf("do you want to enter more?");
    scanf("%d", &dummy);
    scanf("%c", &c);
    }while(c=='y');
}
```

Output :

```
1. Print Hello
2. Print Javatpoint
3. Exit
1
Hello
do you want to enter more?
y

1. Print Hello
2. Print Javatpoint
3. Exit
2
Javatpoint
do you want to enter more?
n
```


Flowchart of do while loop



do while example

There is given the simple program of c language do while loop where we are printing the table of 1.

```
#include <stdio.h>
int main(){
int i=1;
do {
printf("%d\n", i);
i++;
}while(i<=10);
return 0;
}
```


Output :-

```
1
2
3
4
5
6
7
8
9
10
```

Program :- To print table for the given number using do while loop.

```
#include <stdio.h>
int main() {
int i=1, number=0;
printf("Enter a number:");
scanf("%d", &number);
do {
printf("%d\n", (number*i));
i++;
} while(i<=10);
return 0;
}
```


Output :-

```
Enter a number: 5
5
10
15
20
25
30
35
40
45
50
```

```
Enter a number : 10
10
20
30
40
50
60
70
80
90
100
```

Infinitive do while loop

The do-while loop will run infinite times if we pass any non-zero value as the conditional expression.

```
do {
// statement
} while (1);
```

While Loop in C

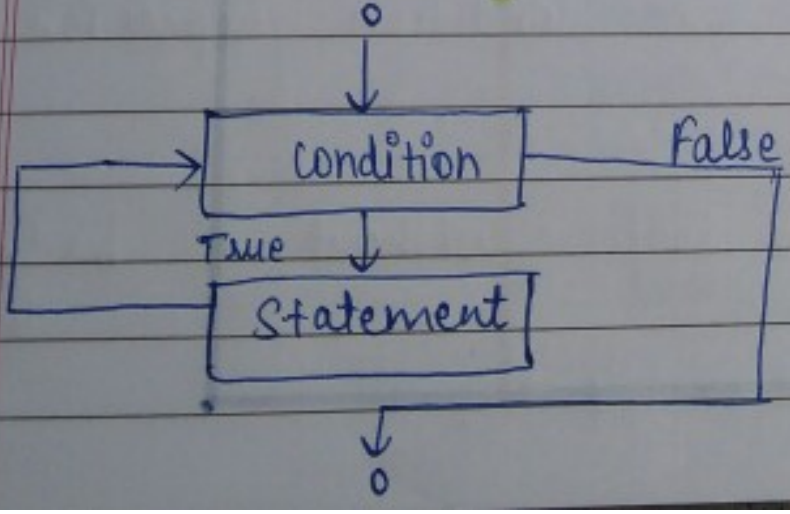
While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

Syntax of while loop in C language

The syntax of while loop in c language is given below:

```
while(condition){
// code to be executed
}
```

Flowchart of while loop in C



Example of the while loop in C

Let's see the simple program of while loop that prints.

```
#include <stdio.h>
int main(){
int i=1;
while (i<=10){
printf("%d\n",i);
i++;
}
return 0;
}
```


Output :-

```
1
2
3
4
5
6
7
8
9
10
```

Program :- To print table for the given number using while loop in c.

```
#include <stdio.h>
int main(){
int i=1, number=0, b=9;
printf("enter a number:");
scanf("%d", &number);
while (i<=10){
printf("%d\n", (number*i));
i++;
}
return 0;
}
```


Output :-

```
Enter a number : 50
50
100
150
200
250
300
350
400
450
500
```

```
Enter a number : 100
100
200
300
400
500
600
700
800
900
1000
```

Properties of while loop

(i) A conditional expression is used to check the condition. The statements defined inside the while loop will repeatedly execute until the given condition fails.

(ii) The condition will be true if it returns 0. The condition will be false if it returns any non-zero number.

(iii) In while loop, the condition expression is compulsory.

(iv) Running a while loop without a body is possible

(v) we can have more than one conditional expression in while loop.

(vi) If the loop body contains only one statement, then the { } braces are optional.

Example 1

```
#include <stdio.h>
void main()
{
  int j = 1;
  while (j+2, j<=10)
  {
    printf("%d", j);
  }
  printf("%d", j);
}
```


Output

```
3 5 7 9 11
```

Example 2

```
#include <stdio.h>
void main()
{
  while()
   {
    printf("hello Javatpoint");
   }
}
```

Output :

compile time error : while loop can't be empty

Example 3

```
#include <stdio.h>
void main()
{
  int x = 10, y = 2;
  while (x+y-1)
  {
    printf("%d %d", x--, y--);
  }
```

Output :- infinite loop

Infinitive while loop in C

If the expression passed in while loop results in any non-zero value then the loop will run the infinite number of times.

```
while (1) {
// statement
}
```


For loop in C

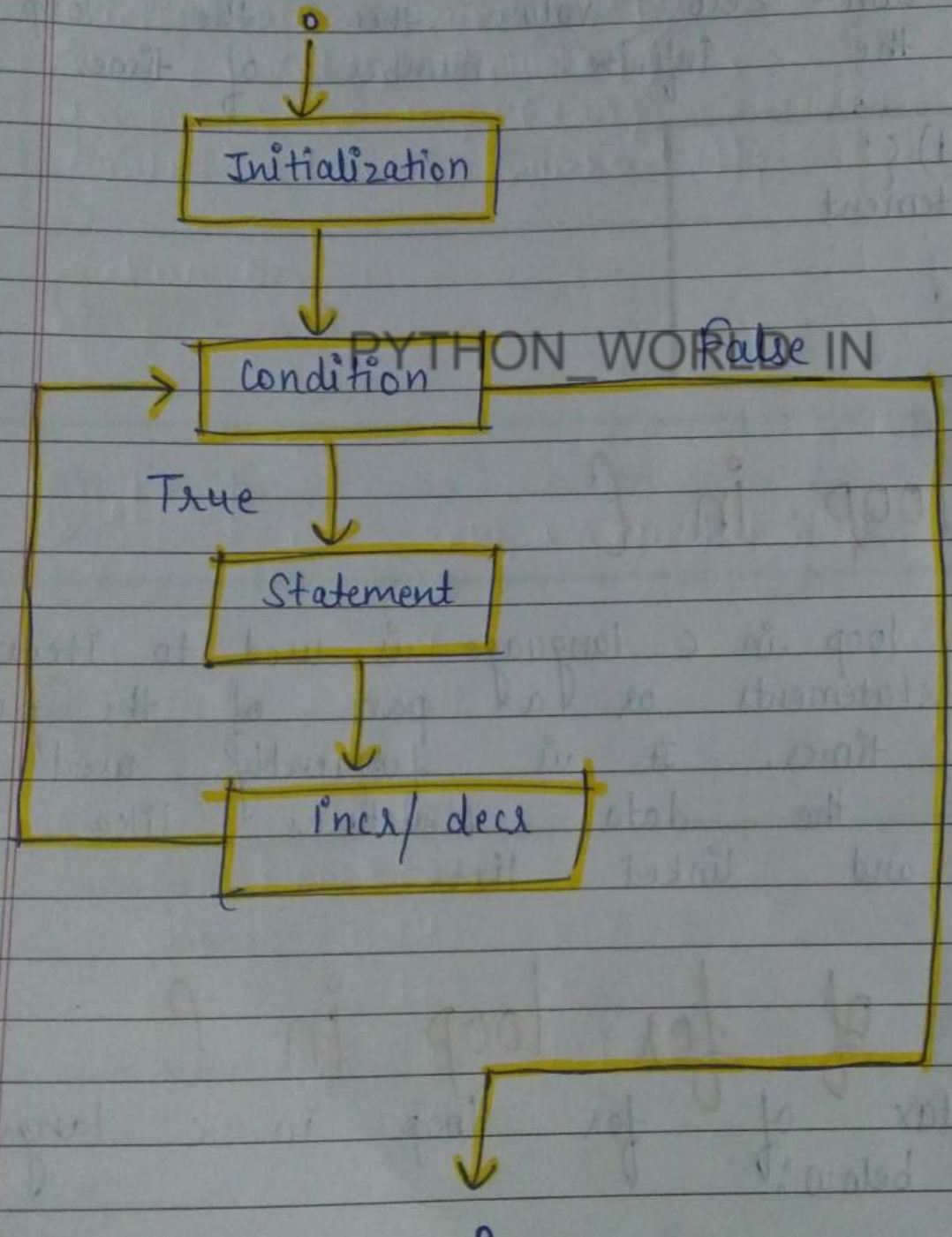
The for loop in c language is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like the array and linked list.

Syntax of for loop in C

The syntax of for loop in c language is given below:

```
For (Expression 1; Expression 2; Expression 3) {
// code to be executed
}
```

Flowchart of for loop in C



C for loop Examples

Let's see the simple program of for loop that print.

```
#include <stdio.h>
int main() {
int i=0;
for (i=1; i<=10; i++) {
printf("%d\n",i);
}
return 0;
}
```


Output :-

```
1
2
3
4
5
6
7
8
9
10
```

Program :- To print table for the given number using c for loop.

```
#include <stdio.h>
int main() {
int i=1, number = 0;
printf("Enter a number :");
scanf("%d", &number);
for(i=1; i<=10; i++){
printf("%d\n", (number * i));
}
return 0;
}
```


Output :-

```
Enter a number : 2
2
4
6
8
10
12
14
16
18
20
```

```
Enter a number : 1000
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
```

Properties of Expression 1

(i) The expression represents the initialization of the loop variable.

(ii) We can initialize more than one variable in Expression 1.

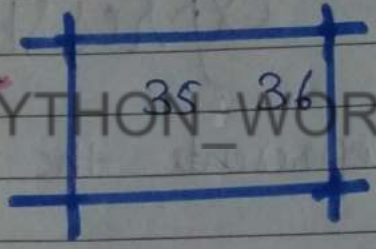
(iii) Expression 1 is optional.

(iv) In C, we can not declare the variables in expression 1. However, it can be an exception in some compilers.

Example 1

```
#include <stdio.h>
int main()
{
    int a, b, c;
    for(a=0, b=12, c=23; a<2; a++)
    {
        printf("%d", a+b+c);
    }
}
```

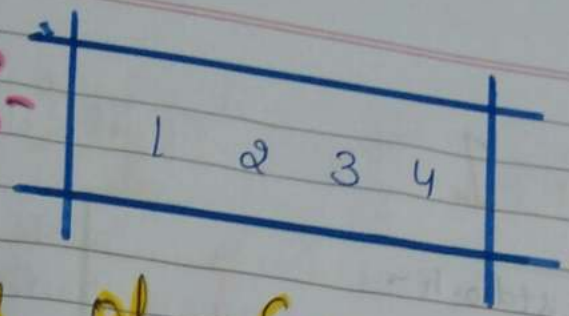
Output :-



Example 2

```
#include <stdio.h>
int main()
{
    int i = 1;
    for(; i<5; i++)
    {
        printf("%d", i);
    }
}
```

Output :-



Properties of Expression 2

(i) Expression 2 is a conditional expression. It checks for a specific condition to be satisfied. If it is not, the loop is terminated.

(ii) Expression 2 can have more than one condition. However, the loop will iterate until the last condition becomes false. other conditions will be treated as statements.

(iii) Expression 2 is optional.

(iv) Expression 2 can perform the task of expression 1 and expression 3. That is, we can initialize the variable as well as update the loop variable in expression 2 itself.

(v) We can pass zero or non-zero value in expression 2. However, in C, any non-zero value is true, and zero is false by default.

Example 1

```
#include <stdio.h>
int main()
{
    int i;
    for(i=0; i<=4; i++)
    {
      printf("%d", i);
    }
}
```

Output :- 0 1 2 3 4

Example 2

```
#include <stdio.h>
int main()
{
    int i, j, k;
    for (i=0, j=0, k=0; i<4, k<8, j<10; i++)
    {
      printf("%d %d %d\n", i, j, k);
      j++=2;
      k++=3;
    }
}
```

Output :

```
000
123
246
369
4812
```

Example 3

```
#include <stdio.h>
int main()
{
    int i;
    for (i=0;;i++)
    {
      printf("%d", i);
    }
}
```


Output :- infinite loop

Properties of Expression 3

(i) Expression 3 is used to update the loop variable.
(ii) We can update more than one variable at the same time.
(iii) Expression 3 is optional.

Example 1

```
#include <stdio.h>
void main()
{
    int i=0, j=2;
    for(i=0; i<5; i++, j=j+2)
    {
        printf("%d %d\n", i, j);
    }
}
```


Output :-

```
0 2
1 4
2 6
3 8
4 10
```

Loop body :-

The braces { } are used to define the scope of the loop. However, if the loop contains only one statement then we don't need to use braces.

Loop without a body is possible.

The braces work as a block separator i.e., the value variable declared inside for loop is valid only for that block and not outside. consider the following example.

```
# include <stdio.h>
void main ()
{
  int i;
  for (i=0; i<10; i++)
  {
    int i=20;
    printf("%d", i);
  }
}
```


Output :-

```
20 20 20 20 20 20 20 20 20 20
```

Infinitive for loop in C

To make a for loop infinite, we need not give any expression in the syntax. Instead of that, we need to provide two semicolons to validate the syntax of the for loop. This will work as an infinite for loop.

```
#include <stdio.h>
void main()
{
   for(;;)
   {
      printf("welcome to Javatpoint");
   }
}
```


If you run this program, you will see above statement infinite times.

break statement

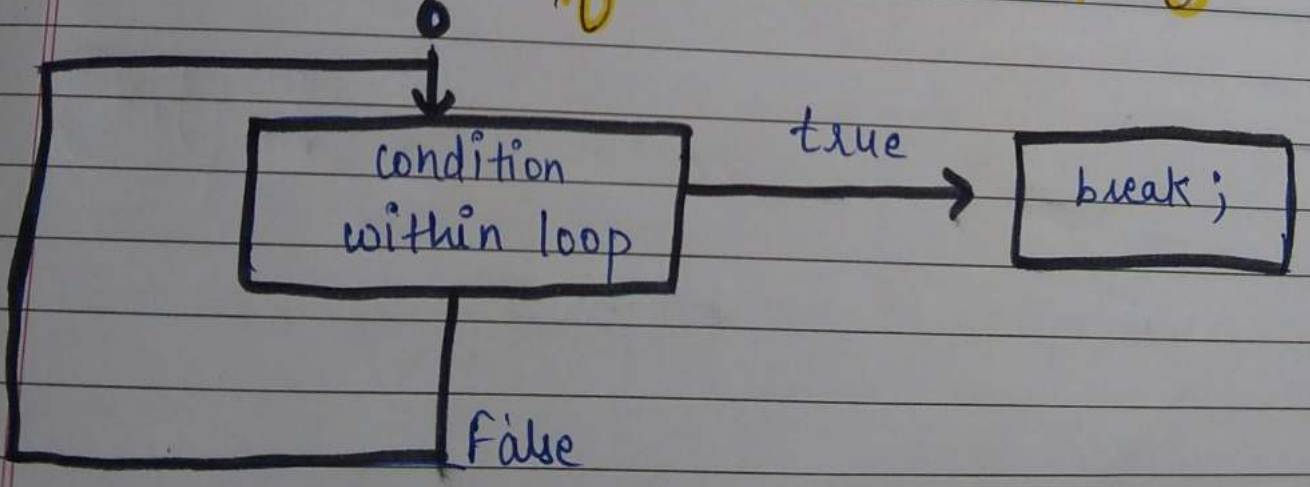
The break is a keyword in C which is used to bring the program control out of the loop. The break statement is used inside loops or switch statement. The break statement breaks the loop one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. The break statement in C can be used in the following two scenarios:

1. with switch case
2. with loop

Syntax :-

```
// loop or switch case
break;
```

Flowchart of break in C



Example

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int i;
    for(i=0; i<10; i++)
    {
      printf("%d", i);
      if(i==5)
      break;
    }
    printf("came outside of loop i = %d", i);
}
```


Output :

```
0 1 2 3 4 5 came outside of loop i = 5
```

C break statement with the Nested loop ...

In such case, it breaks only the inner loop, but not outer loop.

```
#include <stdio.h>
int main() {
int i=1, j=1; // initializing a local variable
for (i=1; i<=3; i++) {
for (j=1; j<=3; j++) {
printf("%d %d\n", i, j);
if (i==2 && j==2) {
break; // will break loop of j only
}
} // end of for loop
return 0;
}
```


Output :-

```
1 1
1 2
1 3
2 1
2 2
3 1
3 2
3 3
```

As you can see the output on the console, 2 3 is not printed because there is a break statement after printing i==2 and j==2. But 3 1, 3 2 and 3 3 are printed because the break statement is used to break the inner loop only

break statement with while loop

Conider the following example to use break statement inside while loop.


```
#include <stdio.h>
void main()
{
  int i = 0;
  while(1)
  {
    printf("%d", i);
    i++;
    if (i==10)
    break;
  }
  printf("came out of while loop");
}
```

Output ↓

```
0 1 2 3 4 5 6 7 8 9 came out of while loop.
```

break statement with do-while loop

consider the following example to use the break statement with a do-while loop.

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

  int n = 2, i, choice;
  do
  {
    i=1;
    while (i<=10)
    {
      printf("%d X %d = %d\n", n, i, n*i);
      i++;
    }
    printf("do you want to continue with the table of %d, enter any non-zero value to continue.", n+1);
    scanf("%d", &choice);
  if (choice==0)
  {
    break;
  }
```

```
        n++;
    } while(1);
}
```

Output :

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20

do you want to continue with the table of 3,
enter any non-zero value to continue.1

3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
```

do you want to continue with the table of 4, enter any non-zero value to continue.0.

Continue Statement in C

The continue statement in c language is used to bring the program control to the beginning of the loop. The continue statement skips some lines of code inside the loop and continues with the next iteration. It is mainly used for a condition so that we can skip some code for a particular condition.

Syntax :-

```
// loop statements
continue;
// some lines of the code which is to be skipped
```

Continue Statement example 1

```
#include <stdio.h>
void main()
{
  int i=0;
  while(i!=10)
  {
    printf("%d", i);
    continue;
    i++;
  }
}
```


Output :-

```
infinite loop
```

Continue Statement example 2

```
#include <stdio.h>
int main(){
int i=1; // initializing a local variable
// starting a loop from 1 to 10.
for (i=1; i<=10; i++){
if (i==5)
{// if value of i is equal to 5, it will continue the loop
continue;
}
printf("%d\n",i);
} // end of for loop
return 0;
}
```


Output :-

```
1
2
3
4
5
6
7
8
9
10
```

As you can see, 5 is not printed on the console because loop is continued at i==5.

Continue Statement with inner loop

In such case, c continue statement continues only inner loop, but not outer loop.

```
#include <stdio.h>
int main(){
int i=1, j=1; // initializing a local variable
for(i=1; i<=3; i++){
for(j=1; j<=3; j++){
if(i==2 && j==2){
continue; // will continue loop of j only
}
printf("%d %d\n",i,j);
}
} // end of for loop
return 0;
}
```


Output :-

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

As you can see, 2 2 is not printed on the console because inner loop is continued at i==2 and j==2.

C goto Statement

The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statement can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement. However, using goto is avoided these days since it makes the program less readable and complecated.

Syntax :

```
label:
// some part of the code;
goto label;
```

goto example

Let's see a simple example to use goto statement in c language.

```
#include <stdio.h>
int main()
{
  int num, i=1;
  printf("Enter the number whose table you want to print?");
  scanf("%d", &num);
  table:
  printf("%d x %d = %d\n", num, i, num*i);
  i++;
  if(i<=10)
  goto table;
}
```


Output :

```
Enter the number whose table you want to print? 10
10 x 1 = 10
10 x 2 = 20
10 x 3 = 30
10 x 4 = 40
10 x 5 = 50
10 x 6 = 60
10 x 7 = 70
10 x 8 = 80
10 x 9 = 90
10 x 10 = 100
```

When should we use goto?

The only continue in which using goto is preferable is when we need to break the multiple loops using a single statement at the same time. consider the following example.

```
# include <stdio.h>
int main()
{
 int i, j, k;
 for (i=0; i<10; i++)
 {

  for (j=0; j<5; j++)
  {
   for (k=0; k<3; k++)
   {
    printF("%d %d %d\n", i, j, k);
    if (j==3)
    {
     goto out;
    }
   }
  }
 }
 out:
 printF("came out of the loop");
}
```

Output :-

```
0 0 0
0 0 1
0 0 2
0 1 0
0 1 1
0 1 2
0 2 0
0 2 1
0 2 2
0 3 0
came out of the loop
```

Type Casting in C

Typecasting allows us to convert one data type into other. In C language, we use cast operator for typecasting which is denoted by (type).

Syntax ↓

```
(type)value;
```


Note :- It is always recommended to convert the lower value to higher for avoiding data loss.

Without Type Casting

```
int f = 9/4;
printf("f : %d\n", f); //output : 2
```

With Type Casting

```
float f = (float) 9/4;
printf("f : %f\n", f); // output : 2.250000
```

Type Casting example

Let's see a simple example to cast int value into the float.


```
#includ <stdio.h>
int main() {
float f = (float) 9/4;
printf("f : %f\n", f);
return 0;
}
```

Output :-

```
f : 2.250000
```

C Control Statement Test

1) which data type cannot be checked in switch case statement?

a. enum

b. character

c. integer

d. float

2) How many times "JavaTpoint" is printed?

```
#include <stdio.h>
int main()
{
  int x;
  for( x = -1; x <= 10; x++)
  {
    if (x<5)
      continue;
    else
      break;
    printf ("JavaTpoint");
  }
  return 0;
}
```

a. 10 times

b. 11 times

c. 0 times

d. infinite times

3) How many times while loop is executed if a short int is 2 byte wide?

```
#include <studio.h>
int main()
{
  int i=1;
  while(i<=155)
  {
    printf("%c %d\n", i, i);
    i++;
  }
  return 0;
}
```


a. 154 times

b. 155 times

c. 156 times

d. infinite times

4) Which statement is correct about the below program?

```
# include <stdio.h>
int main()
{
  int i=8, j=24;
  if (i=8) && if (j=24)
    printF(" welcome programmer");
  return 0;
}
```

a. Welcome programmer

b. Error: undeclared identifier if

c. Error: Expression syntax

d. no output

5) Find out the error, if any in the below program?

```
# include <stdio.h>
int main()
{
  int j=1;
  switch(j)
  {
  printF("Hello programmer!");
  case 1:
```

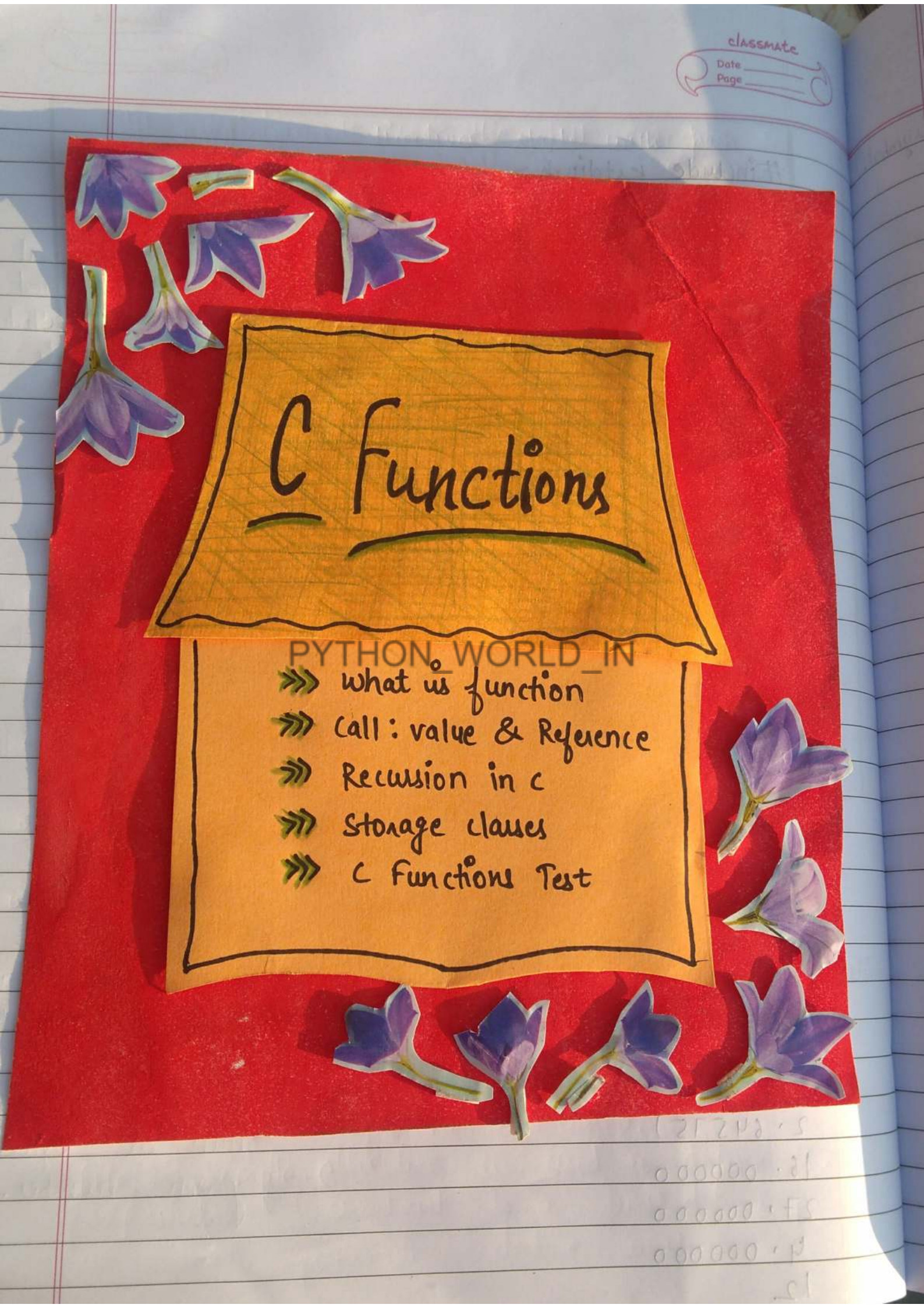
```
        printf("case1");
        break;
    case 2:
        printf("case 2");
        break;
    }
return 0;
}
```

a. No errors in program and prints "case 1"

b. Error: invalid printf statement after switch statement.

c. Errors: No default specified

d. None of the above


C Functions
PYTHON_WORLD_IN
what is function
Call : value & Refuence
Recursion in c
Storage classes
C Functions Test

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 the 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.

(i) By using functions, we can avoid rewriting same logic/code again and again in a program.

(ii) we can call C functions any number of times in a program and from any place in a program.

(iii) we can track a large C program easily when it is divided into multiple functions.

(iv) Reusability is the main achievement of c functions.

(v) However, Function calling is always a overhead in a c program.

Function Aspects

There are three aspects of a c function.

1) 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.

2) 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.

3) 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	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;}

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

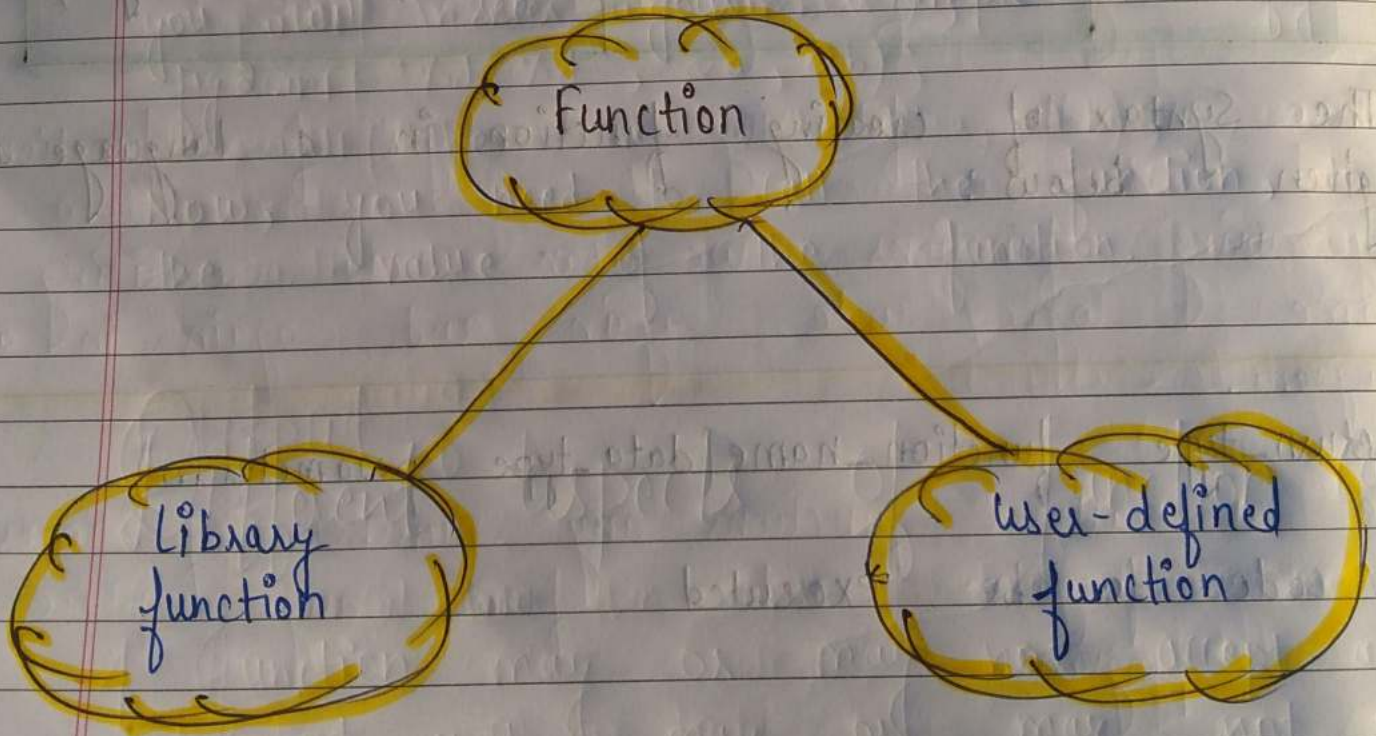
```
return_type function_name(data_type parameter...)
{
// code to be executed
}
```

Types of Functions

There are two types of functions in c programming

1) Library functions: 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: 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 does not return any 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;
}
```

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 function that returns int value from the function. Now you will see your example in fig 1 and fig 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.

(i) function without arguments and without return value.

(ii) function without arguments and with return value.

(iii) function with arguments and without return value.

(iv) function with arguments and with return value.

Example for Function without the argument and return value ...

Example 1

```
#include <stdio.h>
void printName();
void main()
{
  printf("Hello");
  printName();
}
void printName()
{
printf("JavaTpoint");
}
```

Output :-

```
Hello JavaTpoint
```

Example 2

```
#include <stdio.h>
void sum();
void main()
{
  printf("\n Going to calculate the sum of two number:");
  sum();
}
void sum()
{
  int a,b;
  printf("\n Enter two numbers");
  scanf("%d %d", &a, &b);
  printf("The sum is %d", a+b);
}
```


Output :-

```
Going to calculate the sum of two number:
Enter two numbers 10
24
The sum is 34
```

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

Example 1

```
#include <stdio.h>
int sum();
void main()
{
   int result;
   printf("\n Going to calculate the sum of two numbers:");
   result = sum();
   printf("%d", result);
}
int sum()
{
   int a, b;
   printf("\n Enter two numbers");
   scanf("%d %d", &a, &b);
   return a+b;
}
```

Output :

```
Going to calculate the sum of two numbers:
Enter two number 10
24
The sum is 34.
```

Example 2 :- To calculate the area of the square.

```
#include <stdio.h>
int sum();
void main()
{
  printf("Going to calculate the area of the square\n");
  float area = square();
  printf("The area of the square : %f\n", area);
}
int square()
{
  float side;
  printf("Enter the length of the side in meters:");
  scanf("%f", &side);
  return side * side;
}
```

Output ↓

```
Going to calculate the area of the square
Enter the length of the side in meters:
The area of the square : 100.000000
```

Example for function with argument and without return value ...

Example 1

```
# include <stdio.h>
void sum (int, int);
void main()
{
  int a, b, result;
  printf("\n Going to calculate the sum of two numbers:");
  printf("\n Enter two numbers:");
  scanf("%d %d", &a, &b);
  sum(a,b);
}
void sum(int a, int b)
{
  printf("\n The sum is %d", a+b);
}
```

Output :

```
Going to calculate the sum of two numbers:
Enter two numbers to
24
The sum is 34.
```

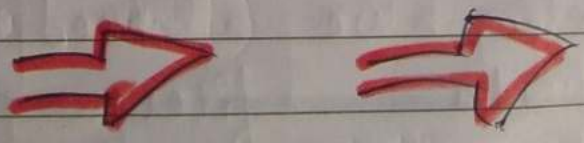
Example 2 :- program to calculate the average of five numbers.


```
#include <stdio.h>
void average (int, int, int, int, int);
void main ()
{
   int a, b, c, d, e;
   printf ("\n Going to calculate the average of five numbers:");
   printf ("\n Enter five numbers:");
   scanf ("%d %d %d %d %d", &a, &b, &c, &d, &e);
   average (a, b, c, d, e);
}
void average (int a, int b, int c, int d, int e)
{
   float avg;
   avg = (a+b+c+d+e) /5;
   printf ("The average of given five numbers: %f", avg);
}
```

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

```
#include <stdio.h>
int sum (int, int);
void main()
{
  int a, b, result;
  printf("\n Going to calculate the sum of two numbers");
  printf("\n Enter two numbers :");
  scanf("%d %d", &a, &b);
  result = sum (a, b);
  printf("\n The sum is : %d", result);
}
int sum(int a, int b)
{
  return a+b;
}
```


Output ↓

```
Going to calculate the sum of two numbers :
Enter two number : 10
20
The sum is : 30
```

Example 2 :- program to check whether a number is even or odd.

```
#include <stdio.h>
int even_odd (int);
void main ()
{
int n, flag = 0;
printf ("\n Going to check whether a number is even or
odd");
printf ("\n enter the number :");
scanf ("%d", &n);
flag = even_odd (n);
if (flag == 0)
{
  printf ("\n The number is odd");
}
else
{
printf ("\n The number is even");
}
}
int even_odd (int n)
{
  if (n % 2 == 0)
  {
    return 1;
  }
  else
  {
```

```
    return 0;
  }
}
```

Output ↓

```
Going to check whether a number is even or odd
Enter the number : 100
The number is even.
```

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 opuations. For example, printf is a library function used to print on the consol. 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.

The list of mostly used header files is given in the following table.

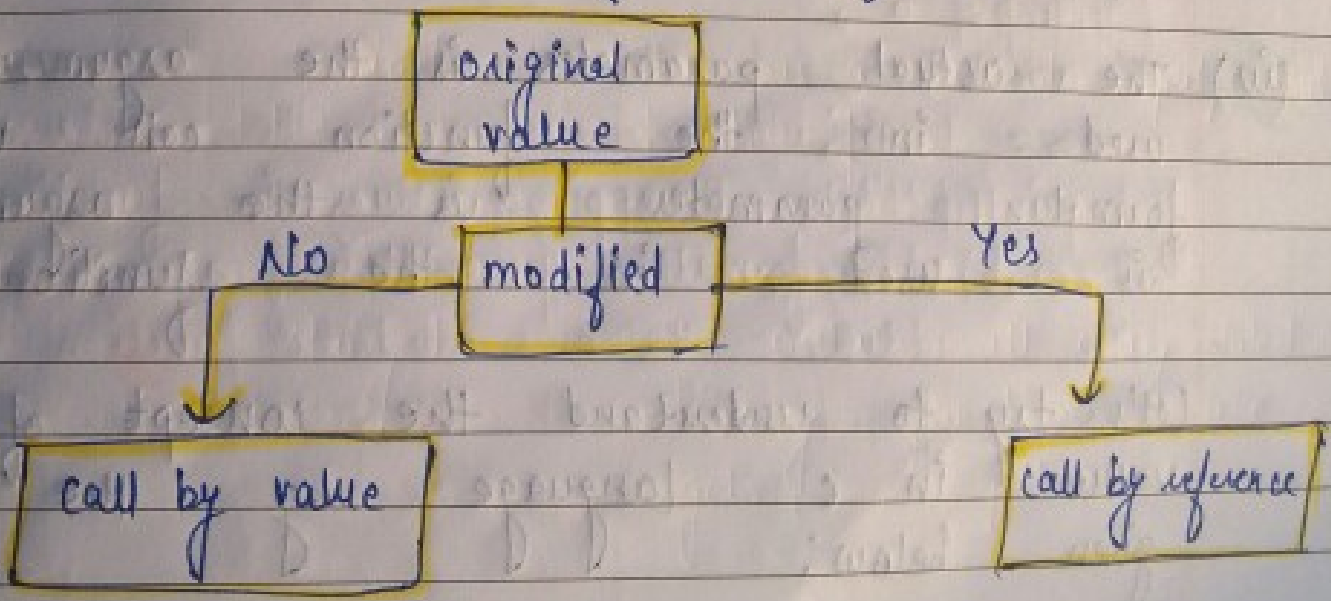
SN	Header file	Description
1.	stdio.h	This is a standard input/output header file. It contains all the library functions regarding standard input/ output.
2.	conio.h	This is a console input/output header file.
3.	string.h	It contains all string related library functions like gets(), puts(), etc.

4.	stdlib.h	This header file contains all the general library functions like malloc(), calloc(), exit(), etc.
5.	math.h	This header file contains all the math operations related functions like sqrt(), pow(), etc.
6.	time.h	This header file contains all the time related functions.
7.	ctype.h	This header file contains all character handling functions.
8.	signal.h	All the signal handling functions are defined in this header file.
9.	stdarg.h	Variable argument functions are defined in this header file.
10.	setjmp.h	This file contains all the jump functions.

11.	locale.h	This file contains locale functions.
12.	errno.h	This file contains error handling functions.
13.	assert.h	This file contains diagnostics functions.

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.



Let's understand call by value and call by reference in c language one by one.

Call by value in C

(i) 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.

(ii) In call by value method, we can not modify the value of the actual parameter by the formal parameters.

(iii) 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.

(iv) 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.

Let's try to understand the concept of call by value in c language by the example given below:

```c
#include <stdio.h>
void change (int num){
    printF (" Before adding value inside function num = %d\n",
num);
    num = num +100;
    printF (" After adding value inside function num = %d
\n", num);
}
int main() {
    int x = 100;
    printF(" Before function call x = %d\n", x);
    change (x); // passing value in function
    printF (" After function call x = %d\n", x);
return 0;
}
```


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

Call by value Example :- Swapping the values of the two variables ...

```
#include <stdio.h>
void swap (int, int); // prototype of the function
int main()
{
  int a = 10;
  int b = 20;
  printf("Before swapping the values in main a = %d, b = %d\n",
a,b); // printing the value of a and b in main.
  swap (a,b);
  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
}
void swap (int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
  printf("After swapping values in function a = %d, b = %d
\n", a,b); // formal parameters, a = 20, b = 10.
```

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

(i) In call by reference, the address of the variable is passed into the function call as the actual parameter.

(ii) The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.

(iii) 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.

Consider the following example for the call by reference.

```
#include <stdio.h>
void change(int* num) {
  printf("Before adding value inside function num=%d\n",
*num);
  (*num) += 100;
  printf("After adding value inside function num=%d\n",
*num);
}
int main() {
   int x = 100;
   printf("Before function call x=%d\n", x);
   change(&x); // passing reference in function
   printf("After function call x=%d\n", x);
return 0;
}
```


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.

```
#include <stdio.h>
void swap (int*, int*); // prototype of the function
int main ()
{
   int a = 10;
   int b = 20;
   printf ("Before swapping the values in main a = %d, b = %d \n", a, b); // printing the value of a and b in main
   swap (&a, &b);
   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
}
void swap (int* a, int* b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
   printf ("After swapping values in function a = %d, b = %d \n", *a, *b); // Formal parameters, a = 20, b = 10
}
```

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 is 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.

Recursion in C

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursive or recursion involves several numbers of the recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however, it is difficult to understand.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For example, recursion may be applied to sorting, searching and traversal problems.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

In the following example, recursion is used to calculate the factorial of a number.

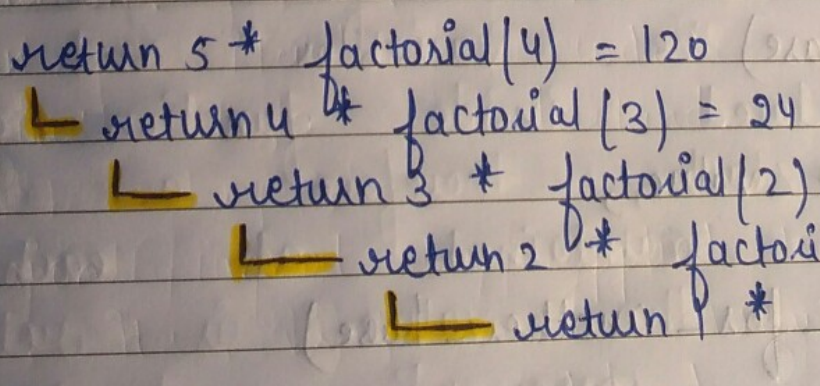
```
#include <stdio.h>
int fact(int);
int main()
{
    int n, f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d", &n);
    f = fact(n);
    printf("factorial = %d", f);
}
int fact(int n)
{
    if (n==0)
    {
        return 0;
    }
    else if (n==1)
    {
```

```
        return 1;
    }
    else
    {
    return n* fact(n-1);
    }
}
```

Output ↓

Enter the number whose factorial you want to calculate?5
factorial = 120

We can understand the above program of the recursive method call by the figure given below:



1 * 2 * 3 * 4 * 5 = 120

Recursive Function

A recursive function performs the tasks by dividing it into the subtasks. There is a termination condition defined in the function which is satisfied by some specific subtask. After this, the recursion stops and the final result is returned from the function.

The case at which the function doesn't recur is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case. All the recursive functions can be written using this format.

Pseudocode for writing any recursive function is given below.

```
if (test_for_base)
{
  return some_value';
}
else if (test_for_another_base)
{
  return some_another_value';
}
else
```

```
{
    // statements;
    recursive call;
}
```

Example of recursion in C

Let's see an example to find the nth term of the fibonacci series.

```
#include <stdio.h>
int fibonacci(int);
void main()
{
    int n, f;
    printf("Enter the value of n?");
    scanf("%d", &n);
    f = fibonacci(n);
    printf("%d", f);
}
int fibonacci (int n)
{
    if (n==0)
    {
    return 0;
    }
    else if (n==1)
    {
        return 1;
```

```
    }
    else
    {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

Output ↓

```
Enter the value of n? 12
144
```


Memory allocation Of Recusive method

Each recursive call creates a new copy of that method in the memory. once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each

recursive call. Therefore, we need to maintain the stack and track the values of the variables defined in the stack.

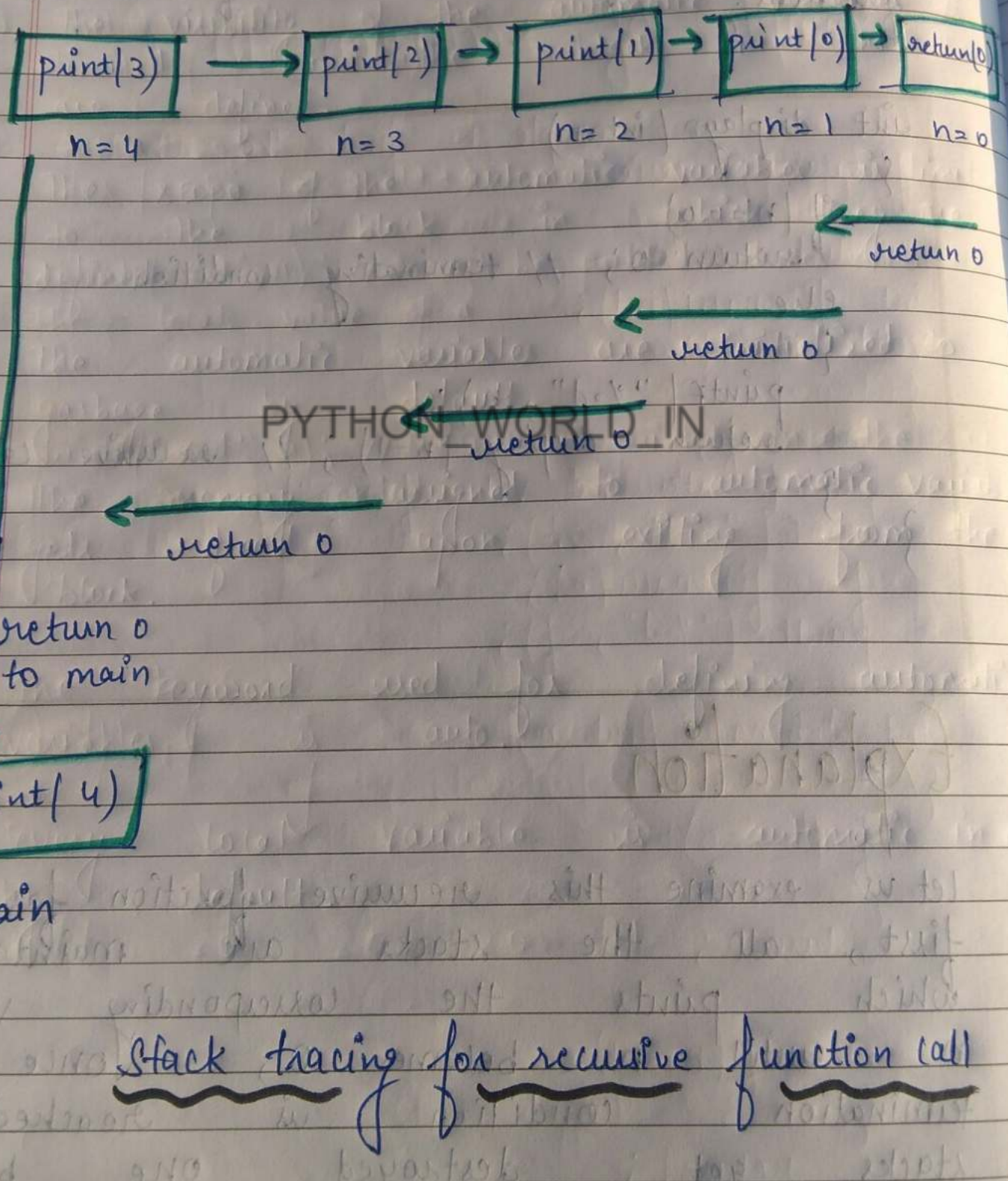
Let us consider the following example to understand the memory allocation of the recursive functions.

```
int display (int n)
{
   if (n==0)
     return 0; // terminating condition
   else
   {
     printf("%d",n);
     return display (n-1); // recursive call
   }
}
```


Explanation

Let us examine this recursive function for n = 4. First, all the stacks are maintained which prints the corresponding value of n until n becomes 0, Once the termination condition is reached, the stacks get destroyed one by one

by returning 0 to its calling stack. Consider the following this image for more information regarding the stack trace for the recursive functions.



Stack tracing for recursive function call

Storage Classes in C

Storage clases in c are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage clases in c.

- >>> Automatic
- >>> External
- >>> static
- >>> Register

Storage clases	Storage Place	Default value	scope
auto	RAM	Garbage value	Local
extern	RAM	zero	Global
Static	RAM	zero	Local
register	Register	Garibage value	Local

Automatic

(i) Automatic variables are allocated memory automatically at runtime.

(ii) The visibility of the automatic variables is limited to the block in which they are defined.

The scope of the automatic variables is limited to the block in which they are defined.

(iii) The automatic variables are initialized to garbage by default.

(iv) The memory assigned to automatic variables gets freed upon exiting from the block.

(v) The keyword used for defining automatic variables is auto.

(vi) Every local variable is automatic in c by default.

Example 1

```
#include <stdio.h>
int main()
{
int a; // auto
char b;
float c;
printf("%d %c %f", a,b,c); // printing initial default value of automatic variables a, b and c.
return 0;
}
```


Output

```
garbage garbage garbage
```

Example 2

```
#include <stdio.h>
int main()
{
int a = 10, i ;
printf ("%d", ++a);
{
int a = 20;
for (i=0; i<3; i++)
{
printf("%d", a); // 20 will be printed 3 times since it is
the local value of a
}
}
printf("%d", a); // 11 will be printed since the scope
of a = 20 is ended.
}
```


Output

11 20 20 20 11

Static

(i) The variables defined as static specifier can hold their value between the multiple function calls.

(ii) Static local variables are visible only to the function or the block in which they are defined.

(iii) A same static variable can be declared many times but can be assigned at only one time.

(iv) The visibity of the static global variable is limited to the file in which it has declared.

(v) Default initial value of the static integral valiable is 0 otherwise null.

(vi) The keyword used to define static variable is static.

Example 1

```
# include <stdio.h>
static char c;
static int i;
static float f;
static char s[100];
void main()
{
printF("%d %d %f %s", c, i, f); // the initial
default value of c, i, and f will be printed.
}
```


Output

```
0  0  0.000000(null)
```

Example 2

```
# include <stdio.h>
void sum()
{
static int a = 10 ;
static int b = 24 ;
printf ("%d %d \n" , a,b);
a++;
b++;
}
void main()
{
int i;
for(i=0; i<3; i++)
{
sum(); // The static variables holds their value between
multiple function calls.
}
}
```


Output

```
10   24
11   25
12   26
```

Register

(i) The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.

(ii) We can not dereference the register variables i.e, we can not use & operator for the register variable.

(iii) The access time of the register variables is faster than the automatic variables.

(iv) The initial default value of the register local variables is 0.

(v) The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler's choice whether or not; the variables can be stored in the register.

(vi) We can store pointers into the register, i.e., a register can store the address of a variable.

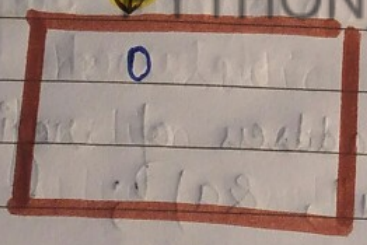
(vii) Static variables can not be stored into the register since we can not use mor

than one storage specifier for the same variable.

Example 1

```
# include <stdio.h>
int main()
{
register int a; // variable a is allocated memory in the CPU register. The initial default value of a is 0.
printf("%d", a);
}
```

Output



Example 2

```
#include <stdio.h>
int main()
{
register int a = 0;
printf("%u", &a); // This will give a compile time
error since we can not access the address
of a register variable.
}
```

Output :


```
main.c:5:5: error: address of register variable 'a'
requested printf("%u", &a);
^~~~~~
```

External

(i) The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.

(ii) The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.

(iii) The default initial value of external integral type is 0 otherwise it is null.

(iv) we can only initialize the extern variable globally. i.e. we can not initialize the external variable within any block or method.

(v) An external variable can be declared many times but can be initialized at only once.

(vi) If a variable is declared as external then the compiler searches for that variable to be initialized some where in the program which may be extern or static. If it is not, then the compiler will show an error.

Example 1

```
#include <stdio.h>
int main()
{
extern int a;
printf("%d", a);
}
```

Output :)

```
main.c : (.text + 0x6): undefined reference to 'a'
collect2: error: ld returned 1 exit status
```

Example 2

```
#include <stdio.h>
int a;
int main()
{
extern int a; // variable a is defined globally, the memory
will not be allocated to a
printf("%d", a);
}
```

Output :

0

Example 3

```
#include <stdio.h>
int a;
int main()
{
extern int a=0; // this will show a compiler error since
we can not use extern and initializer at
same time
printf("%d", a);
}
```

Output


```
compile time error
main.c: In function 'main':
main.c:5:16: error: 'a' has both 'extern' and
initializer extern int a=0;
```

Example 4

```
#include <stdio.h>
int main()
{
extern int a; // compiler will search here for a variable
a defined and initialized somewhere in the
program or not.
printf ("%d", a);
}
int a = 20;
```


Output :

20

Example 5

```
extern int a;
int a = 10;
# include <stdio.h>
int main()
{
printf("%d", a);
}
int a = 20; // compiler will show an error at this line.
```

Output :

compile time error

C Functions Test 1

1) what is the built-in library function for the comparing the two strings?

a. strcmp()

b. equals()

c. str_compare()

d. string_cmp()

2) what is paused when we pas an array as a function argument?

a. Base address of an array

b. Address of the last element of array

c. First value of elements in array

d. All value of element in array.

3) Which function finds the fist occurrence of a substring in another string?

a) strchr()

b) strnset()

c) strstr()

d) None of these

4) What is the built-in library function for adjusting the allocated dynamic memory size.

a. calloc

b. malloc

c. realloc

d. resize

5) which keyword is used to transfer control from a function back to the calling function?

a. return

b. go back

c. switch

d. go to


Array in c
PYTHON_WORLD_IN

Array in C

An array is defined as the collection of similar type of data items stored at contiguous memory locations. arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

C array is beneficial if you have to store similar elements. for example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variables for the marks in each subjects at the contiguous memory locations.

By using the array, we can access the elements easily. only a few lines of code are required to access the elements of the array.

Properties of Array

The array contains the following properties.

(i) Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.

(ii) Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.

(iii) Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantage of Array

1) Code optimization : less code to the access the data.

2) Ease of traversing : By using the for loop, we can retrieve the elements of an array easily.

3) Ease of sorting : To sort the elements of the array, we need a few lines of code only.

4) Random Access : We can access any element randomly using the array.

Disadvantage of Array

Fixed size : whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like Linked List which we will learn later.

Declaration of C Array

We can declare an array in the c language in the following way.

```
data_type array_name[array_size];
```

Now, let us see the example to declare the array.

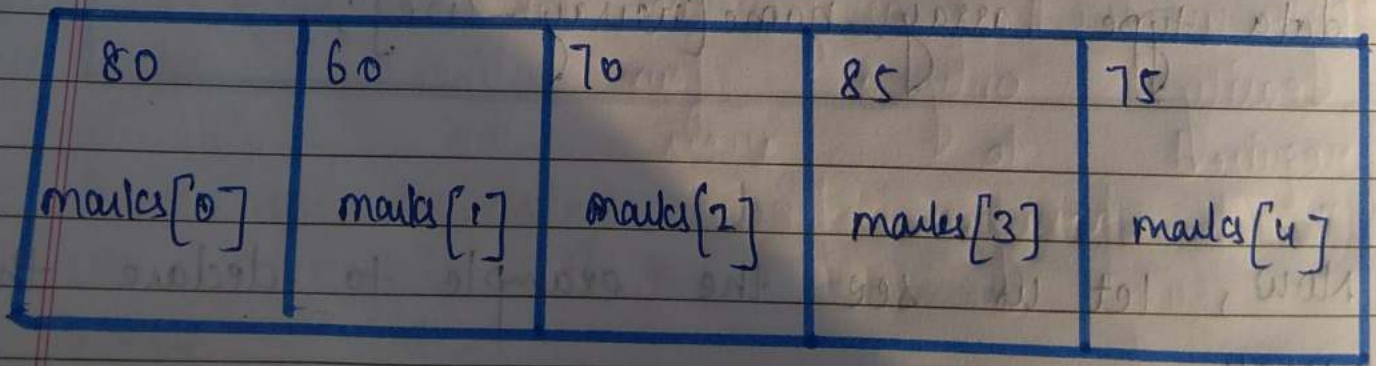
```
int marks[5];
```

Here, int is the data_type, marks are the array_name, and 5 is the array_size.

Initialization of C Array

The simplest way to initialize the array is by using the index of each element. We can initialize each element of the array by using the index. Consider the following example.


```
marks[0] = 80; // initialization of array
marks[1] = 60;
marks[2] = 70;
marks[3] = 85;
marks[4] = 75;
```



C array example

```
#include <stdio.h>
int main (){
int i=0;
int marks[5]; // declaration of array
marks[0] = 80; // initialization of array
marks[1] = 60;
marks[2] = 70;
marks[3] = 85;
marks[4] = 75;
// traversal of array
for (i=0; i<5; i++){
printf("%d\n", marks[i]);
} // end of for loop
return 0;
}
```


Output :-

```
80
60
70
85
75
```

C Array : Declaration with the initialization

We can initialize the c array at the time of declaration. let's see the code.

```
int marks[5] = {20, 30, 40, 50, 60};
```

In such case, there is no requirement to define the size. So it may also be written as the following code.


```
int marks[] = {20, 30, 40, 50, 60};
```

Let's see the C program to declare and initialize the array in C.

```
#include <stdio.h>
int main () {
int i=0;
int marks [5] = {20,30,40,50, 60}; // declaration and
initialization of array.
// traversal of array
for (i=0; i<5; i++) {
printf ("%d\n", marks [i]);
}
return 0;
}
```


Output :

```
20
30
40
50
60
```

C Array Example : Sorting an array ...

In the following program, we are using bubble sort method to sort the array in ascending order.

```
#include <stdio.h>
void main()
{
  int i, j, temp;
  int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
  for(i=0; i<10; i++)
  {
    for(j=i+1; j<10; j++)
    {
      if(a[j] > a[i])
      {
        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
      }
    }
  }
printf("Printing Sorted Element list ...\n");
for(i=0; i<10; i++)
{
```

```
    printf("%d\n", a[i]);
  }
}
```

Program :- To print the largest and second largest element of the array.

```
#include <stdio.h>
void main()
{
int arr[100], i, n, largest, sec_largest;
printf("enter the size of the array?");
scanf("%d", &n);
printf("enter the elements of the array?");
for(i=0; i<n; i++)
{
 scanf("%d", &arr[i]);
}
largest = arr[0];
sec_largest = arr[1];
for(i=0; i<n; i++)
{
  if(arr[i] > largest)
  {
    sec_largest = largest;
    largest = arr[i];
  }
```

```
        else if(arr[i] > sec_largest && arr[i] != largest)
        {
            sec_largest = arr[i];
        }
    }
    printf("largest = %d, second largest = %d", largest,
sec_ largest);
}
```

Two Dimensional Array in C

The two-dimensional array can be defined as an array of arrays. The 2d array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions whenever required.

Declaration of two dimensional Array in C ...

The syntax to declare the 2D array is given below.

```
data_type array_name [rows][columns];
```

Consider the following example.


```
int twodimen[4][3];
```

Here, 4 is the number of rows, and 3 is the number of columns.

Initialization of 2D Array in C

In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously. However, this will not work with 2D arrays. We will have to define at least the second dimension of the array. The two dimensional array can be declared and defined in the following way.

```
int arr[4][3] = {{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
```

Two dimensional array example in C

```
#include <stdio.h>
int main(){
int i=0,j=0;
int arr[4][3] = {{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
// traversing 2D array
for (i=0; i<4; i++){
for (j=0; j<3; j++){
  printf("arr[%d][%d] = %d\n", i,j, arr[i][j]);
} // end of j
} // end of i;
```

```
return 0;
}
```

Output :

```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6
```

C 2D array example : Storing elements in a matrix and printing it

```
#include <stdio.h>
void main()
{
  int arr[3][3], i, j;
  for (i=0; i<3; i++)
  {
    for (j=0; j<3; j++)
    {
      printf("enter a[%d][%d]:", i, j);
      scanf("%d", &arr[i][j]);
    }
  }
  printf("\n printing the elements ....\n");
  for (i=0; i<3; i++)
  {
    printf("\n");
    for (j=0; j<3; j++)
    {
      printf("%d\t", arr[i][j]);
    }
  }
}
```

Output :

```
Enter a[0][0] : 56
Enter a[0][1] : 10
Enter a[0][2] : 30
Enter a[1][0] : 34
Enter a[1][1] : 21
Enter a[1][2] : 34

Enter a[2][0] : 45
Enter a[2][1] : 56
Enter a[2][2] : 78
```


printing the elements ....

56	10	30
34	21	34
45	56	78

Passing Array to function in C

In C, there are various general problems which requires passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and then passing into the function, we can declare and initialize an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values.

As we know, that the array-name contains the address of the first element. Here, we must notice that we need to pass only the name of the array in the function which is intended to accept an array. The array defined as the formal parameter will automatically refer to the array specified by the array name

defined as an actual parameter.

Consider the following syntax to pass an array to the function.

```
functionname(arrayname); // passing array
```

Methods to declare a function that received an array as an argument.

There are 3 ways to declare the function which is intended to receive an array as an argument.

First way :-

```
return_type function(type arrayname[])
```

Declaring blank subscript notation [] is the widely used technique.

Second way :-

```
return_type function(type arrayname[SIZE])
```

Optionally, we can define size in subscript notation [].

Third way :-

```
return_type function(type *arrayname)
```

you can also use the concept of a pointer. In pointer, we will discuss about it.

C language passing an array to function Example ...

```
#include <stdio.h>
int minarray (int arr[], int size){
int min = arr[0];
int i=0;
for (i=1; i<size; i++){
if (min > arr[i]){
min = arr[i];
}
} // end of for
return min;
} // end of function

int main(){
int i=0, min=0;
int numbers[] = {4,5,7,3,8,9}; // declaration of array

min = minarray(numbers, 6); // passing array with size
printf("minimum number is %d\n", min);
return 0;
}
```


Output :- minimum number is 3

C function to sort the array...

```
#include <stdio.h>
void Bubble_Sort(int[]);
void main()
{
 int arr[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
 Bubble_Sort(arr);
}
void Bubble_Sort(int a[]) // array a[] points to arr.
{
int i, j, temp;
 for(i=0; i<10; i++)
 {

  for(j=i+1; j<10; j++)
  {
   if(a[j] < a[i])
   {
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
   }
  }
 }
 printf("Printing Sorted Element list ...\n");
 for(i=0; i<10; i++)
{
  printf("%d\n", a[i]);
}
}
```

Output :

```
Printing Sorted element list ...
7
9
10
12
23
23
34
44
78
101
```

Returning array from the function ...

As we know that, a function can not return more than one value. However, if we try to write the return statement as return a, b, c; to return three values (a, b, c), the function will return the last mentioned value which is c in our case. In some problems, we may need to return multiple values from a function. In such cases, an array is returned from the function.

Returning an array is similar to passing the array into the function. The name of the array is returned from the function. To make a function returning an array, the following syntax is used.

```
int *Function_name(){
// some statements;
return array_type;
}
```

To store the array returned from the function, we can define a pointer which points to that array. we can traverse the array by increasing that pointer since pointer initially points to the base address of the array. consider the example that contains a functions returning the sorted array.

```
#include <stdio.h>
int *Bubble_Sort(int []);
void main()
{
 int arr[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
 int *p = Bubble_Sort(arr), i;
 printf("printing sorting element ... \n");
 for(i=0; i<10; i++)
 {
   printF("%d\n", *(p+i));
 }
}
int *Bubble_Sort(int a[]) // array a[] points to arr.
{
int i, j, temp;
 for(i=0; i<10; i++)
 {
   for(j= i+1; j<10; j++)
   {
    if (a[j] < a[i])
    {
```

```
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    return a;
}
```

Output :

```
Printing Sorted Element List ...
7
9
10
12
23
23
34
44
78
101
```

C Array Test

1) In C, if we pass an array an as argument to a function, what actually get passed?

a. Addres of the last element of array.

b. Base address of the array

c. value of elements in array

d. First element of the array.

2) what will be the output of the below program?

```
#include <stdio.h>
main()
{
  char X[] = "JavaTpoint", Y[] = "JavaTpoint";
  if (x==y) {
  printf ("Strings are Equals");
  }
}
```

a. Strings are Equals

b. No output

c. Runtime error

d. compilation error

3) what will be the output of the below program

```
#include <stdio.h>
main() {
  char x[] = "Hi\0Hello";
  printf("%d %d", strlen(x), sizeof(x));
}
```


a. 5 9

b. 9 20

c. 2 9

d. 2 5

4) A pointer to a block of memory is effectively same as an array.

a. True

b. False

5) which of the following statements are correct about array in C?

1. The expression num[2] represents the very second element in the array.

2. The declaration of num[SIZE] is allowed if SIZE is a macro.

3. The array of int num[20]; can store 20 elements.

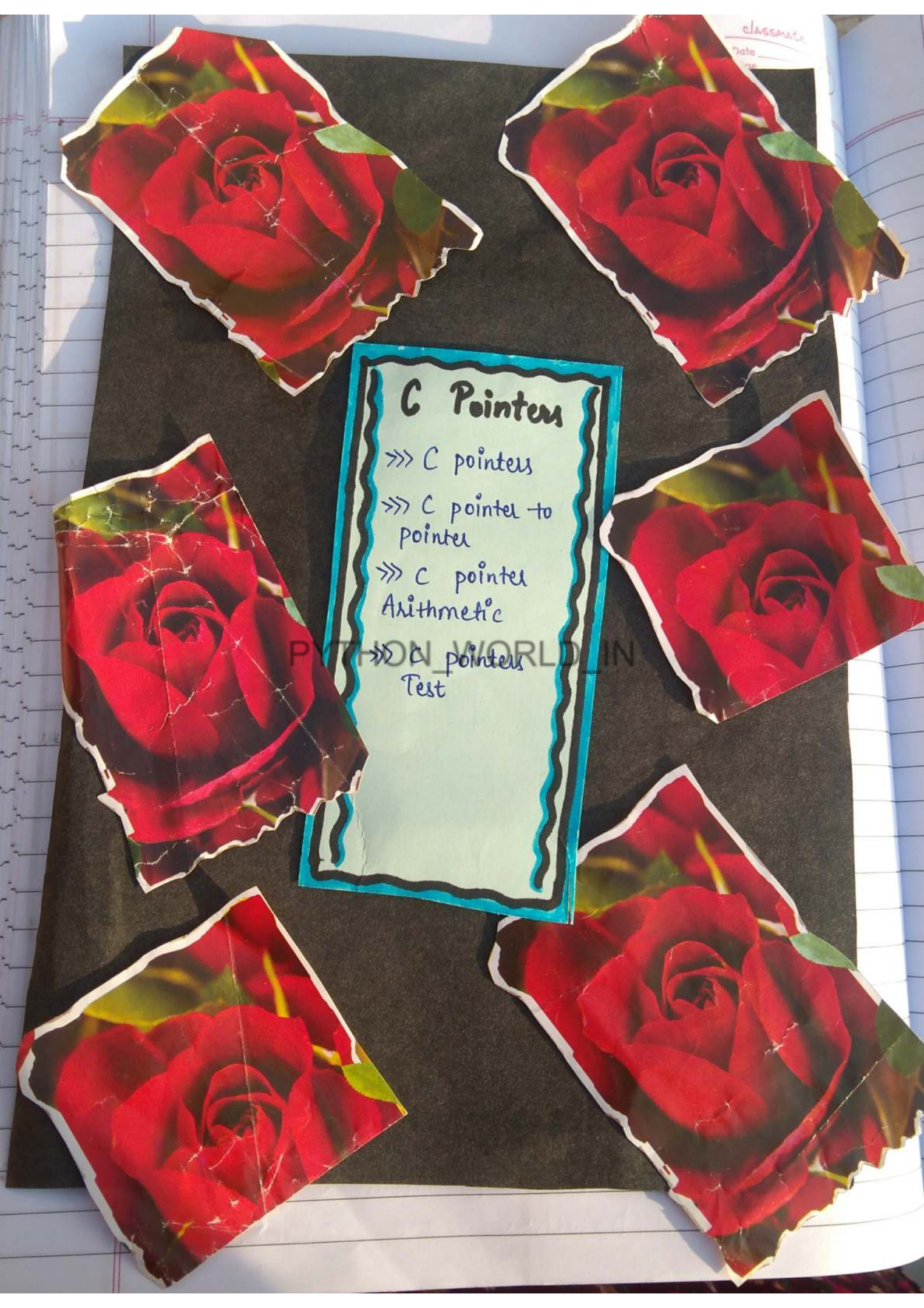
4. It is necessary to initialize array at the time of declaration.

a. 2

b. 2, 3

c. 1, 4

d. 2, 4


C Pointers
>>> C pointers
>>> C pointer to pointer
>>> C pointer Arithmetic
>>> C pointers Test
PYTHON_WORLD_IN

Pointers in C

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

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


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

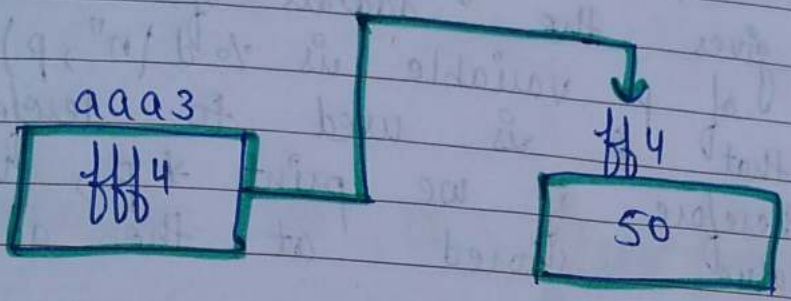
Declaring a pointer

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

```
int *a; // pointer to int
char *c; // pointer to char
```

Pointer Example

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



As you can see in the above figure, pointer variable stores the address of number variable, i.e., ffff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

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

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

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


Output :-

Address of number variable is fff4.
Address of p variable is fff4.
value of p variable is 50

Pointer to array

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

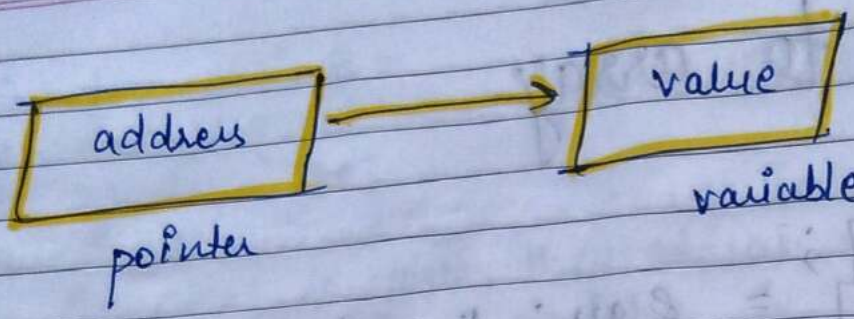
Pointer to a function

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


Pointer to Structure

```
struct st {
    int i;
    float f;
} ref;

struct st *p = &ref;
```



Advantage of pointer

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

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

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

Usage of pointer

There are many applications of pointers in c language.

Dynamic memory allocation

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

2) Arrays, functions and structures
pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address of (&) Operator

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


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

Output :

value of number is 50, address of number is fff4.

Null pointer

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

```
int *p = NULL;
```


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

Pointer Program :- To swap two numbers without using the 3rd variable.

```
#include <stdio.h>
int main() {
int a = 10, b = 20, *p1 = &a, *p2 = &b;
printf("Before swap : *p1 = %d *p2 = %d", *p1, *p2);
*p1 = *p1 + *p2;
*p2 = *p1 - *p2;
*p1 = *p1 - *p2;

printf("\n After swap : *p1 = %d *p2 = %d", *p1, *p2);
return 0;
}
```


Output :

Before swap : *p1 = 10 *p2 = 20
After swap : *p1 = 20 *p2 = 10

Reading complex pointers

There are several things which must be taken into the consideration while reading the complex pointers in c, let's see the precedence and the associativity of the operators which are used regarding pointers.

Operator	Precedence	Associativity
(), []	1	Left to Right
*, identifier	2	Right to left
Data type	3	-

Here, we must notice that,

(i) (): This operator is a bracket operator used to declare and define the function.

(ii) []: This operator is an array subscript operator.

(iii) *: This operator is a pointer operator.

(iv) identifier: It is the name of the pointer. The priority will always be assigned to this.

(v) Data type: Data type is the type of the variable to which the pointer is intended to point. It also includes the modifier like signed int, long, etc.

How to read the pointer: int (*p)[10].

To read the pointer, we must see that () and [] have the equal precedence. Therefore, their associativity must be considered here. The associativity is left to right, so the priority goes to ()

Inside the bracket (), pointer operator * and pointer name (identifier) p have the same precedence. Therefore, their associativity must be considered here and which is right to left, so the priority goes to p, and the second priority goes to *.

Asign the 3rd priority to [] since the data type has the last precedence. Therefore, the pointer will look like following.

- char -> 4
- * -> 2
- p -> 1
- [10] -> 3

The pointer will be read as p is a pointer to an array of integers of size 10.

Example

How to read the following pointer?

int (*p) (int (*) [2], int (*) void))

Explanation

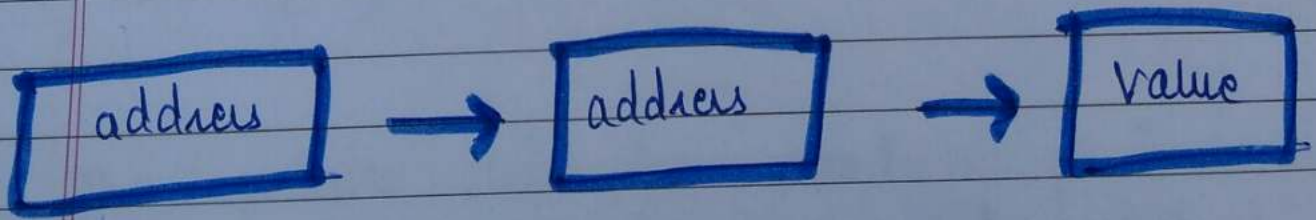
This pointer will be read as p is a pointer to such function which accepts the first parameter as the pointer to a one dimensional array of integers of size two and the second parameter as the pointer to a function which parameter is void and return type is the Integer.

C Double pointer (pointer to Pointer)

As we know that, a pointer is used to store the address of a variable in c. pointer reduces the access time of a variable. However, In c, we can also define a pointer to store the address of another pointer (pointer to pointer). Such pointer is known as a double pointer (pointer to pointer).

The first pointer is used to store the address of variable whereas the second pointer is used to store the address of the first pointer.

Let's understand it by the diagram given below.



pointer pointer variable

The syntax of declaring a double pointer is given below.

```
int **p; // pointer to a pointer which is pointing to an integer.
```

Consider the following example.

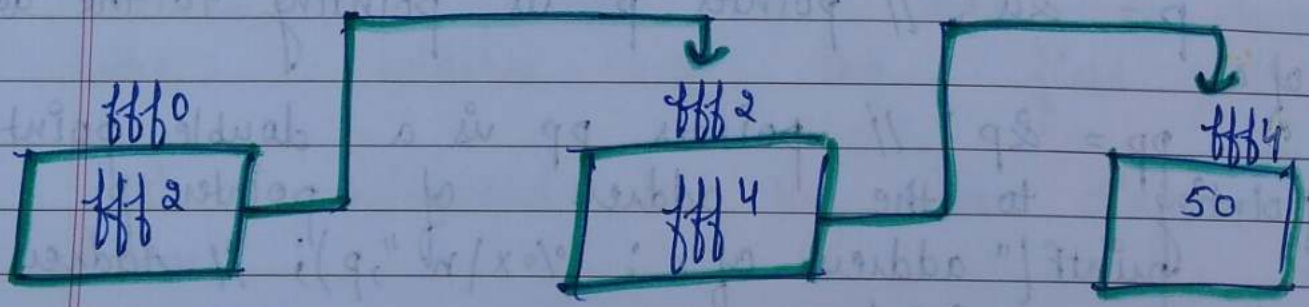
```
#include <stdio.h>
void main()
{
    int a = 10;
    int *p;
    int **pp;
    p = &a; // pointer p is pointing to the address of a
    pp = &p; // pointer pp is a double pointer pointing to the address of pointer p
    printf("address of a: %x\n",p); // Address of a will be printed.
    printf("address of p: %x\n", pp); // Address of p will be printed.
    printf("value stored at p: %d\n", *p); // value stored at the address contained by p i.e. 10 will be printed.
    printf("value stored at pp: %d\n", **pp); // value stored at the address contained by the pointer stoyred at pp.
}
```

Output ↓

```
address of a : d2698734
address of p : d2698738
value stored at p : 10
value stored at pp : 10
```

C double pointer example

Let's see an example where one pointer points to the address of another pointer.



As you can see in the above figure, p2 contains the address of p (ffff2); and p contains the address of number variable (ffff4).

```c
#include <stdio.h>
int main() {
int number = 50;
int *p; // pointer to int
int **p2; // pointer to pointer
p = &number; // stores the address of number variable
p2 = &p;
printf("Address of number variable is %x \n", &number);
printf("Address of p variable is %x \n", p);
printf("value of *p variable is %d \n", *p);
printf("Address of p2 variable is %x \n", p2);
printf("value of **p2 variable is %d \n", *p);
return 0;
}
```


Output :

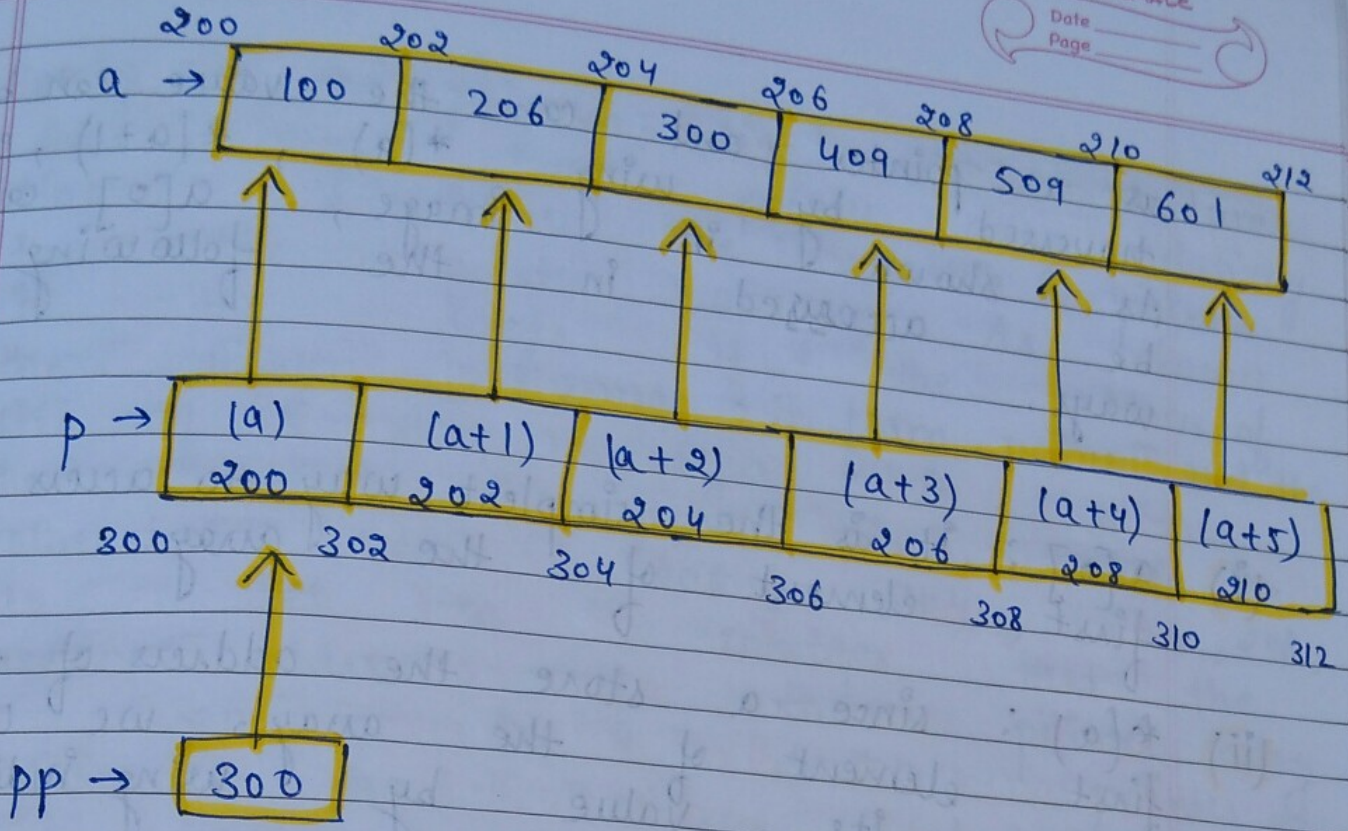
```
Address of number variable is fff4
Address of p variable is fff4.
value of *p variable is 50.
Address of p2 variable is fff2
value of **p variable is 50
```

Question :- what will be the output of the following program?

```
#include <stdio.h>
void main()
{
    int a[10] = {100, 206, 300, 409, 509, 601}; //Line1
    int *p[] = {a, a+1, a+2, a+3, a+4, a+5}; //Line2
    int **pp = p; // line 3
    pp++; // line 4
    printf("%d %d %d\n", pp-p, *pp-a, **pp); //line5
    *pp++; // line 6
    printf("%d %d %d\n", pp-p, *pp-a, **pp); // line 7
    ++*pp; // line8
    printf("%d %d %d\n", pp-p, *pp-a, **pp); // line 9
    ++**pp; // line 10
    printf("%d %d %d\n", pp-p, *pp-a, **pp); // line11
}
```


Explanation

In the above question, the pointer arithmetic is used with the double pointer. An array of 6 elements is defined which is pointed by an array of pointer p.



To access a[0] → a[0] = *(a) = *p[0] = **(p+0) = **(pp+0) = 100

The pointer array p is pointed by a double pointer pp. However, the above image gives you a brief idea about How the memory is being allocated to the array a and the pointer array p. The elements of p are the pointers that are pointing to every element of the array a. Since we know that the array name contains the base address of the array hence, it will work

as a pointer and can the value can be traversed by using *(a), *(a+1), etc. As shown in image, a[0] can be accessed in the following ways.

(i) a[0]: it is the simplest way to access the first element of the array.

(ii) *(a): since a store the address of the first element of the array, we can access its value by using indirection pointer on it.

(iii) *p[0]: if a[0] is to be accessed by using a pointer p to it, then we can use indirection operator (*) on the first element of the pointer array p, i.e., *p[0].

(iv) **(pp): as a pp stores the base address of the pointer array, *pp will give the value of the first element of the pointer array that is the address of the first element of the integer array. **pp will give the actual value of the first element of the integer array.

coming to the program, line 1 and 2 declare the integer and pointer array relatively. Line 3 initializes the double pointer to the pointer array p. As shown in the image, if the address of the array starts from 200 and the size of the integer is 2, then the pointer array will contain the values as 200, 202, 204, 206, 208, 210, let us consider that the base address of the pointer array is 300; the double pointer pp contains the address of pointer array i.e., 300. line number 4 increases the value of pp by 1, i.e., pp will now point to address 302.

line number 5 contains an expression which print three values, i.e., pp - p, *pp - a, **pp.

Let's calculate them each one of them.

(i) pp = 302, p = 300 $\Rightarrow$ pp - p = (302 - 300)/2
$\Rightarrow$ pp - p = 1
i.e., 1 will be printed.

(ii) pp = 302, *pp = 202, a = 200 $\Rightarrow$ *pp - a
= 202 - 200 = 2/2 = 1
i.e., 1 will be printed.

(iii) pp = 302, *pp = 202, *(*pp) = 206,
i.e., 206 will be printed.

Therefore, as the result of line 5, The output 1, 1, 206 will be printed on the console. on line 6, *pp++ is written. Here, we must notice that two unary operators * and ++ will have the same precedence. Therefore, by the rule of associativity, it will be evaluated from right to left. Therefore the expression *pp++ can be rewritten as (*(pp++)). Since, pp = 302 which will now become, 304. * pp will give 204.

On line 7, again the expression is written which prints three values i.e., pp-p, *pp-a, *pp. let's calculate each one of them.

(i) pp = 304, p = 300 ⇒ pp-p = (304-300)/2
⇒ pp-p = 2
i.e., 2 will be printed.

(ii) pp = 304, *pp = 204, a = 200 ⇒ *pp - a
= (204 - 200) / 2
= 2
i.e., 2 will be printed.

(iii) pp = 304, *pp = 204, *(*pp) = 300
i.e., 300 will be printed.

Therefore, as the result of line 7, The output 2, 2, 300 will be printed on the console. on line 8, ++*pp is written.

According to the WWW PYTHON_WORLD_IN associativity this can be rewritten as, (++(*(pp))).
Since, pp = 304, *pp = 204, the value of *pp = *(p[2]) = 206 which will now point to a[3].

On line 9, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. let's calculate each one of them.

(i) pp = 304, p = 300 ⇒ pp - p = (304 - 302)/2
⇒ pp - p = 2

i.e., 2 will be printed.

(ii) pp = 304, *pp = 206, a = 200 => *pp - a
= (206 - 200) / 3
= 3

i.e., 3 will be printed.

(iii) pp = 304, *pp = 206, *(*pp) = 409,
i.e., 409 will be printed.

Therefore, as the result of line 9, the output 2, 3, 409 will be printed on the console. on line 10, ++**pp is written

according to the rule of associativity, this can be rewritten as, (++(*(*pp))). pp = 304, *pp = 206, **pp = 409,
++**pp => *pp = *pp + 1 = 410.

In other words, a[3] = 410.

On line 11, again the expression is written which prints three values. i.e., pp-p, *pp-a, *pp.

Let's calculate each one of them.

(i) pp = 304, p = 300 ⇒ pp - p = (304 - 300)/2
⇒ pp - p = 2,
i.e., 2 will be printed.

(ii) pp = 304, *pp = 206, a = 200 ⇒ *pp - a = (206 - 200)/2
= 3,
i.e., 3 will be printed.

(iii) On line 8, **pp = 410

Therefore, as the result of line 9, the output 2, 3, 410 will be printed on the console.

At last the output of the complete program will be given as:

Output ↓

```
1 1 206
2 2 300
2 3 409
2 3 410
```

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 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:

(i) Increement

(ii) Decrement

(iii) Addition

(iv) Subtraction

(v) Comparison.

Incrementing pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is some what 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:

```
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.


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

Output :

Address of p variable is 3214864300
After increment: Address of p variable is 3214864304.

Traversing array by using pointer

```
#include <stdio.h>
void main()
{
  int ass[5] = {1,2,3,4,5};
  int *p = ass;
  int i;
  printf("printing array elements... \n");
  for(i=0; i<5; i++)
  {
    printf("%d", *(p+i));
  }
}
```


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:

```
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.

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

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:

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.

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


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 the subtracting value from the pointer variable is given below:

```
new_address = current_address - (number * size_of(data type))
```

32 bit

For 32bit 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.

```
#include <stdio.h>
int main(){
int number = 50;
int *p; // pointer to int
p = &number; // stores the address of number variable
printF(" Address of p variable is %04\n", p);
p = p-3; // subtracting 3 from pointer variable
printF("After subtracting 3: Address of p variable
is %u \n", p);
return 0;
}
```


Output :-

Address of p variable is 3214864300
After subtracting 3: Address of p variable is 3214

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,

Address2 - Address1 = (subtraction of two addresses)/size of data type which pointer points to

Consider the following example to subtract one pointer from another.

```
#include <stdio.h>
void main()
{
    int i=100;
    int *p= &i;
    int *temp;
    temp = p;
    p=p+3;
    printf("Pointer Subtraction: %d - %d = %d",
p, temp, p-temp);
}
```

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 topic, 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.


```
#include <stdio.h>
int addition ();
int main()
{
   int result;
   int (*ptr) ();
   ptr = &addition;
   result = (*ptr) ();
   printf("The sum is %d", result);
}
int addition()
{
  int a, b;
  printf("Enter two numbers:");
  scanf("%d %d", &a, &b);
  return a+b;
}
```

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.

```
#include <stdio.h>
int show();
int showadd(int);
int (*arr[3])();
int (*(*ptr)[3])();

int main()
```

```
{
    int result1;
    arr[0] = show;
    arr[1] = showadd;
    ptr = &arr;
    result1 = (**ptr)();
    printf(" printing the value returned by show: %d",
result1);
    (*(*ptr+1))(result1);
}
int show()
{
    int a = 65;
    return a++;
}
int showadd(int b)
{
    printf("\n Adding 90 to the value returned by show: %d",
b+90);
}
```


Output :

Printing the value returned by show: 65
Adding 90 to the value returned by show: 155

C Pointers Test

1) In a structure, if a variable works as a pointer then from the given below operators which operator is used for accessing data of the structure using the variable pointer?

a. %

b. ->

c. .

d. #

2) For the array element a[i][j][k][2], determine the equivalent pointer expression.

a. *(*(*(*(a+i)+j)+k)+2)

b. *(((a+m)+n+o+p)

c. ((((a+m)+n)+o)+p)

d. *((((a+m)+n)+o+p)

3) Are the expression ++*ptr and *ptr++ are same?

a. True

b. False

4) Select the correct statement which is a combination of these two statements,

```
statement 1: p = (char*) malloc(100);
statement 2: char *p;
```

a. char *p = (char*) malloc(100);

b. char *p = (char) malloc(100);

c. char p = *malloc(100);

d. None of the above

5) For the below mention c statement, what is your comment?

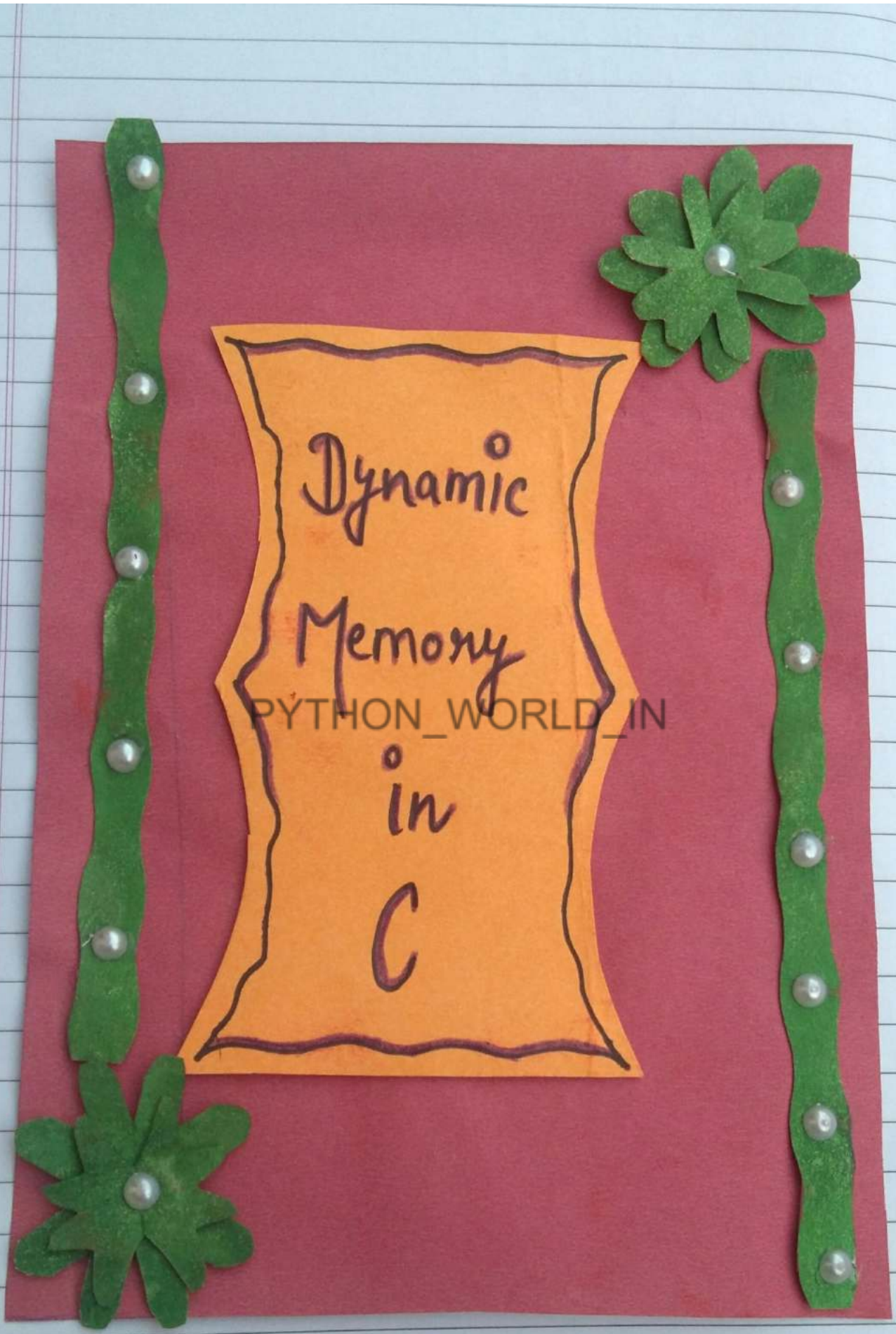
```
signed int *p = (int*) malloc(sizeof(unsigned int));
```

a. would throw Runtime error

b. Improper typecasting

c. Memory will be allocated but cannot hold an int value in the memory.

d. No problem with the statement.


Dynamic
Memory
PYTHON_WORLD_IN
in
C

Dynamic Memory ...

Dynamic memory allocation in C

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

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

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

S.No	Static memory allocation	dynamic memory allocation
(i)	memory is allocated at compile time.	memory is allocated at run time.
(ii)	memory can't be increased while executing program.	memory can be increased while executing program
(iii)	used in array.	used in linked list.

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

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

malloc() function in C

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

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

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

ptr = (cast-type*) malloc (byte-size)

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

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int n, i, *ptr, sum=0;
    printf("Enter number of elements:");
    scanf("%d", &n);
    ptr = (int*) malloc(n* sizeof(int)); // memory allocated using malloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array:");
    for(i=0; i<n; ++i)
    {
        scanf("%d", ptr+i);
        sum += *(ptr+i);
    }
    printf("Sum = %d", sum);
    free(ptr);
return 0;
}
```

Output :-

```
Enter elements of array : 3
Enter elements of array : 10
10
10
Sum = 30
```

Calloc() function in C

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

It initially all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

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

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

```
#include <stdio.h>
#include <stdlib.h>
int main(){
int n, i, *ptr, sum=0;
    printf("Enter number of elements:");
    scanf("%d", &n);
    ptr = (int*)calloc(n, sizeof(int)); //memory
 allocated using calloc
    if (ptr == NULL)
    {
      printf("Sorry! unable to allocate memory");
      exit(0);
    }
    printf("Enter elements of array:");
    for(i=0; i<n; ++i)
    {
        scanf("%d", ptr+i);
        sum += *(ptr+i);
    }
    printf("Sum = %d", sum);
    free(ptr);
return 0;
}
```

Output :

```
Enter elements of array : 3
Enter elements of array : 10
10
10
Sum = 30
```

realloc() function in C

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

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

```
ptr = realloc(ptr, new-size)
```

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit. let's see the syntax of free() function.

```
free(ptr)
```

C Strings

- String in c
- gets() & puts()
- String functions
- strlen() in c
- strcat() in c
- strcpy() in c
- strcmp() in c
- strrev() in c
- strrev() in c
- strlwr() in c
- strupr() in c
- strstr() in c
- c String Test

C Strings ...

The string can be defined as the one dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important since it is the only way to identify where the string ends. when we define a string as char s[10], the character s[10] is implicitly initialized with the null in the memory.

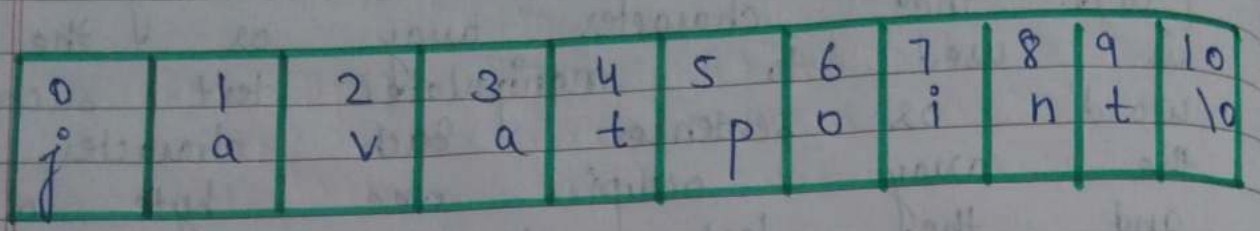
There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring string by char array in c language.

```
char ch[10] = { 'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't',
                '\0' };
```

As we know, array index starts from 0, so it will be represented as in the figure given below.



0	1	2	3	4	5	6	7	8	9	10
j	a	v	a	t	p	o	i	n	t	\0

while declaring string, size is not mandatory so we can write the above code as given below:

```
char ch[] = {'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
```


we can also define the string by the string literal in C language. For example:

```
char ch[] = "javatpoint";
```

In such case, '\0', will be appended at the end of the string by the compiler.

Difference between char array and string literal ...

There are two main differences between char array and literal.

>>> we need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.

>>> The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

String Example in C

Let's see a simple example where a string is declared and being printed. The '%s' is used as a format specifier for the string in c language.

```
#include <stdio.h>
#include <string.h>
int main() {
    char ch[11] = {'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't',
                   'o'};
    char ch2[11] = "javatpoint";
    printf("char array value is: %s\n", ch);
    printf("string literal value is: %s\n", ch2);
return 0;
}
```

Output :

```
char array value is: javatpoint
string literal value is: javatpoint
```


Traversing String

Traversing the string is one of the most important aspects in any of the programming languages. We may need to manipulate a very large text which can be done by traversing the text.

Traversing string is some what different from the traversing an integer array.

we need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end of the string and terminate the loop.

Hence, there are two ways to traverse a string.

>>> By using the length of string

>>> By using the null character

Let's discuss each one of them.

Using the length of string

Let's see an example of counting the number of vowels in a string.

```
#include <stdio.h>
void main()
{
  char s[11] = "javatpoint";
  int i = 0;
  int count = 0;
  while (i<11)
  {
    if(s[i] == 'a' || s[i] = 'e' || s[i] == 'i' || s[i] == 'o' || s[i] == 'u')
```

```
        {
            count++;
        }
        i++;
    }
    printF("The number of vowels %d", count);
}
```

Output :

The number of vowels 4

Using the null character

Let's see the same example of counting the number of vowels by using the null character.

```
#include <stdio.h>
void main()
{
    char s[11] = "Javatpoint";
    int i = 0;
    int count = 0;
    while (s[i] != NULL)
    {
```

```
        if (s[i] == 'a' || s[i] == 'e' || s[i] == 'i' || s[i] ==
'o' || s[i] == 'u')
        {
            count ++;
        }
        i++;
    }printf("The number of vowels %d", count);
}
```

Output 1

The number of vowels 4

Accepting string as the Input

Till now, we have used scanf to accept the input from the user. However, it can also be used in the case of strings but with a different scenario. Consider the below code which stores the string while space is encountered.

```
#include <stdio.h>
void main()
{
    char s[20];
    printf("enter the string?");
    scanf("%s", s);
    printf("you entered %s", s);
}
```

Output 1

```
Enter the string? javatpoint is the best
you entered javatpoint
```


It is clear from the output that, the above code will not work for space separated strings. To make this code working for the space separated strings, the minor changed required in the scanf function, i.e., instead of writing scanf ("%s", s) we must write: scanf ("%[^\n]s", s) which instructs the compiler to store the string s while the new line (\n) is encountered.

Let's consider the following example to store the space separated strings.

```
# include <stdio.h>
void main()
 {
    char s[20];
    scanf("%[^\n]s", s);
    printf("you entered %s", s);
 }
```

Output


```
Enter the string? javatpoint is the best
you entered javatpoint is the best
```

Here we must also notice that we do not need to use address of (&) operator in scanf to store a string since string s is an array of characters and the name of the array. i.e., s indicates the base address of the string (character array) therefore we need not use & with it.

Some important points

However, there are the following points which must be noticed while entering the strings by using scanf.

(i) The compiler does not perform bounds checking on the character array. Hence, there can be a case where the length of the string can exceed the dimension of the character array which may always overwrite some important data.

(ii) Instead of using scanf, we may use gets() which is an inbuilt function defined in a header file string.h. The gets() is capable of receiving only one string at a time.

Pointers with strings:

We have used pointers with the array, function and primitive data types so far. However, pointers can be used to point to strings. There are various advantages of using pointers to point strings.

let's us consider the following example to access the string via the pointer.

```
#include <stdio.h>
void main()
{
   char s[11] = "Javatpoint";
   char *p = s; // pointer p is pointing to string s.
   printf("%s" p); // the string javatpoint is printed if we print p.
}
```

Output :

```
Javatpoint
```


char s[11] = "javatpoint"

index	0	1	2	3	4	5	6	7	8	9	10
values	j	a	v	a	t	p	o	i	n	t	\0
Address	20	21	22	23	24	25	26	27	28	29	30
variable	ptr										
value	20										
Address	10										

char *ptr = s

As we know that string is an array of characters, the pointers can be used in the same way they were used with arrays. In the above example, p is declared as a pointer to the array of characters s. p affects similar to s since s is the base address of the string and treated as a pointer internally.

However, we can not change the content of s or copy the content of s into another string directly. For this purpose, we need to use the pointers to store the strings. In the following example, we have shown the use of pointers to copy the content of a string into another.


```
#include <stdio.h>
void main()
{
  char *p = "hello javatpoint";
  printf("String p: %s\n", p);
  char *q;
  printf("copying the content of p int q... \n");
  q = p;
  printf("String q: %s\n", q);
}
```

Output

String p : hello javatpoint
copying the content of p into q ...
String q : hello javatpoint

Once a string is defined, it cannot be reassigned to another set of characters. However, using pointers, we can assign the set of characters to the string. consider the following example.


```
#include <stdio.h>
void main()
{
  char *p = "hello javatpoint";
  printf("Before assigning : %s\n");
  p = "hello";
  printf(" After assigning : %s\n", p);
}
```

Output

Before assigning : hello javatpoint
After assigning : hello

C gets() & puts()

The gets() and puts() are declared in the header file stdio.h. Both the functions are involved in the input/ output operations of the stings.

C gets() function

The gets() function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string. The gets() allows the user to enter the space separated string. It returns the string entered by the user.

Declaration

```
char [] gets (char []);
```

Reading string using gets()

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


Output

```
Enter the string?
javatpoint is the best
you entered javatpoint is the best.
```

The gets() function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered. It suffers from buffer overflow, which can be avoided by using fgets(). The fgets() make sure that not more than the maximum limit of characters are read.

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

char str[20];
printf("Enter the string?");
gets(str, 20, stdin);
printf("%s", str);
}
```

Output

Enter the string? javatpoint is the best website
javatpoint is the b

C puts() function

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

Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by puts() will always be equal to the number of characters present in the string plus 1.

Declaration

```
int puts (char [])
```

Let's see an example to read a string using gets() and print it on the console using puts().


```
#include <stdio.h>
#include <string.h>
int main()
{
char name [50];
printf(" enter you name:");
gets(name);          // reads string from user
printf(" your name is:");
puts (name);         // displays string
return 0;
}
```

Output 1

Enter your name: Satinder Kaur
your name is : Satinder Kaur

C String Functions

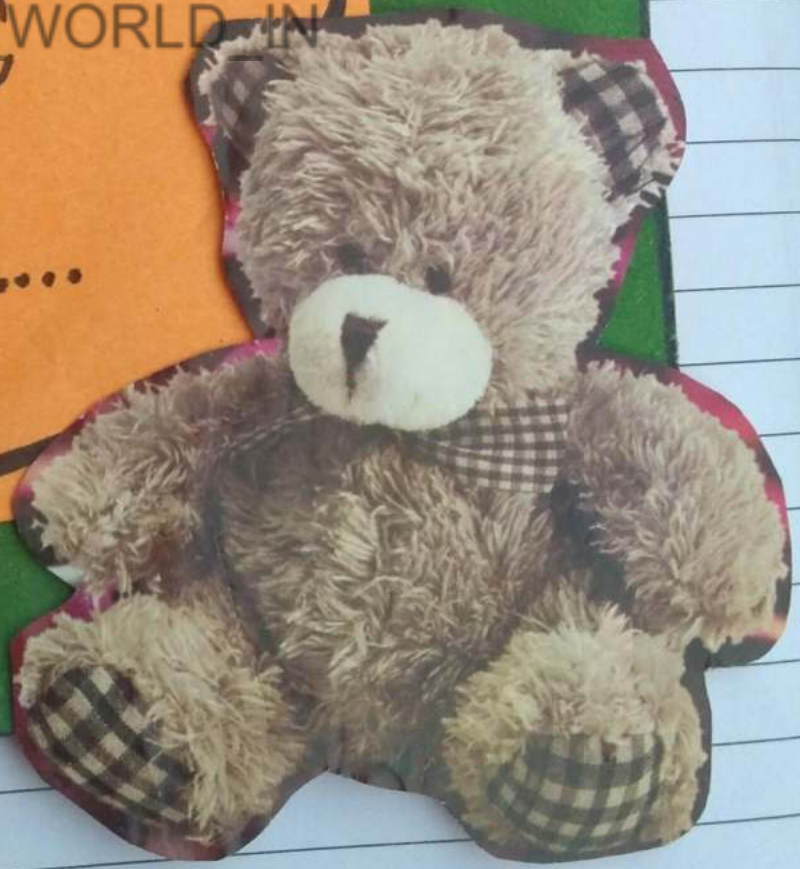
Function	Desciption
☆ strlen(string name)	returns the length of string name.
☆ strcpy(destination, source)	copies the contents of source string to destination string.
☆ strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.

★ strcmp (first_string, second_string)	Compares the first string with second string. If both stings are same, it returns 0.
★ strrev (string)	returns reverse string.
★ strlwr (string)	returns string characters in lowercase.
★ strupr (string)	returns string characters in uppercased.
★ strstr (string)	The strstr() function returns pointer to the first occurrence of the matched string in the given string. It is used to return substring from first match till the last character.

C Math

>>> Math Functions in C

.....



C Math

C programming allows us to perform mathematical operations through the functions defined in <math.h> header file. The <math.h> header file contains various methods for performing mathematical operations such as sqrt(), pow(), ceil(), floor() etc.

C Math functions ↓

There are various methods in math.h header file. The commonly used functions of math.h header file are given below.

No.	Function	Desciption
1)	ceil(number)	rounds up the given number. It returns the integer value which is greater than or equal to given number.

2)	floor(number)	rounds down the given number. It returns the integer value which is less than or equal to given number.
3)	sqrt(number)	returns the square root of given number
4)	pow(base, exponent)	returns the power of given number.
5)	abs(number)	returns the absolute value of given number.

C Math Example

Let's see a simple example of math functions found in math.h header file.

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

printf("\n%f", ceil(3.6));
printf("\n%f", ceil(3.3));
printf("\n%f", floor(3.6));
printf("\n%f", floor(3.2));
printf("\n%f", sqrt(16));
printf("\n%f", sqrt(7));
printf("\n%f", pow(2,4));
printf("\n%f", pow(3,3));
printf("\n%d", ceil(3.3));
printf("\n%d", abs(-12));
return 0;
}
```


Output :

```
4.000000
4.000000
3.000000
3.000000
4.000000
2.645751
16.000000
27.000000
4.000000
12
```

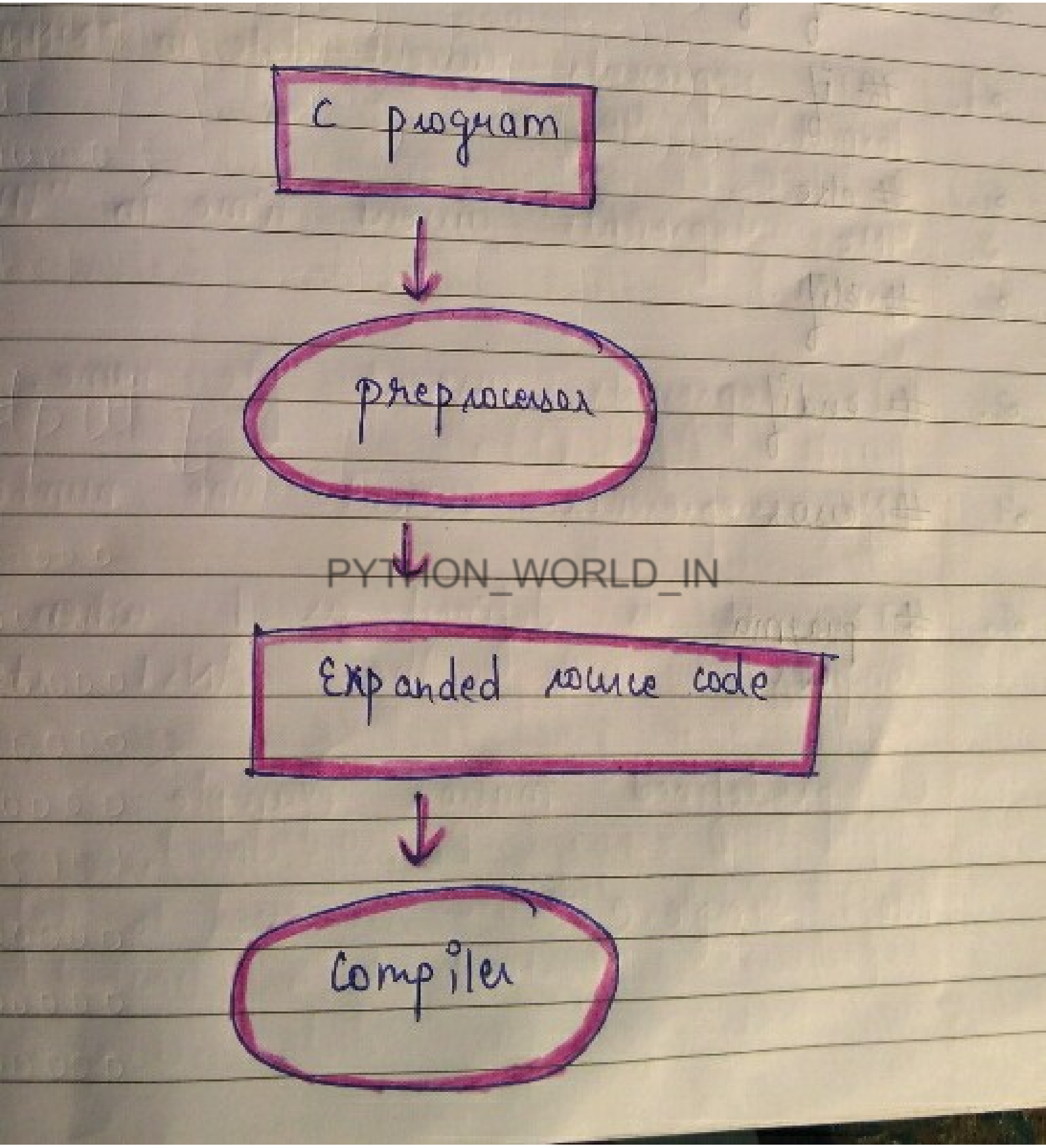
C Preprocessor

- C preprocessor
- Macros in c
- #include in c
- #under in c
- #ifdef in c
- #ifndef in c
- #if in c
- #else in c
- #error in c
- #pragma in c

C Preprocessor Directives...

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.

Note :- Preprocessor directives are executed before the compilation.


C program
preprocessor
PYTHON_WORLD_IN
Expanded source code
Compiler

All preprocessor directives starts with hash # symbol.

Let's see a list of preprocessor directives

- # include
- #define
- # undef
- # ifdef
- # ifndef
- # if
- # else
- # elif
- # endif
- # error
- # pragma

Macros in C

A macro is a segment of code which is replaced by the value of macro. macro define by #define directive.

There are two types of macros:

1) object - like - Macros.
2) Function - like - Macros

The object - like macro → The object - like macro is an identifier that is replaced by value. It is widely used to represent numeric contants.

for example: #define PI 3.14

Here, PI is the macro name which will be replaced by the value 3.14.

Function - like - macro → The function like macro looks like function call. for example:

#define MIN (a,b) ((a)>(b)? (a): (b))

Here MIN is the macro name.

vist #define to see the full example of object like and function like macros.

C Predefined Macro

ANSI C defines many predefined macros that can be used in C program.

No. Macro Desciption

1. DATE represents current date in "MMM DD YYYY" format.
2. TIME represents current time in "HH:MM:SS" format.
3. FILE represents current file name.
4. LINE represents current line number
5. STDC it is defined as 1 when compiler complies with the ANSI standard.

C predefined macros example

File : Simple.c

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

printf("File : %s\n", __FILE__);
printf("DATE : %s\n", __DATE__);
printf("Time : %s\n", __TIME__);
printf("STDC : %d\n", __STDC__);
printf("LINE : %d\n", __LINE__);
return 0;
}
```

Output :-

```
File : Simple.c
Date : Nov 1 2021
Time : 12:28:46
STDC : 1
LINE : 6
```

#Undef in C

The #undef preprocessor directive is used to undefine the constant or macro defined by #define.

Syntax : #undef token

Let's see a simple example to define and the undefine a constant.

```
#include <stdio.h>
# define PI 3.14
# undef PI
main()
{
printf ("%f", PI);
}
```


Output :-

Compile Time Error : 'PI' undeclared.
The # undef directive is used to define the preprocessor constant to a limited scope so that you can declare constant again

Let's see an example where we are defining and undefining number variable. But before undefined, It was used by square variable.

```
#include <stdio.h>
#define number 15
int square = number * number;
#undef number
main()
{
printf("%d", square);
}
```


Output :-

```
225
```

#ifdef in C

The #ifdef preprocessor directive checks if macro is defined by #define.

If yes, it executes the code otherwise #else code is executed, if present.

Syntax :-

```
#ifdef MACRO
// code
#endif
```


Syntax with #else :-

```
#ifdef MACRO
// successful code
#else
// else code
#endif
C #ifdef example
```

Let's see a simple example to use #ifdef preprocessor directive.

```
#include <stdio.h>
#include <conio.h>
#define NOINPUT
void main()
{

int a = 0;
#ifdef NOINPUT
a=2;
#else
printf("Enter a :");
scanf("%d", &a);
#endif
printf("value of a : %d\n",a);
getch();
}
```


Output ↓

```
value of a : 2
```

#include :-

The #include preprocessor directive is used to paste code of given file into current file. It is used to include system defined and user-defined header files. If included file is not found, compiler renders error.

By the use of #include directive, we provide information to the preprocessor where to look for the header files.

There are two variants to use #include directive:

1) #include <filename>

2) #include "filename"

#define :-

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax :- #define token value

e.g. :- #define PI 3.14

#if :-

The #if preprocessor directive evaluates the expression or condition. If condition

is true, it executes the code otherwise #elseif or #else or #endif code is executed.

#else :- The #else preprocessor directive evaluates the expression or condition. if condition of #if is false. It can be used with #if, #elif, #ifdef and ifndef directives.

#error :- The #error preprocessor directives indicates error. The compiler gives fatal error if #error directive is found and skips further compilation proces.

#pragma :- The #pragma preprocessor directive is used to provide additional information to the compiler. The #pragma directive is used by the compiler to offer machine or operating system feature.

Syntax :- #pragma token


Command line arguments
PYTHON_WORLD_IN

Command line Arguments

Command line arguments in C, The arguments passed from command line are called command line arguments. These arguments are handled by main() function.

To support command line argument, you need to change the structure of main() function as given below :

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

Here, argc counts the number of arguments. It counts the file name as the first argument.

The argv[] contains the total number of arguments. The first argument is file name always.

Example

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

Program :1

```
#include <stdio.h>
void main (int argc, char * argv[])
{
printf("Program name is : %s\n", argv[0]);
If (argc < 2)
{ printf("No argument passed through command
        line.\n");
}
else
{
printf("First argument is : %s\n", argv[1]);
}
}
```


Run this Program as follows in Linux :

./program hello

Run this program as follows in windows from command line :

program.exe hello

Output :

Program name is : Program
Firstargument is : hello

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

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

Output:

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

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

Written by :
Satinder kaur