C-Basics:

Started 15 june 2023: Author:-Kommi Druthendra

→**C is the basic level language**←

Comments and escape sequence:

→ Comments are meant for humans. Compiler ignores comments.
→ Comments are used to provide hints for users and make it easy to understand for other coders.

//This is a single line comment.
/* This is a multi
line comment*/

\n → used to creat new line.
\t → used to creat tab.
\\ → displays \
\' → displays '
\" → displays "

EXAMPLE:

```
#include <stdio.h>

int main(){
  //this is a single line comment
  /* this is a
  multi line comments
  */
  printf("Hello\n");
  printf("\tWorld\"");


  return 0;
}
```

OUTPUT:

Hello
World"

Variables:

variables allocate space in memory to store a value. We access variable by referring it name.

```
//initializing variable
int x;
//declaring variable
x = 5;
//printing variable
printf("%d",x);
```

FIRST CODE IN C LANGUAGE:
->HOW TO PRINT HELLO WORLD IN C.
EXAMPLE:

```
#include <stdio.h>
Int main(){
printf(“Hello World”);
return 0;
}
```

OUTPUT:
Hello World
[Program finished]

EXPLANATION:
#Include ------- same as **import** in python.
#include <stdio.h> ----- includes the **standard input output** library functions.

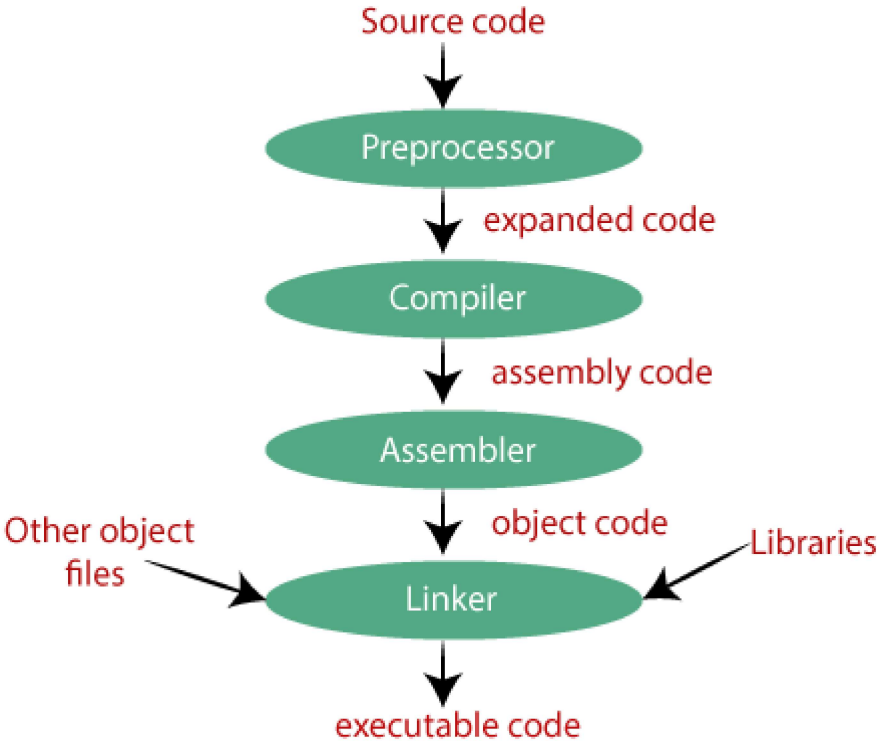
int main() -------- The **main() function is the entry point of every program** in c language.

printf() **-------** function used to print data in the console.

return 0 ------- The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

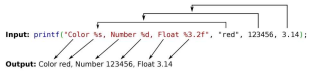
HOW COMPILATION WORKS IN C:
THERE ARE 4 STEPS NAMED AS

- **Preprocessor**
- **Compiler**
- **Assembler**
- **Linker**



printf() and scanf():

printf(); :



printf() is used to print output in console

SYNTAX OF printf():

printf(string_formate, argument_list);

scanf(); :

scanf() is used to read input data from the console.

SYNTAX OF scanf():

scanf("string_formate",&argument_list);

NOTE: To get string input '&' operator is not used.

Let's understand with a simple program.

Let's find the square of the number.

EXAMPLE 1:

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

OUTPUT:

Enter the number: 25

Square of the number is: 625

[Program finished]

EXPLANATION:

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

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

EXAMPLE 2:

Let's go ahead on printf() and scanf():

Let's print addition of two numbers:

```
#include <stdio.h>
```

```
int main(){
int x=0, y=0, result=0;
printf("Enter the first number: ");
scanf("%d",&x);
printf("Enter the second number: ");
scanf("%d",&y);
result=x+y;
printf("Sum of the two numbers is :%d", result);
return 0;
}
```

OUTPUT:
Enter the first number: 1
Enter the second number: 2
Sum of the two numbers is : 3
[Program finished]

Format Specifiers in C:

Format specifiers define the type of data to be printed on standard output. You need to use format specifiers whether you're printing formatted output with printf() or accepting input with scanf().

Some of the % specifiers that you can use in ANSI C are as follows:

SPECIFIED	USED FOR
%c	a single character
%s	a string
%hi	short (signed)
%hu	short (unsigned)
%Lf	long double
%n	prints nothing
%d	a decimal integer (assumes base 10)
%i	a decimal integer (detects the base automatically)

%o	**an octal (base 8) integer**
%x	**a hexadecimal (base 16) integer**
%p	**an address (or pointer)**
%f	**a floating point number for floats**
%u	**int unsigned decimal**
%e	**a floating point number in scientific notation**
%E	**a floating point number in scientific notation**
%%	**the % symbol**

Examples:
%c single character format specifier:

```
#include <stdio.h>

int main() {
  char first_ch = 'f';
  printf("%c\n", first_ch);
  return 0;
}
```

Output:

f

%s string format specifier:

```
#include <stdio.h>

int main() {
  char str[] = "freeCodeCamp";
  printf("%s\n", str);
  return 0;
}
```

Output:

freeCodeCamp

Character input with the %c format specifier:

```
#include <stdio.h>

int main() {
  char user_ch;
  scanf("%c", &user_ch); // user inputs Y
  printf("%c\n", user_ch);
  return 0;
}
```

Output:

Y

String input with the %s format specifier:

```
#include <stdio.h>

int main() {
  char user_str[20];
  scanf("%s", user_str); // user inputs fCC
  printf("%s\n", user_str);
  return 0;
}
```

Output:

fCC

%d and %i decimal integer format specifiers:

```
#include <stdio.h>

int main() {
  int found = 2015, curr = 2020;
  printf("%d\n", found);
  printf("%i\n", curr);
  return 0;
}
```

Output:

2015
2020

%f and %e floating point number format specifiers:

```
#include <stdio.h>

int main() {
  float num = 19.99;
  printf("%f\n", num);
```

```
  printf("%e\n", num);
  return 0;
}
```
Output:

19.990000
1.999000e+01

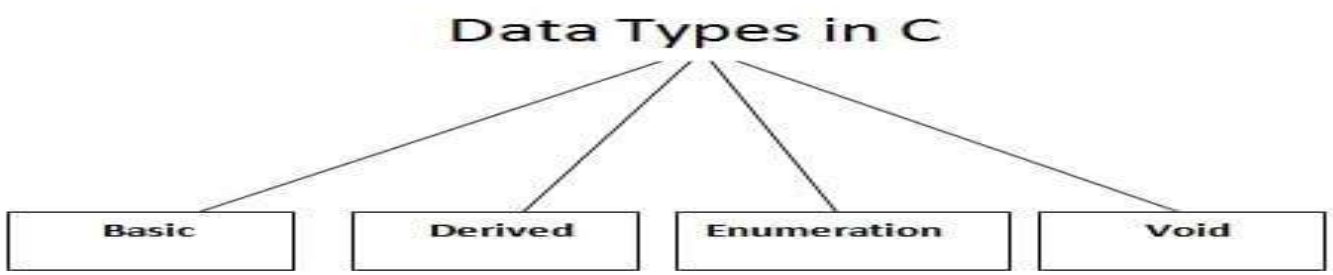
%o octal integer format specifier:
```
#include <stdio.h>

int main() {
  int num = 31;
  printf("%o\n", num);
  return 0;
}
```
Output:

37

%x hexadecimal integer format specifier:
```
#include <stdio.h>

int main() {
  int c = 28;
  printf("%x\n", c);
  return 0;
}
```
Output:

1c

Data-Types in c:



There are following Data types in C language:

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

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, continue,default,do ,double,else,enum,extern,float,for,goto,if,intlong,register,return,short,signed,sizeof,static,struct,switch,typedef,union,unsigned,void,volatile,while.
We will learn all these keywords in future.

C identifiers:

"Identifiers" or "symbols" are the names you supply for variables, types, functions, and labels in your program. Identifier names must differ in spelling and case from any keywords. You can't use keywords (either C or Microsoft) as identifiers; they're reserved for special use. You create an identifier by specifying it in the declaration of a variable, type, or function. In this example, result is an identifier for an integer variable, and main and printf are identifier names for functions.

Types of identifiers:
Internal identifier
External identifier

Internal Identifier:

If the identifier is not used in the external linkage, then it is known as an internal identifier. The internal identifiers can be local variables.

External Identifier:

If the identifier is used in the external linkage, then it is known as an external identifier. The external identifiers can be function names, global variables

Example:

```
#include <stdio.h>

int main()
{
   int result;

   if ( result != 0 )
      printf( "Bad file handle\n" );
}
```

Syntax:

identifier:
   nondigit
   identifier nondigit
   identifier digit

nondigit: one of
_ a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of
  0 1 2 3 4 5 6 7 8 9

The first character of an identifier name must be a nondigit (that is, the first character must be an underscore or an uppercase or lowercase letter). ANSI allows six significant characters in an external identifier's name and 31 for names of internal (within a function) identifiers. External identifiers (ones declared at global scope or declared with storage class extern) may be subject to more naming restrictions because these identifiers have to be processed by other software such as linkers.

Differences between Keyword and Identifier:

KEYWORDS	IDENTIFIERS
Keyword is a pre-defined word.	The identifier is a user-defined word
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letters.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical character.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

Example:

```
int main()
{
   int a=10;
   int A=20;
   printf("Value of a is : %d",a);
   printf("\nValue of A is :%d",A);
   return 0;
}
```

Output:

Value of a is : 10
Value of A is :20

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

Example:

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

Output:

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

Constants in C:

Two ways to define constants in C:
1.const keyword
2.#define preprocessor

1) C const keyword:

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.

Example:

```
#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

NOTE: If you try change the value of const it will return's compile error.

2) C #define preprocessor:

Syntax:

```
#define token value
```

Example:

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

Output:

3.140000

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

Example:

```
#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 #ifdef:

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
```

Macros in C?

Macro in c is defined by the #define directive. Macro is a name given to a piece of code, so whenever the compiler encounters a macro in a program, it will replace it with the macro value. In the macro definition, the macros are not terminated by the semicolon(;).

Syntax:

```
#define macro_name macro_value
```

Example:

```
#include <stdio.h>

#define a 10 //macro definition

int main()

{

   printf("The value of a is: %d", a);

   return 0;

}
```

Output:

The value of a is : 10

Explaination:

From the above code, “a” is a macro name and 10 is the value. Whenever the compiler encounters a macro name, it will replace it with the macro value.

Types of macros in C:
1)Object Like Macros
2)Function Like Macros

Object Like Macros:

A macro is replaced by the value in object-like macros. Generally, a defined value can be a constant numerical value.

Let’s look at the examples of how object-like macros are used in c.

Example:

```
#include <stdio.h>

#define pi 3.14 //macro definition
```

```
int main()

{

    int r = 4;

    float circum;

    circum = 2*pi*r;

    printf("circumference of a circle is: %f", circum);

    return 0;

}
```

Output:
circumference of a circle is: 25.10021

Explaination:

From the above-given code, 3.14 is a value of a macro name pi. In this program, we will calculate a circle's circumference. In the formula 2*pi*r, the pi is replaced by the value 3.14, and the r value is declared in the program.

Consider another example, where we have a macro name pi multiple times in the program.

Example:

```
#include <stdio.h>
#define pi 3.14

int main(){
    int r, area, circum;
    printf("Enter radius of the circle:\n");
    scanf("%d", & r);

    area = pi*r*r;
    printf("Area of the circle is : %d\n", area);
    circum = 2*pi*r;
    printf("Circumference of the circle is : %d", circum);

    return 0;
}
```

Output:
Enter the radius of the circle :
2
Area of the circle is : 12
Circumference of the circle is : 12

Function Like Macros:

The way function call happen in C programs. Similarly, the function is defined in functions like macros, and arguments are passed by the #define directive.

We will use the examples to help you understand how to use functions like macros in c.

Example :

```
#include <stdio.h>

#define add(a, b) (a + b)  //macro definition

int main()

{

    int a = 10, b = 15, result;

    result = add(a, b);

    printf("Addition of two numbers is: %d",result);

    return 0;

}
```

Output:

Addition of two numbers is: 25

Explaination:

Once the compiler finds the add (a,b) function, it will replace it with (a+b) and perform the operation.

Predefined Macros in C:

C programming language provides several predefined data types and functions. Similarly, c also provides predefined macros that can be used in c programs.

Predefined Macros	Description
__FILE__	It contains the current file name .
__DATE__	It displays today's date.
__TIME__	It displays the current time.
__LINE__	Contains line number.
__STDC__	When it is compiled, it will return a nonzero integer value

Example:

```
#include <stdio.h>

int main()

{

  printf("File name is:%s\n", __FILE__ );

  printf("Current Date is:%s\n", __DATE__ );

  printf("Current Time is:%s\n", __TIME__ );

  printf("The Line no is:%d\n", __LINE__ );

  printf("Standard C :%d\n", __STDC__ );

  return 0;

 }
```

Output:
Current Date is: Jun 23 2023
Current Time is:12:02:11
The Line no is:13
Standard C :1

If Statement:

Syntax:

```
if (condition){
  // Code to be executed || if condition is true
    }
```

Example:

```
#include <stdio.h>

int main()

{
   int x;
   printf("Enter the number: ");
   scanf("%d",& x);

   if (x % 2 == 0){
      printf("%d is an even number", x);

   }

  return 0;

 }
```

Output:

Enter the number: 4
4 is an even number

If-else Statement:

Syntax:

```
if (condition){
  // Code to be executed || if condition is true
    } else {
  // Code to be executed || if condition is false
    }
```

Example:

```
#include <stdio.h>

int main()

{
   int x;
   printf("Enter the number: ");
   scanf("%d",& x);

   if (x % 2 == 0){
      printf("%d is an even number", x);

   } else {
      printf("%d is an odd number", x);
   }

  return 0;

 }
```

Output:

Enter the number: 3
3 is an odd number

if else-if statement:

Syntax:

```
if(condition1){
    //code to be executed if condition1 is true
}else if(condition2){
    //code to be executed if condition2 is true
}
else if(condition3){
    //code to be executed if condition3 is true
}
...
else{
    //code to be executed if all the conditions are false
}
```

Example:

```
#include<stdio.h>
int main(){
int number=0;
printf("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

switch Statement:

Syntax:

```
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;
}
```

Example:

```
#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

Nested switch Statement:

Example:

```
#include <stdio.h>
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 swi tch: 10

The value of j evaluated in nested switch: 20

Exact value of i is : 10

Exact value of j is: 20

C - Loops:

Types of C Loops:

1) **do while**
2) **while**
3) **for**

do while :

Syntax:

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

Example:

```
#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\n");
      break;
      case 2:
      printf("Javatpoint\n");
      break;
      case 3:
      exit(0);
      break;
      default:
      printf("please enter valid choice\n");
```

```
    }
    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

1

Hello

do you want to enter more?n

Explaination:

Here, first the code which is in do loop will be executed once then it will check the while condition.If while condition is true then do loop will be executed again this process stops when while condition becomes false.

A simple another **Example:**

```
#include <stdio.h>

int main(){
```

```
    int i = 0;
    do {
        printf("The value of x is : %d\n", i);
        i ++;

    }while(i <= 5);
}
```

Output:

The value of x is : 1
The value of x is : 2
The value of x is : 3
The value of x is : 4
The value of x is : 5

Infinitive do while loop:

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

Syntax:

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

While Loop:

Syntax:

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

Example:

```
#include <stdio.h>
```

```
int main(){
    int i = 0;
    while(i <= 3){
        printf("The value of i is : %d\n", i);
        i++;
    }
}
```

Output:

The value of i is : 1
The value of i is : 2
The value of i is : 3

Example(to print odd numbers using while):

```
#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

Infinitive while loop:

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

Syntax:

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

for Loop:

Syntax:

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

Example:

```
#include<stdio.h>
int main(){
int i=0;
for(i=1;i<=5;i++){
printf("%d \n",i);
}
return 0;
}
```

Output:

1
2
3
4
5

Example-2:

```
#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

Example-3:

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

Output:
0 0 0
1 2 3
2 4 6

Infinitive for loop:

Syntax:

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

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

Nested Loops in C:

Syntax:

```
Outer_loop
```

```
{
   Inner_loop
  {
       // inner loop statements.
  }
     // outer loop statements.
}
```

break statement:

Syntax:

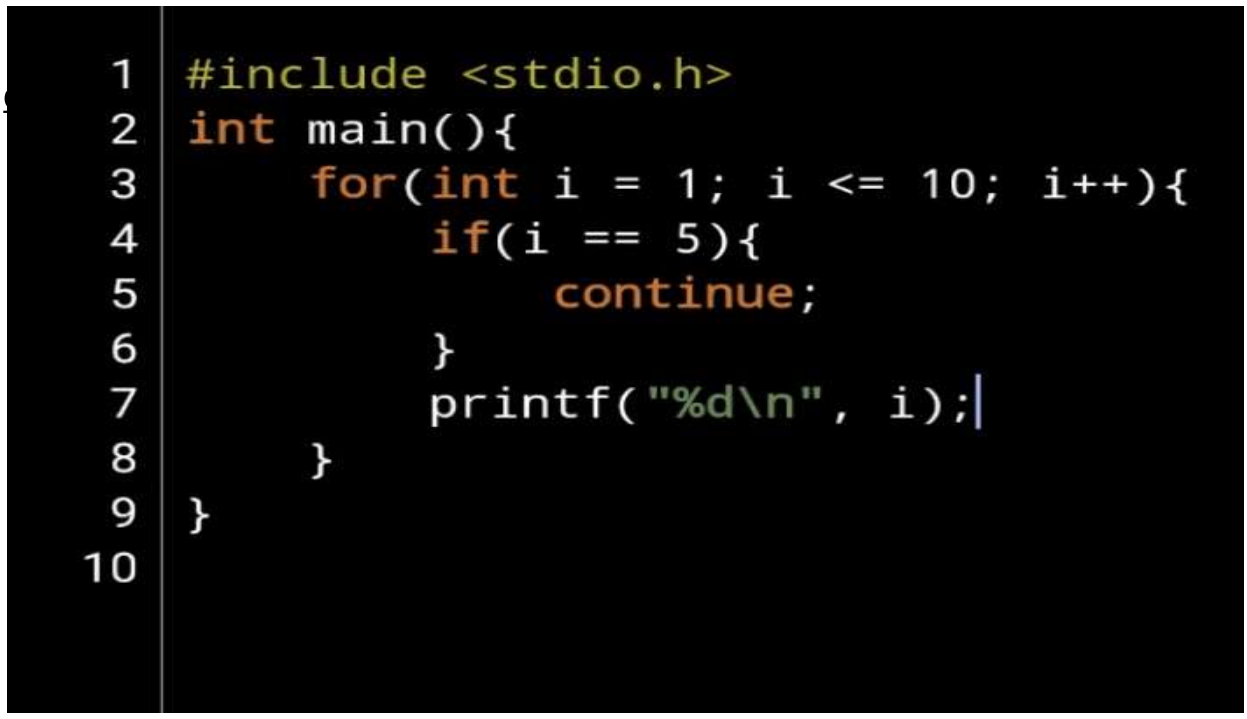
```
//Loop or switch case
break;
```

continue statement:

Syntax:

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

Example:



```
#include <stdio.h>
int main(){
    for(int i = 1; i <= 10; i++){
        if(i == 5){
            continue;
        }
        printf("%d\n", i);
    }
}
```

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

goto Statement:

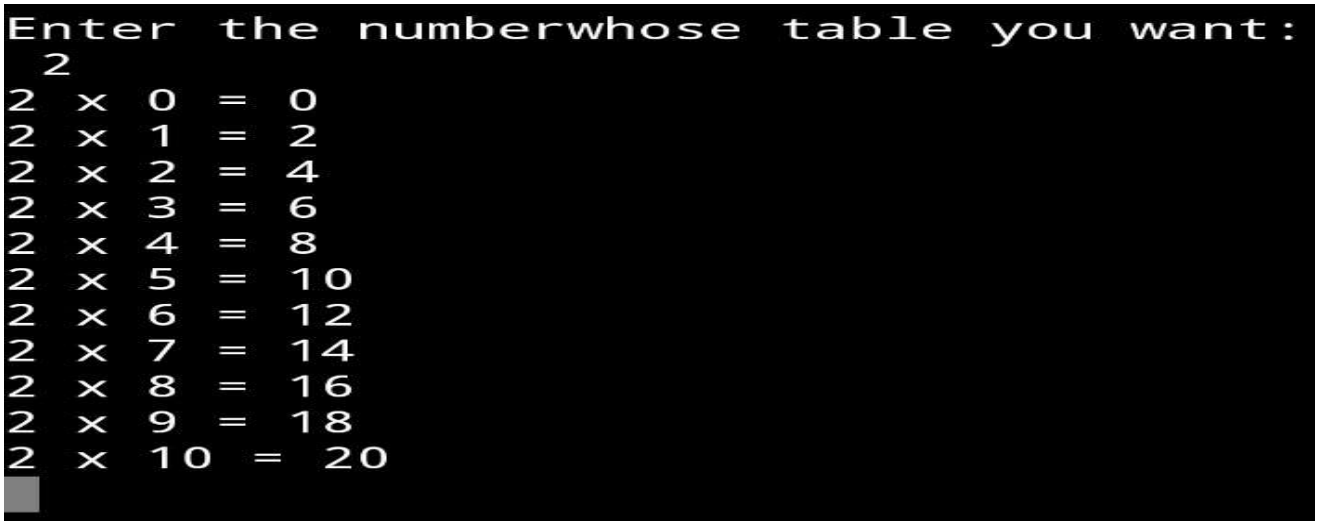
Syntax:

label:
//Some lines of code;
goto label;

Example:

```c
#include <stdio.h>
int main(){
    int i = 0, num;
    printf("Enter the number whose table you want: ");
    scanf("%d",& num);
    table:
    printf("%d x %d = %d\n", num, i ,num*i);
    i++;
    if(i <= 10)
        goto table;

}
```

Output:



Type-casting in c:

Definition:*It is nothing but the new way of converting a data type of one variable to other data type.*

Syntax:

```
(type) value;
```

Example:

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

Output:

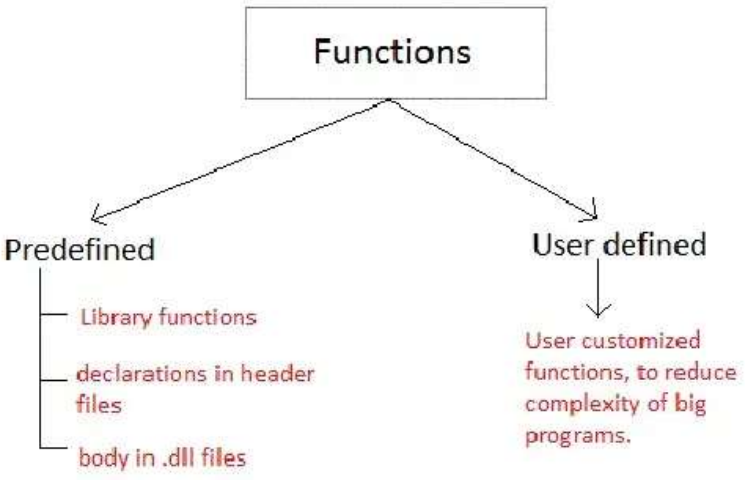
f: 2.250000

C Functions:

SN	C function aspects	Syntax
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

Syntax:

```
return _type function_name(type1 parameter , type2 parameter .....){
//code to be executed
}
```



Library Functions:

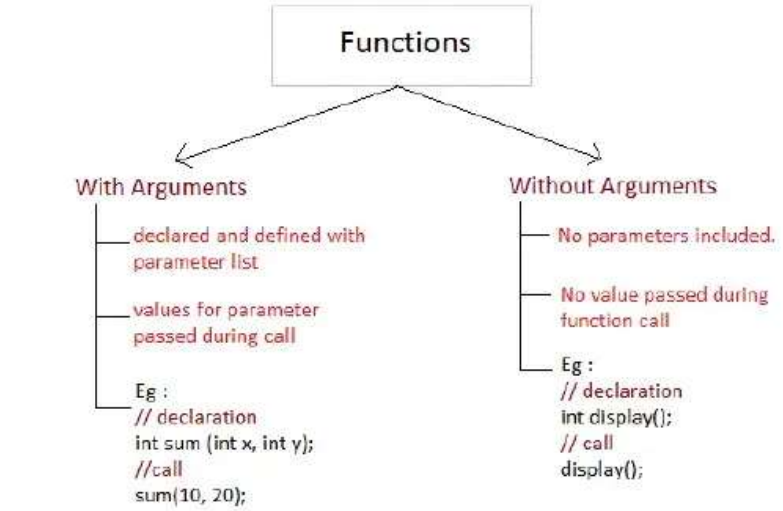
Are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.

The list of mostly used header files are given below in the 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	stdarg.h	Variable argument functions are defined in this header file.
9	signal.h	All the signal handling 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.

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.




```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ...
    result = multiply(i, j);
    ... ...
}

int multiply(int a, int b)
{
    ... ...
}
```

providing arguments while calling function

Return value:

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.

Example:

```
int get():
    return 10;
```

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

Example:

```
#include<stdio.h>

void main ()
{
 int get();
    return 10;

printf("%d", get());
}
```

OUTPUT:

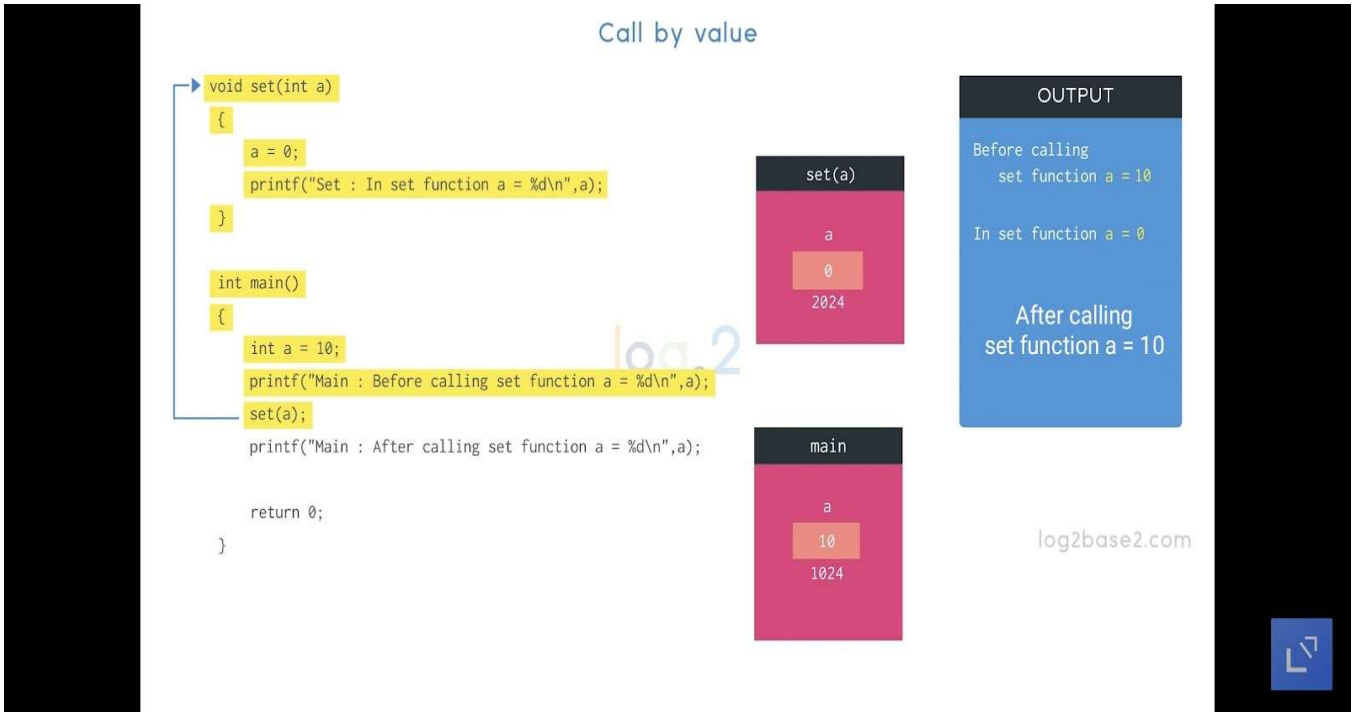
10

Different aspects of function calling:

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

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

Call by value:



Example:

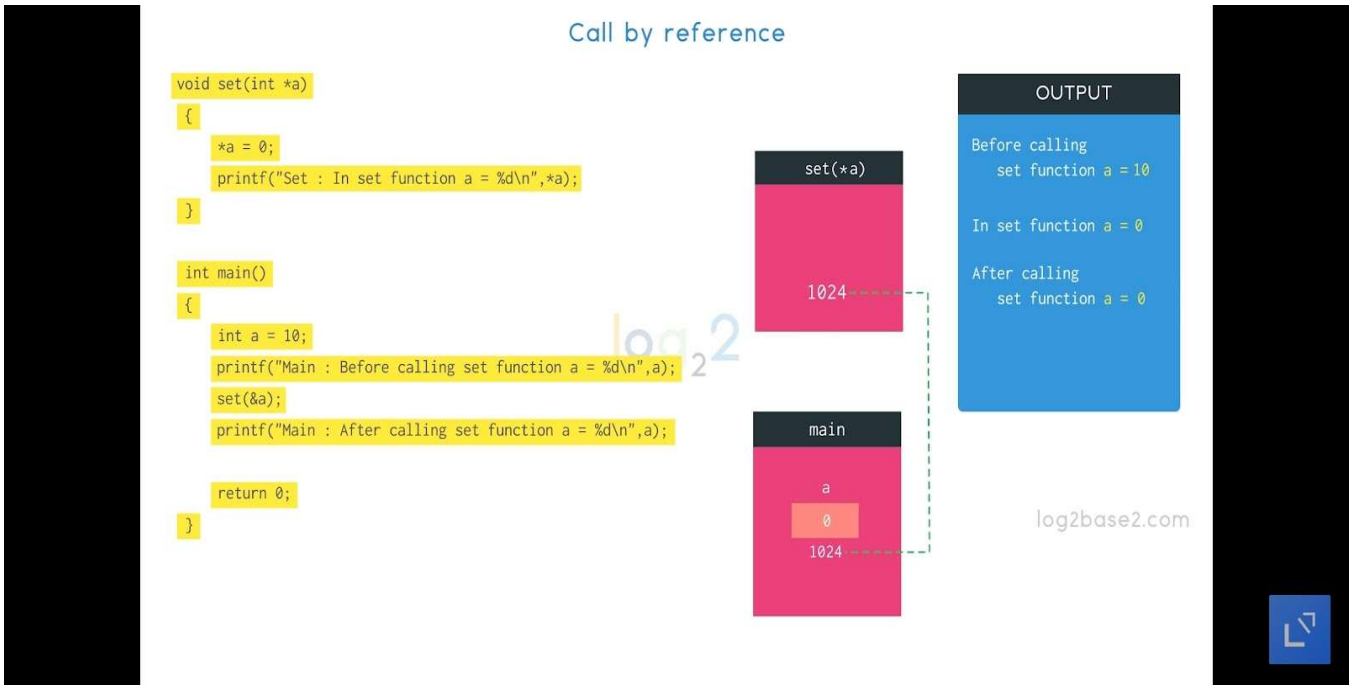
```
#include<stdio.h>
void change(int num) {
   printf("Before adding value inside function num=%d \n",num);
   num=num+100;
   printf("After adding value inside function num=%d \n", num);
}
int main() {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(x);//passing value in function
   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=200
```

→Call by value doesn't effect the main function although you made made changes to set(in above pic) function.

Call by Reference:



Example:

```
#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;
   }
```

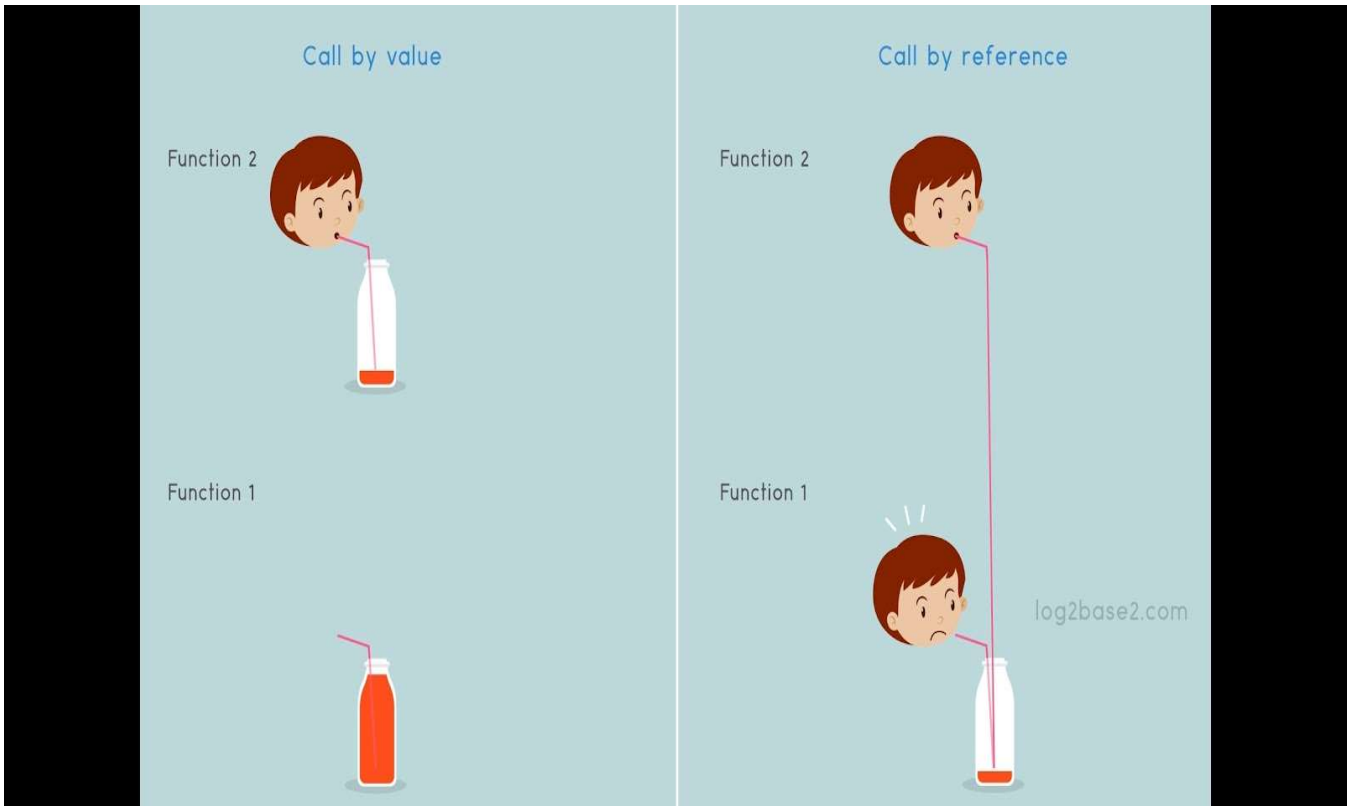
OUTPUT:

```
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 effects the changes you made in set(in above pic) function same as in main function.

Call by Value (vs) Call by Reference:



Recursion in C:

Any function which calls itself is called recursive function, and such function calls are called recursive calls.

Example:

NUMBER FACTORIAL USING RECURSION:

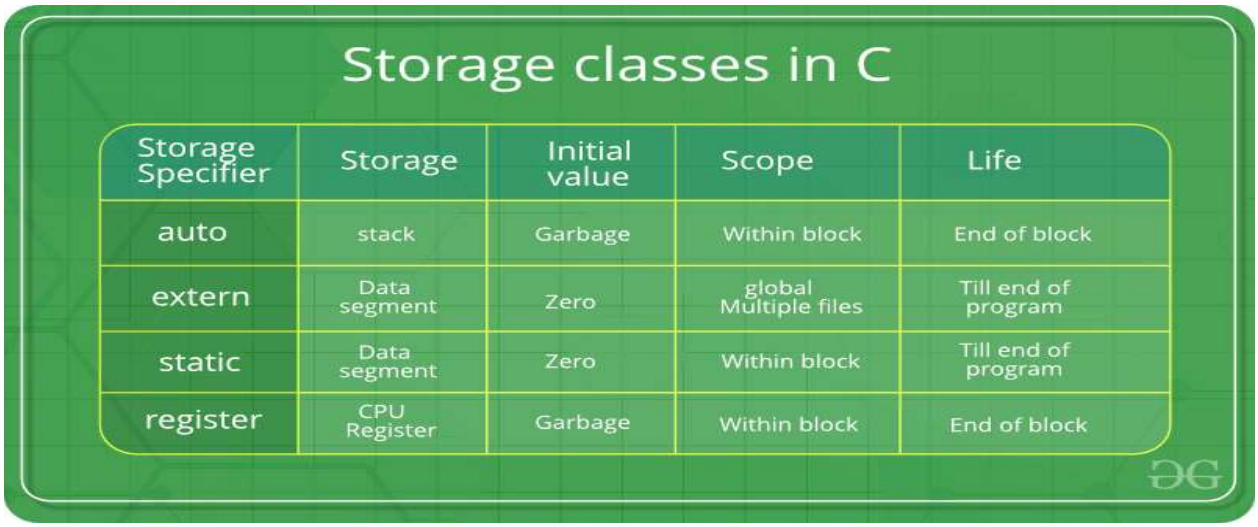
```c
#include <stdio.h>

unsigned long long int factorial(unsigned int i) {

  if(i <= 1) {
     return 1;
  }
  return i * factorial(i - 1);
}

int  main() {
  int i = 12;
  printf("Factorial of %d is %d\n", i, factorial(i));
  return 0;
}
```

OUTPUT:

```
Factorial of 12 is 479001600
```

Storage Classes in C:

C language uses 4 storage classes, namely:



Syntax:
storage_class var_data_type *var_name*;

Let's cover with example with all the storage classes:

Example:

```
// A C program to demonstrate different storage
// classes
#include <stdio.h>
 // declaring the variable which is to be made extern
// an initial value can also be initialized to x
int x;
 void autoStorageClass()
{
    printf("\nDemonstrating auto class\n\n");
    // declaring an auto variable (simply
   // writing "int a=32;" works as well)
   auto int a = 32;
    // printing the auto variable 'a'
   printf("Value of the variable 'a'"
          " declared as auto: %d\n",
          a);
    printf("--------------------------------");
}
 void registerStorageClass()
{
    printf("\nDemonstrating register class\n\n");
    // declaring a register variable
   register char b = 'G';
    // printing the register variable 'b'
   printf("Value of the variable 'b'"
          " declared as register: %d\n",
          b);
    printf("--------------------------------");
}
 void externStorageClass()
{
    printf("\nDemonstrating extern class\n\n");
    // telling the compiler that the variable
   // x is an extern variable and has been
```

```
   // defined elsewhere (above the main
   // function)
   extern int x;
    // printing the extern variables 'x'
   printf("Value of the variable 'x'"
          " declared as extern: %d\n",
          x);
    // value of extern variable x modified
   x = 2;
    // printing the modified values of
   // extern variables 'x'
   printf("Modified value of the variable 'x'"
          " declared as extern: %d\n",
          x);
    printf("--------------------------------");
}
 void staticStorageClass()
{
   int i = 0;
    printf("\nDemonstrating static class\n\n");
    // using a static variable 'y'
   printf("Declaring 'y' as static inside the loop.\n"
          "But this declaration will occur only"
          " once as 'y' is static.\n"
          "If not, then every time the value of 'y' "
          "will be the declared value 5"
          " as in the case of variable 'p'\n");
    printf("\nLoop started:\n");
    for (i = 1; i < 5; i++) {
        // Declaring the static variable 'y'
       static int y = 5;
        // Declare a non-static variable 'p'
       int p = 10;
        // Incrementing the value of y and p by 1
       y++;
       p++;
        // printing value of y at each iteration
       printf("\nThe value of 'y', "
              "declared as static, in %d "
              "iteration is %d\n",
```

```
            i, y);
        // printing value of p at each iteration
       printf("The value of non-static variable 'p', "
              "in %d iteration is %d\n",
              i, p);
   }
    printf("\nLoop ended:\n");
    printf("---------------------------------");
}
 int main()
{
    printf("A program to demonstrate"
          " Storage Classes in C\n\n");
    // To demonstrate auto Storage Class
   autoStorageClass();
    // To demonstrate register Storage Class
   registerStorageClass();
    // To demonstrate extern Storage Class
   externStorageClass();
    // To demonstrate static Storage Class
   staticStorageClass();
    // exiting
   printf("\n\nStorage Classes demonstrated");
    return 0;
}
 // This code is improved by KOMMI DRUTHENDRA NADH CHOWDARY
```

OUTPUT:

A program to demonstrate Storage Classes in C

Demonstrating auto class

Value of the variable 'a' declared as auto: 32

Demonstrating register class

Value of the variable 'b' declared as register: 71

Demonstrating extern class

Value of the variable 'x' declared as extern: 0

Modified value of the variable 'x' declared as extern: 2

Demonstrating static class

Declaring 'y' as static inside the loop.
But this declaration will occur only once as 'y' is static.
If not, then every time the value of 'y' will be the declared value 5 as in the case of variable 'p'

Loop started:

The value of 'y', declared as static, in 1 iteration is 6
The value of non-static variable 'p', in 1 iteration is 11

The value of 'y', declared as static, in 2 iteration is 7
The value of non-static variable 'p', in 2 iteration is 11

The value of 'y', declared as static, in 3 iteration is 8
The value of non-static variable 'p', in 3 iteration is 11

The value of 'y', declared as static, in 4 iteration is 9
The value of non-static variable 'p', in 4 iteration is 11

Loop ended:

C ARRAYS:

1-D Array:

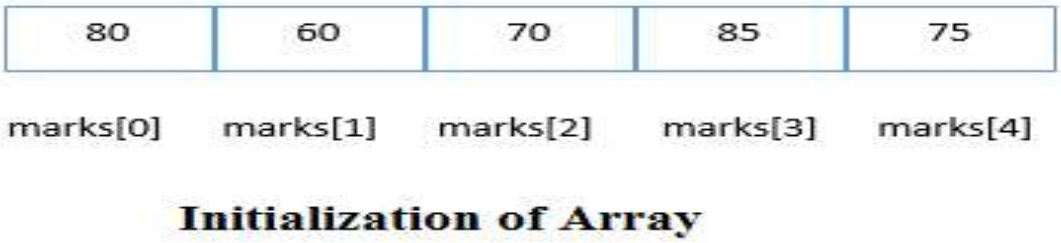
Syntax:
data_type array_name[array_size];

Initialization of C Array:

The simplest way to initialize an 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.

1. marks[0]=80;//initialization of array
2. marks[1]=60;

3. marks[2]=70;
4. marks[3]=85;
5. marks[4]=75;



Initialization of Array

Declaration with Initialization:

Example:

```c
#include <stdio.h>

int main(){
   int num[5] = {10, 15, 20, 25, 30};
   // Here we want to specify how many characters in array.
   for(int i; i<= 5; i++){
       printf("%d\n", num[i]);
   }
   return 0;
}
```

OUTPUT:

10
15
20
25
30

Sorting an array:

Example:

```c
#include <stdio.h>

void main(){
   int i,j,temp;
   int num[5] = {10, 24, 2, 16, 7};
   for(int i; i<5;i ++){
       for(int j = i + 1; j<5; j ++){
           if(num[j]>num[i]){
               temp = num[i];
               num[i] = num[j];
               num[j] = temp;
           }
       }
   }
   for(i; i<5; i++){
       printf("%d\n", num[i]);
   }
}
```

OUPUT:

24
16
10
7
2

2-D Array:

Syntax:

data_type array_name[rows][columns];

Example:

Storing elements in matrix and printing it.

```c
   #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]);
            }
        }
    }
    #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]: 2

Enter a[0][1]: 3
Enter a[0][2]: 4
Enter a[1][0]: 1
Enter a[1][1]: 9
Enter a[1][2]: 6
Enter a[2][0]: 3
Enter a[2][1]: 1
Enter a[2][2]: 3

printing the elements ....

2 3 4
1 9 6
3 1 3

Return an array in-C:

Example:

```c
#include <stdio.h>
void getarray(int arr[])
{
   for(int i = 0; i<5 ; i++){
       printf("%d\n", arr[i]);
   }
}

   int main(){
       int arr[5] = {1, 546, 94, 45, 34};
       getarray(arr);
       return 0;
   }
```

OUTPUT:

1
546
94
45
34

Returning an Array as a pointer:

Example:

```
#include <stdio.h>

void pointrarray(int *arr){
   for(int i; i< 8; i++){
       printf("%c\t", arr[i]);
   }
}
int main(){
   int arr[8] = {'C', 'H', 'O', 'W', 'D', 'A', 'R', 'Y'};
   pointrarray(arr);
   return 0;
}
```

OUTPUT:

C H O W D A R Y

Return an array from a function returning pointer pointing to the array:

Example:

```
   #include <stdio.h>
   int *getarray()
   {
       int arr[5];
       printf("Enter the elements in an array : ");
       for(int i=0;i<5;i++)
       {
           scanf("%d", &arr[i]);
       }
       return arr;
   }
   int main()
   {
     int *n;
     n=getarray();
     printf("\nElements of array are :");
     for(int i=0;i<5;i++)
```

```
        {
            printf("%d", n[i]);
        }
        return 0;
    }
```

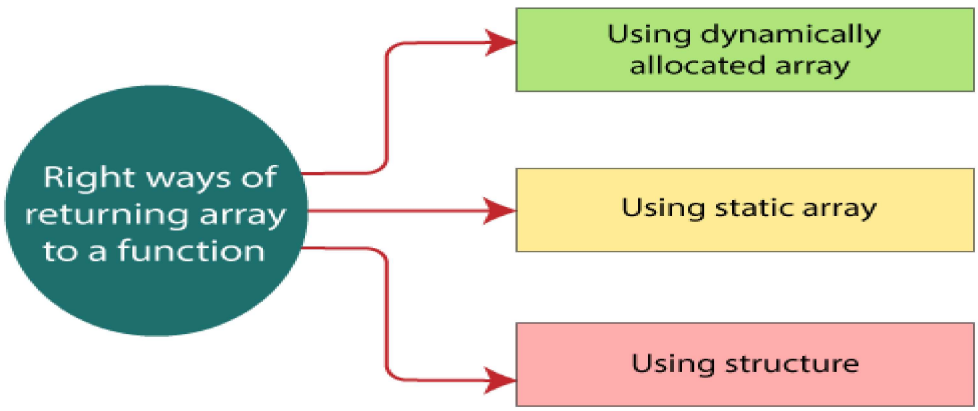
In the above program, getarray() function returns a variable 'arr'. It returns a local variable, but it is an illegal memory location to be returned, which is allocated within a function in the stack. Since the program control comes back to the main() function, and all the variables in a stack are freed. Therefore, we can say that this program is returning memory location, which is already deallocated, so the output of the program is a segmentation fault.

Output:

```
input
main.c:27:12: warning: function returns address of local variable [-Wreturn-local-addr]
Array inside function: 1
2
3
4
5
Array outside function:
Segmentation fault (core dumped)
```

There are three right ways of returning an array to a function:

- Using dynamically allocated array
- Using static array
- Using structure



Returning array by passing an array which is to be returned as a parameter to the function.

Example:

```c
#include <stdio.h>

void *getarray(int *a){
   printf("Enter the elements  of the array: \n");
   for(int i; i<5; i++){
       scanf("%d", &a[i]);
   }
   return a;
}
int main(){
   int a[5];
   int *n;
   n = getarray(a);
   printf("The elements of the array is: \n");
   for(int j; j<5 ; j++){
       printf("%d\t", n[j]);
   }
   return 0;
}
```

Output:

Enter the elements of the array:
1
8
4
8
4
The elements of the array is:
1 8 4 8 4

Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc():

There are 4 library functions provided by C defined under ***<stdlib.h>*** header file to facilitate dynamic memory allocation in C programming. They are:

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

C malloc() method:

The **“malloc”** or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

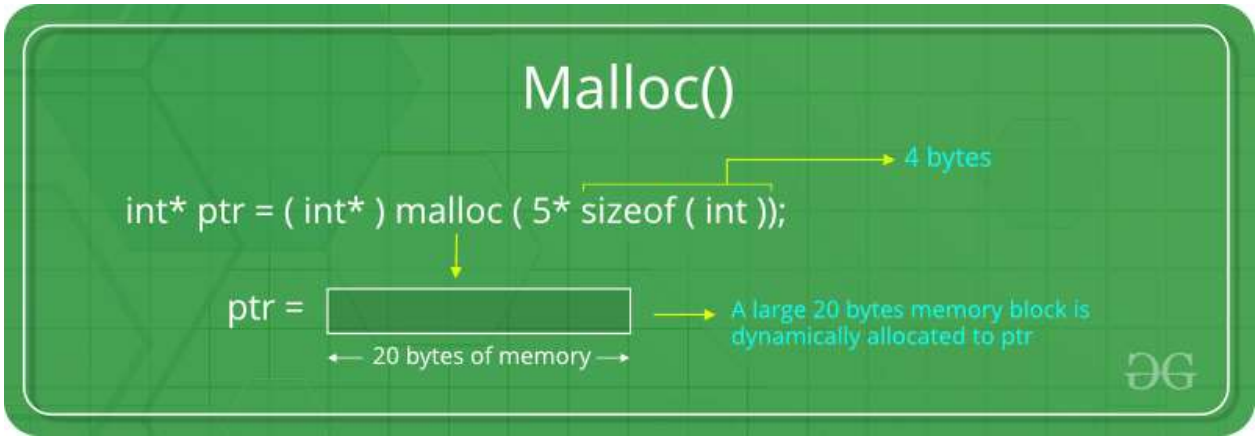
Syntax:

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

For Example:

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.



If space is insufficient, allocation fails and returns a NULL pointer.

Example of malloc() in C:

```
#include <stdio.h>
#include <stdlib.h>
```

```c
int main()
{

   // This pointer will hold the
   // base address of the block created
   int* ptr;
   int n, i;

   // Get the number of elements for the array
   printf("Enter number of elements:");
   scanf("%d",&n);
   printf("Entered number of elements: %d\n", n);

   // Dynamically allocate memory using malloc()
   ptr = (int*)malloc(n * sizeof(int));

   // Check if the memory has been successfully
   // allocated by malloc or not
   if (ptr == NULL) {
       printf("Memory not allocated.\n");
       exit(0);
   }
   else {

       // Memory has been successfully allocated
       printf("Memory successfully allocated using malloc.\n");

       // Get the elements of the array
       for (i = 0; i < n; ++i) {
           ptr[i] = i + 1;
       }

       // Print the elements of the array
       printf("The elements of the array are: ");
       for (i = 0; i < n; ++i) {
           printf("%d, ", ptr[i]);
       }
   }
```

```
    return 0;
}
```

Output:
Enter number of elements:3
Entered number of elements: 3
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3

C calloc() method:

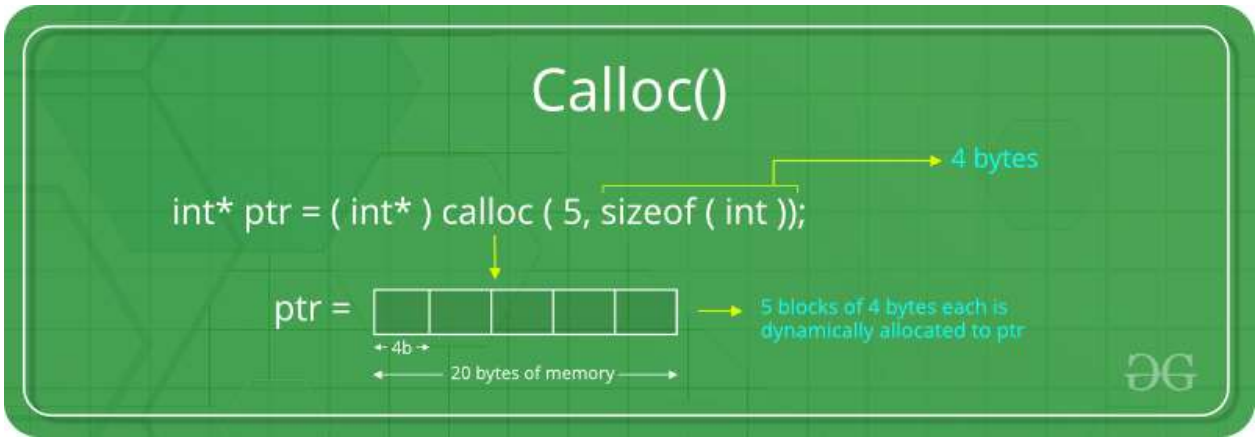
1. **“calloc”** or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

Syntax:

ptr = (cast-type*)calloc(n, element-size);
here, n is the no. of elements and element-size is the size of each element.

For Example:

ptr = (float*) calloc(25, sizeof(float));
This statement allocates contiguous space in memory for 25 elements each with the size of the float.



If space is insufficient, allocation fails and returns a NULL pointer.

Example of calloc() in C:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    printf("Enter number of elements: \n");
    scanf("%d", &n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
```

```
    }

    return 0;
}
```

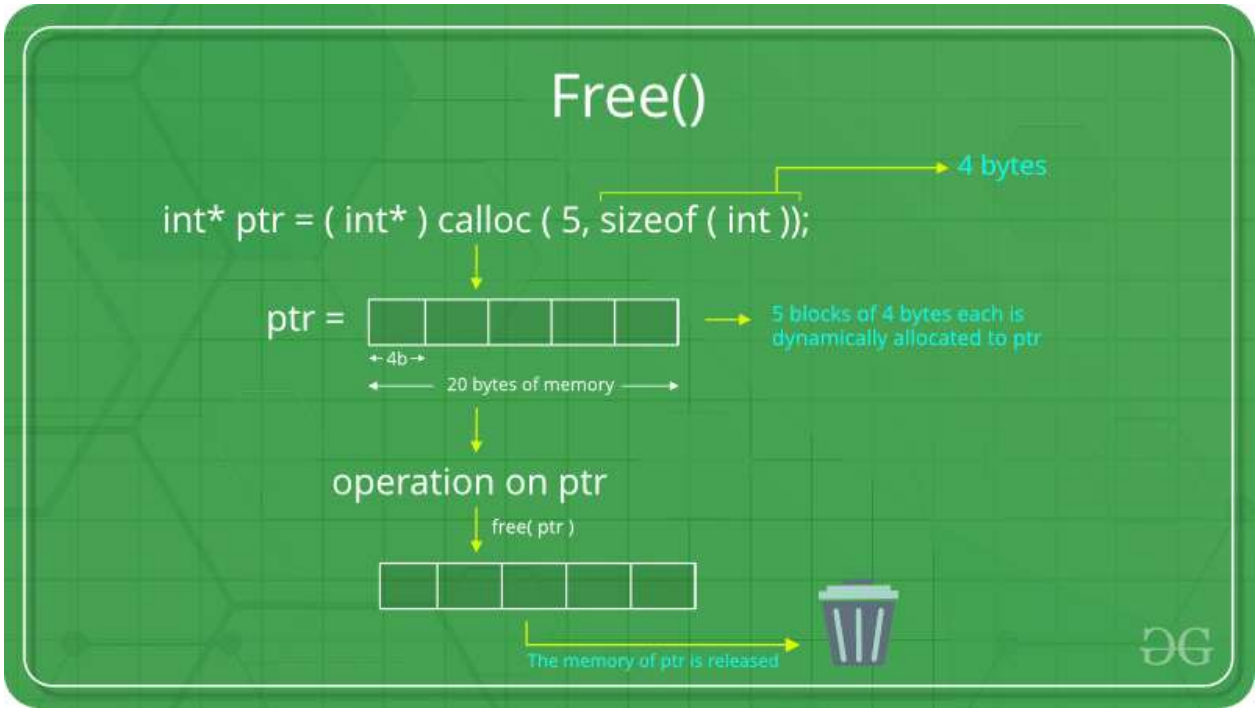
Output:
Enter number of elements:
5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5

C free() method:

“free” method in C is used to dynamically **de-allocate** the memory**.** The memory allocated using functions malloc() and calloc() is not de-allocated on their own**.** Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it**.**

Syntax:

free(ptr);



Example of free() in C:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{

    // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i;

    // Get the number of elements for the array

    printf("Enter number of elements: %d\n");
    scanf("%d", &n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Dynamically allocate memory using calloc()
    ptr1 = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Free the memory
        free(ptr);
        printf("Malloc Memory successfully freed.\n");

        // Memory has been successfully allocated
        printf("\nMemory successfully allocated using calloc.\n");
```

```
        // Free the memory
        free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }

    return 0;
}
```

Output:
Enter number of elements: 5
Memory successfully allocated using malloc.
Malloc Memory successfully freed.

Memory successfully allocated using calloc.
Calloc Memory successfully freed.

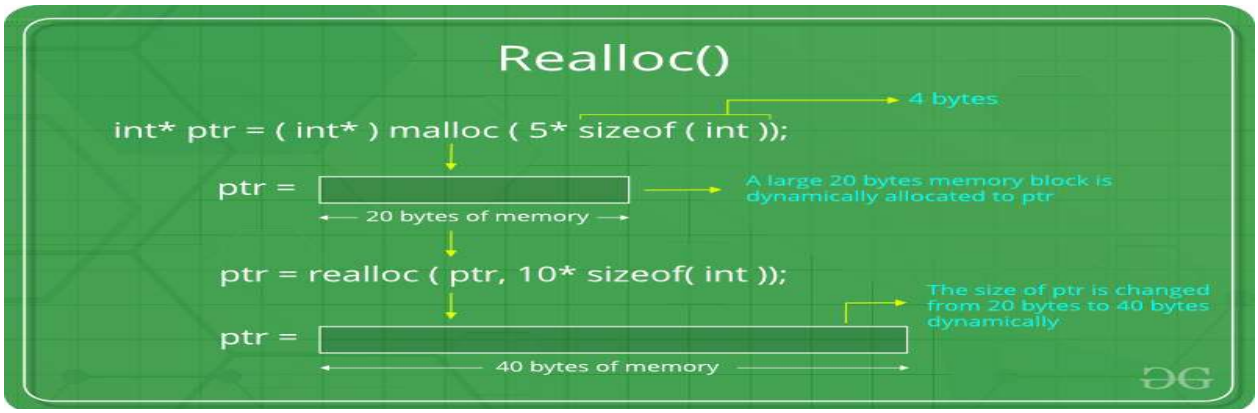
C realloc() method:

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory.** re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax of realloc() in C

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.



If space is insufficient, allocation fails and returns a NULL pointer.

Example of realloc() in C:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);


    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
```

```
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        // Get the new size for the array
        n = 10;
        printf("\n\nEnter the new size of the array: %d\n", n);

        // Dynamically re-allocate memory using realloc()
        ptr = realloc(ptr, n * sizeof(int));

        // Memory has been successfully allocated
        printf("Memory successfully re-allocated using realloc.\n");

        // Get the new elements of the array
        for (i = 5; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }

    return 0;
}
```

Output:

Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10
Memory successfully re-allocated using realloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Returning array using malloc() function:

Example:

```c
#include <stdio.h>
#include <stdlib.h>

void *sync(){
   int n, i;
   printf("Enter the number of elements: \n");
   scanf("%d", &n);
   int *ptr = (int*)malloc(n*sizeof(int));
   for(i; i<n; ++i){
       ptr[i] = i + 1;
   }
   return ptr;
}
int main(){
   int *k;
   k = sync();
   for(int j = 0;k[j] != '\0'; ++j){
       printf("%d\t", k[j]);

   }
   return 0;
}
```

Output:

Enter the number of elements:
5
1 2 3 4 5

Using Static Variable:

Example:

```c
#include <stdio.h>
#include <stdlib.h>

void *sync(){
   int n;
   static int arr[5];
   for(int i; i < 5; ++i){
```

```
        arr[i] = i + 1;
    }
    return arr;
}
int main(){
    int *k;
    k = sync();
    for(int j; k[j] != '\0'; ++j){
        printf("%d\t",k[j]);
    }
    return 0;
}
```

Output:

1 2 3 4 5

Using Structure:

The structure is a user-defined data type that can contain a collection of items of different types. To access the structure, you must create a variable of it.

Example:

```
#include <stdio.h>
#include<malloc.h>
struct array
{
    int arr[8];
};
struct array getarray()
{
    struct array y;
    printf("Enter the elements in an array : ");
    for(int i=0;i<8;i++)
    {
        scanf("%d",&y.arr[i]);
    }
    return y;
}
int main()
{
 struct array x=getarray();
```

```
 printf("Elements that you have entered are :");
 for(int i=0;x.arr[i]!='\0';i++)
 {
     printf("%d ", x.arr[i]);
 }
}
```

Output:

Enter the elements in an array : 4
6
8
9
7
9
4
5
Elements that you have entered are :4 6 8 9 7 9 4 5

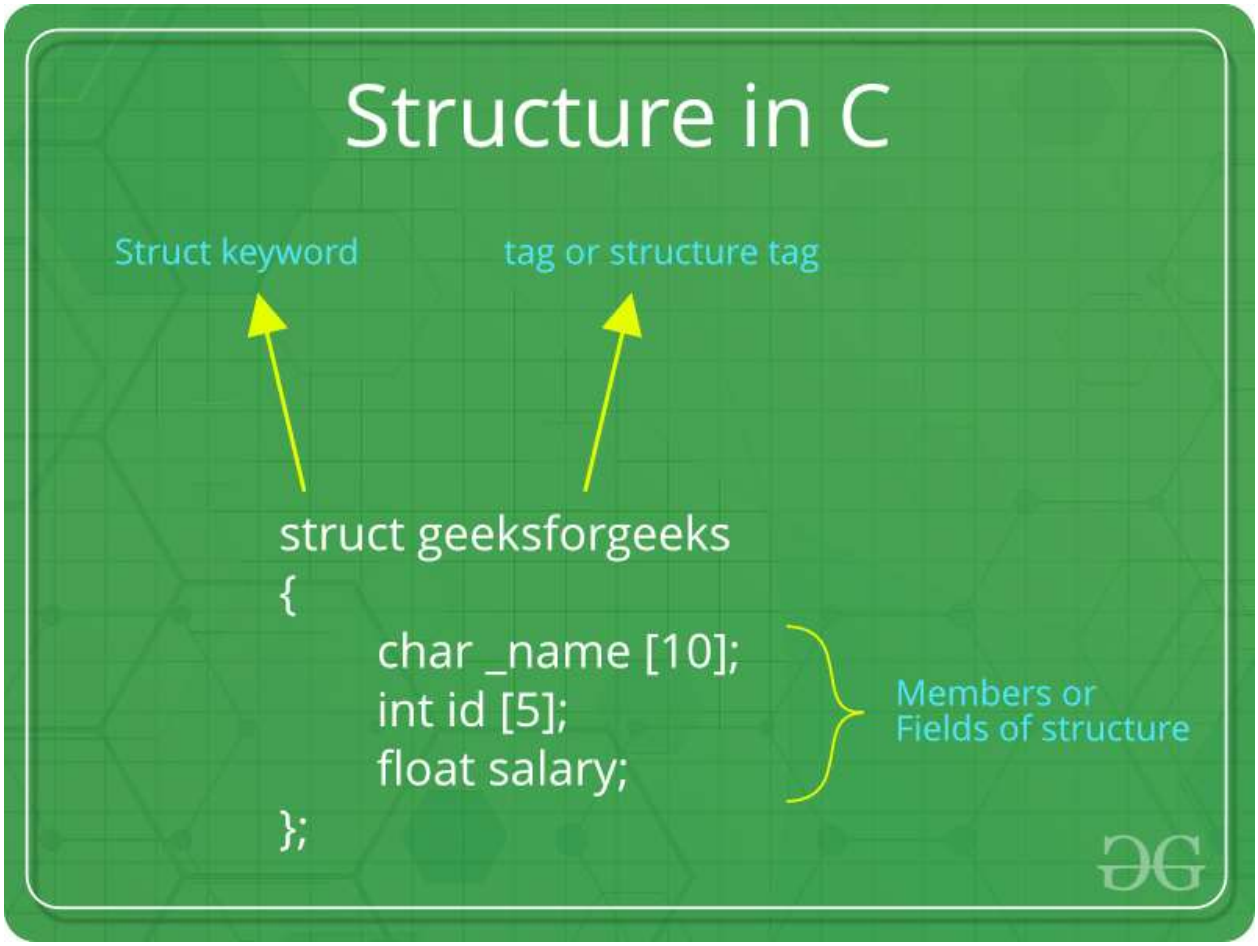
Structures:

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure.

Unlike an array, a structure can contain many different data types (int, float, char, etc.).

Create a Structure:

You can create a structure by using the `struct` keyword and declare each of its members inside curly braces:



Syntax:

```
struct structure_name {
    data_type member_name1;
    data_type member_name1;
    ....
    ....
};
```

We can define structure variables using two methods:

1. Structure Variable Declaration with Structure Template

```
struct structure_name {
```

data_type member_name1;

data_type member_name1;

....

....

}***variable1, variable2, ...***;

2. Structure Variable Declaration after Structure Template

// structure declared beforehand

struct *structure_name* ***variable1, variable2***, ...;

Access Structure Members

We can access structure members by using the **(.) dot operator.**

Syntax

structure_name.member1;

strcuture_name.member2;

In the case where we have a pointer to the structure, we can also use the arrow operator to access the members.

Example:

```
// C program to illustrate the use of structures

#include <stdio.h>



// declaring structure with name str1

struct str1 {

   int i;

   char c;
```

```
    float f;
    char s[30];
};


// declaring structure with name str2
struct str2 {
    int ii;
    char cc;
    float ff;
} var; // variable declaration with structure template


// Driver code
int main()
{
    // variable declaration after structure template
    // initialization with initializer list and designated
    // initializer list
    struct str1 var1 = { 1, 'A', 1.00, "Mad_Drax" },
                 var2;
    struct str2 var3 = { .ff = 5.00, .ii = 5, .cc = 'a' };


    // copying structure using assignment operator
    var2 = var1;
```

```
   print("Struct 1:\n\ti = %d, c = %c, f = %f, s = %s\n",
        var1.i, var1.c, var1.f, var1.s);
   print("Struct 2:\n\ti = %d, c = %c, f = %f, s = %s\n",
        var2.i, var2.c, var2.f, var2.s);
   print("Struct 3\n\ti = %d, c = %c, f = %f\n", var3.ii,
        var3.cc, var3.ff);


   return 0;
}
```

Output:

Struct 1:

i = 1, c = A, f = 1.000000, s = Mad_Drax

Struct 2:

i = 1, c = A, f = 1.000000, s = Mad_Drax

Struct 3

i = 5, c = a, f = 5.000000

typedef for Structures:

The typedef is a keyword that is used to provide existing data types with a new name. The C typedef keyword is used to redefine the name of already existing data types.

Syntax:

typedef existing_name alias_name;

Example:

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

```
   INTEGER n = 5;
   printf("Here in place of int we can use INTEGER\n");
}
```

Output:

Here in place of int we can use INTEGER

typedef in structure:

Example:

```
#include <stdio.h>

typedef struct {
   char name[10];
   int age;
}std;
int main(){

   std one = {"Mad_Drax", 18};
   std two = {"EPIC444", 17};
   printf("name : %s\n", one.name);
   printf("age : %d\n", one.age);
   printf("name : %s\n", two.name);
   printf("age : %d\n", two.age);
   return 0;

}
```

Output:

name : Mad_Drax
age : 18
name : EPIC444
age : 17

typedef in pointers:

Example:

```
#include <stdio.h>
typedef int* ptr;
int main(){
   ptr var;
   *var = 10;
```

```
    printf("%d\n", *var);
    return 0;
}
```

Output:

10

typedef with Array:

Example:

```
// C program to implement typedef with array
#include <stdio.h>

typedef int Arr[4];

// Driver code
int main()
{
    Arr temp = { 10, 20, 30, 40 };
    printf("typedef using an array\n");

    for (int i = 0; i < 4; i++) {
        printf("%d ", temp[i]);
    }
    return 0;
}
```

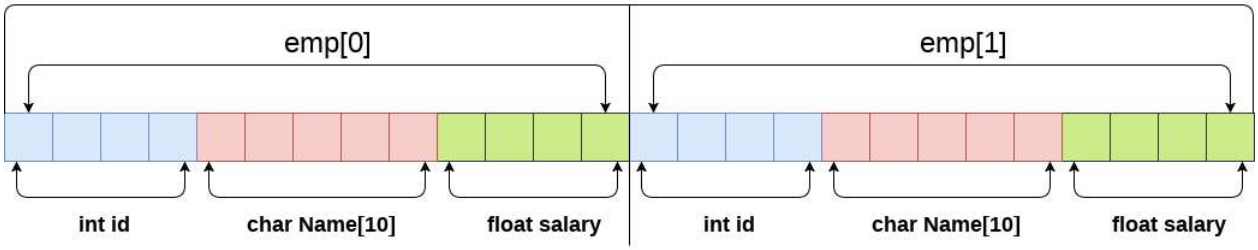
Output:

typedef using an array
10 20 30 40

Array of structures in c:

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

Array of structures




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

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

sizeof (emp[2]) = 26 bytes

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

```c
#include<stdio.h>

#include <string.h>

struct student{

int rollno;

char name[10];

};

int main(){

int i;

struct student st[5];

printf("Enter Records of 5 students");

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

printf("\nEnter Rollno:");
```

```
scanf("%d",&st[i].rollno);

printf("\nEnter Name:");

scanf("%s",&st[i].name);

}

printf("\nStudent Information List:");

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

printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);

}

   return 0;

}
```

Output:

Enter Rollno:1

Enter Name:kelly

Enter Rollno:2

Enter Name:alok

Enter Rollno:3

Enter Name:wukong

Enter Rollno:4

Enter Name:thiva

Enter Rollno:5

Enter Name:kla

Student Information List:
Rollno:1, Name:kelly
Rollno:2, Name:alok
Rollno:3, Name:wukong
Rollno:4, Name:thiva
Rollno:5, Name:kla

union in C:

A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time.

Syntax:

```
union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

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

Accessing Union Members:

→To access any member of a union, we use the member access operator (.).

Example:

```
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};
```

```
int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

Output:

```
data.i  :  1917853763
data.f  :  4122360580327794860452759994
368.000000
data.str  :  C Programming
```

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

Passing Array to Function in C:

Syntax:

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

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

First way:

1. return_type function(type arrayname[])

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

Second way:

1. return_type function(type arrayname[SIZE])

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

Third way:

1. return_type function(type *arrayname)

C Pointers:

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

→**One byte is equivalent to eight bits**←

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

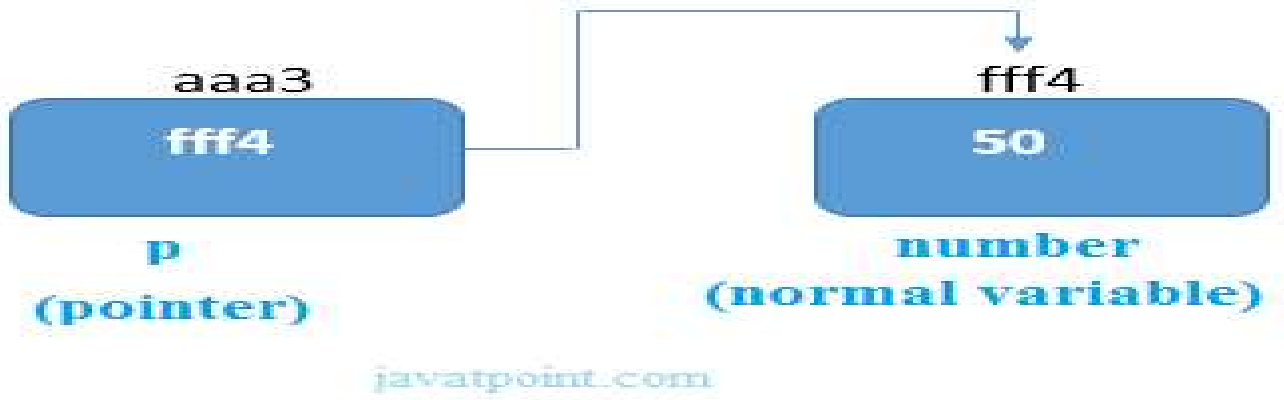
Example:

```
int n = 10;

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

Declaration of Pointer:

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

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



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

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

We can understand above information with an example:

```
#include <stdio.h>
int main(){
    int *p;
    int num = 50;//stores the address of num
    p = &num;
    printf("Address of p is %x\n", p);//To access the address of pointer we simply use p here.
    printf("value of num is %d", *p);//To access the value of pointer we use '*'.
}
```

Output:

```
Address of p is c8b554a4
value of num is 50
```

In above output c8b554a4 is the address of pointer which stores num variable of its value, And 50 is the value assigned to that num.

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.

Example:

```c
#include <stdio.h>

int main(){

    int a =10;

    printf("value of a is = %d and address of a is = %u\n", a, &a);

      int *p = &a;//Here operator '&' is used to point the address of variable 'a' to pointer 'p'.

    printf("value of pointer p is = %d and address of pointer p is = %u", *p, &p);

    return 0;

}
```

Output:

```
value of a is = 10 and address of a is = 6422300

value of pointer p is = 10 and address of pointer p is = 6422296
```

Pointer to array:

```c
int arr[10];

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

Example:

```c
#include <stdio.h>
void add(int a, int b){
    printf("Addition of two numbes is: %d", a+b);
}
void sub(int a, int b){
    printf("subtraction of two numbes is: %d", a-b);
```

```
}
void mul(int a, int b){
    printf("multiplication of two numbes is: %d", a*b);
}
void divi(int a, int b){
    printf("Division of two numbes is: %d", a/b);
}
void rem(int a, int b){
    printf("Remainder of two numbes is: %d", a%b);
}
int main(){
    unsigned int choice, a, b;
    void (*fun_ptr[])(int, int) = {add, sub, mul, divi, rem};
    printf("Enter the value of a: ");
    scanf("%d",& a);
    printf("Enter the value of b: ");
    scanf("%d",& b);
    printf("Enter choice : 0 for addition \n 1 for subtrcaction \n 2 for
multiplication \n 3 for division \n 4 for to get remainder");
    scanf("%d",& choice);
    if (choice > 4)
    {
        return 0;
    }
    (*fun_ptr[choice])(a, b);

    return 0;
}
```

Output:

Enter the value of a: 4
Enter the value of b: 5
Enter choice : 0 for addition
1 for subtrcaction
2 for multiplication
3 for division
4 for to get remainder2
multiplication of two numbes is: 20

Pointer to a function:

```
void show (int);

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

Example:

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
     void (*fun_ptr)(int) = fun;  // if we put & before the fun and *
before the fun_ptr in below line the result does not change.

    fun_ptr(10);

    return 0;
}
```

Output:

Value of a is 10

Lets look into another **Example:**

```
#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?3

5

The sum is 8

A function that receives a simple function:

```
// A simple C program to show function pointers as parameter

#include <stdio.h>



// Two simple functions

void fun1() { printf("Fun1\n"); }
```

```
void fun2() { printf("Fun2\n"); }



// A function that receives a simple function

// as parameter and calls the function

void wrapper(void (*fun)())

{

    fun();

}



int main()

{

    wrapper(fun1);

    wrapper(fun2);

    return 0;

}
```

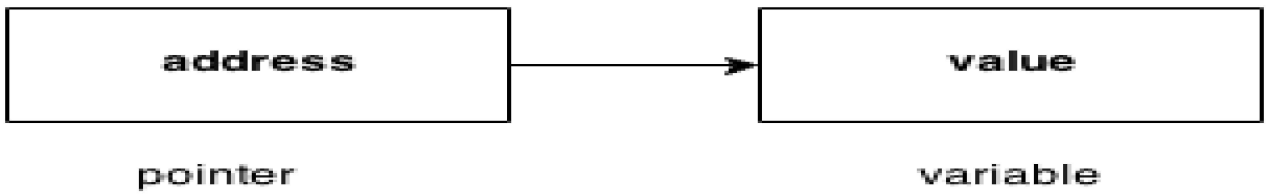
Output:

Fun1

Fun2

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.

NULL pointer:

A null pointer in C is a pointer that does not point to any location but NULL. It is used to initialize a pointer variable when that pointer variable hasn't been assigned any valid memory address yet. It is also used to check for a null pointer before accessing any pointer variable.

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

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

For more reference :Null Pointer(Click This).

Swapping two numbers with pointers:

```c
#include<stdio.h>
int main(){
    int a=5,b=10,*p1=&a, *p2=&b;
    *p1 = *p1 + *p2;
    *p2 = *p1 - *p2;
    *p1 = *p1 - *p2;
    printf("a = %d and b = %d", a, b);
    return 0;
}
```

Output:

```
a = 10 and b = 5
```

Reading complex pointers:

There are several things which must be taken into the consideration while reading the complex pointers in C. Lets see the precedence and 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,

- (): This operator is a bracket operator used to declare and define the function.
- []: This operator is an array subscript operator

- * : This operator is a pointer operator.
- Identifier: It is the name of the pointer. The priority will always be assigned to this.
- 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 which is right to left, so the priority goes to p, and the second priority goes to *.

Assign 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?

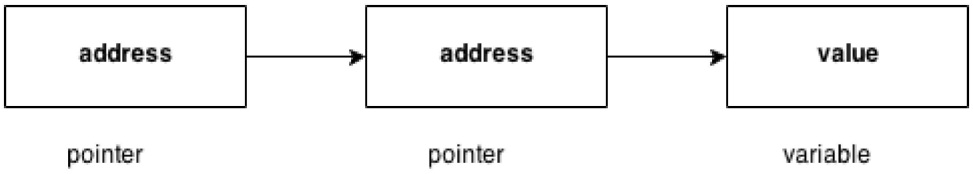
```
int (*p)(int (*)[2], int (*)void))
```

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 whose parameter is void and return type is the integer.

Double Pointer (Pointer to Pointer):

a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.



The syntax for declaring a double pointer is given below.

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

Let us understand the above information with a simple 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 stored at pp

}
```

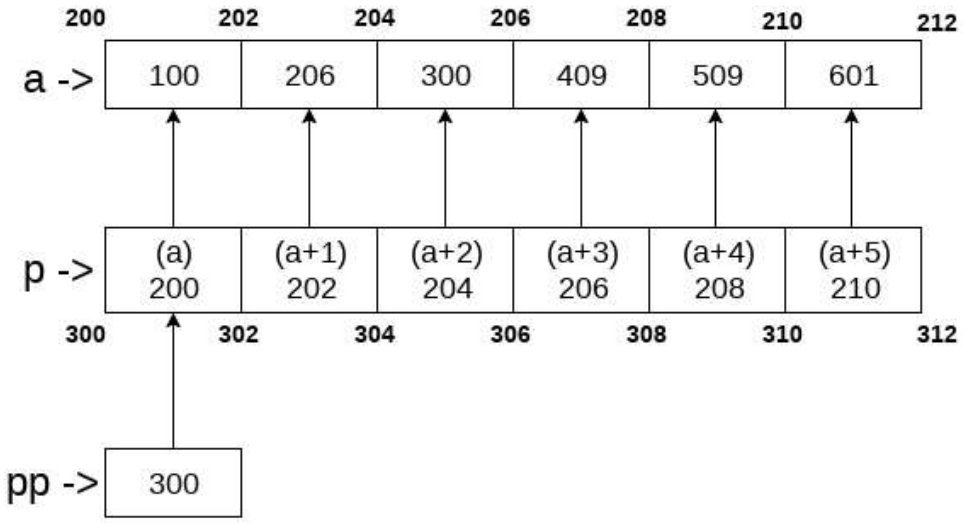
Output:

```
address of a: 61ff18

address of p: 61ff14

value stored at p: 10

value stored at pp: 10
```

Brief understanding with an example:

```
#include<stdio.h>

void main ()



   int a[10] = {100, 206, 300, 409, 509, 601}; //Line 1

   int *p[] = {a, a+1, a+2, a+3, a+4, a+5}; //Line 2

   int **pp = p; //Line 3

   pp++; // Line 4

   printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 5

   *pp++; // Line 6

   printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 7
```

```
++*pp; // Line 8

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); // Line 11
```



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

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. 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 the image, a[0] can be accessed in the following ways.

- a[0]: it is the simplest way to access the first element of the array

- *(a): since a store the address of the first element of the array, we can access its value by using indirection pointer on it.
- *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].
- **(pp): as 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. **p 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 prints three values, i.e., pp - p, *pp - a, **pp. Let's calculate them each one of them.

- pp = 302, p = 300 => pp-p = (302-300)/2 => pp-p = 1, i.e., 1 will be printed.
- pp = 302, *pp = 202, a = 200 => *pp - a = 202 - 200 = 2/2 = 1, i.e., 1 will be printed.
- 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.

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

- pp = 304, *pp = 204, a = 200 => *pp-a = (204 - 200)/2 = 2, i.e., 2 will be printed.
- 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 rule of 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.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
- 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 writen. 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.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
- 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:

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

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

Incrementing Pointer in C:

Syntax:

```
new_address= current_address + i * size_of(data type)
```

Here 'i' is the integer.

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

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

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.

Traversing an array by using pointer:

Example:

```
#include<stdio.h>
void main ()
{
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    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:

Syntax:

```
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.

C Pointer Addition:

Syntax:

```
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.

Lets see a example from 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 6422296

After adding 3: Address of p variable is 6422308
```

C Pointer Subtraction

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

Syntax:

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

32-bit

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

64-bit

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

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

```
#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; //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 6422296

After subtracting 3: Address of p variable is 6422284
```

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

If two pointers are of the same type,

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

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

```
#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 Array of functions in C:

→**Funtion pointer are like normal pointers but they have the capability of point to a function.**
For more reference (Youtube Link:https://youtu.be/BRsv3ZXoHto).
n C, an array of function pointers can be defined as follows:

```
return_type (*array_name[size])(parameter_list);
```

Here, `array_name` is the name of the array, `size` is the number of elements in the array, `return_type` is the data type of the function pointer that the array will hold, and `parameter_list` is a comma-separated list of parameters that the function pointer can accept.

Lets see deep understanding with a example:

```
#include<stdio.h>

int show();

int showadd(int);

int (*arr[3])();
```

```
int (*array[3])();



int (*(*ptr)[3])();

//int: This is the return type of the function.

//(*(*ptr)[3]): This is a pointer to an array of 3 pointers.



int main ()

{

    int result1;

     // Assigning the address of show() function to the first element of
arr[] array

    arr[0] = show;



     // Assigning the address of showadd() function to the second element
of arr[] array

    arr[1] = showadd;



    // Assigning the address of arr[] array to ptr pointer

    ptr = &arr;



    // Calling the first function using double indirection

    result1 = (**ptr)();



    // Printing the value returned by show()
```

```
    printf("printing the value returned by show : %d",result1);



     // Calling the second function using double indirection and passing
result1 as argument

    (*(*ptr+1))(result1);

}



// Defining show() function which returns an integer value

int show()

{

    int a = 65;

    return a++;  //Here in function returnig we cannot use ++ --.

}



// Defining showadd() function which takes an integer argument and returns
nothing

int showadd(int b)

{

    printf("\nAdding 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
```

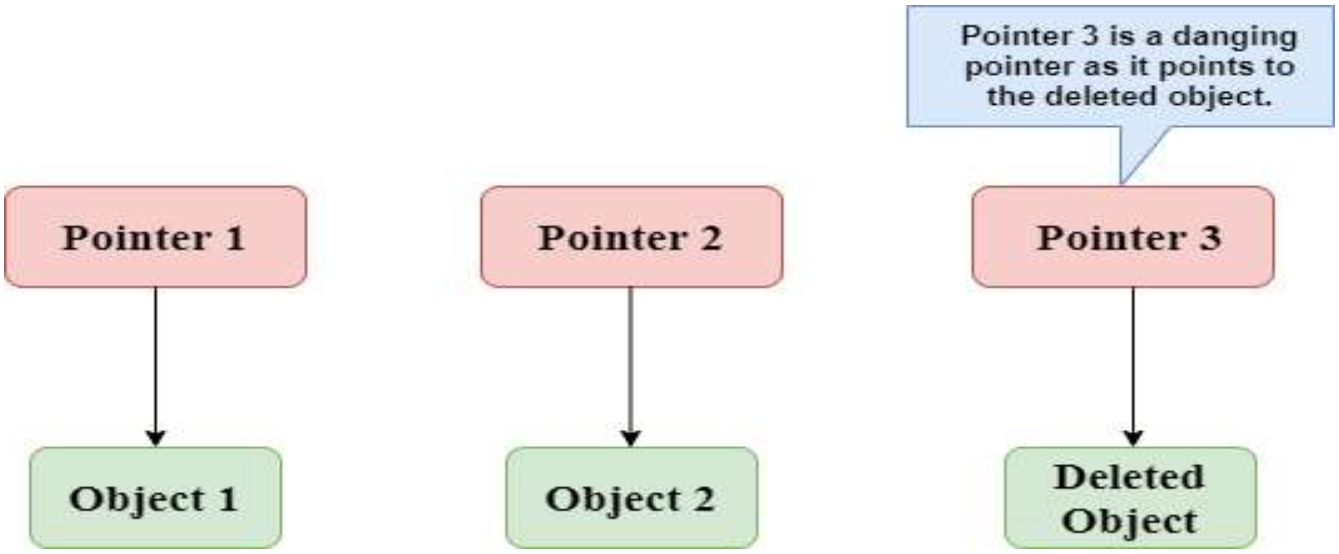
Explaination:

This is a C program that demonstrates the use of function pointers. The program declares an array of function pointers `arr` of size 3. The first element of the array is assigned the address of the function `show()`, and the second element is assigned the address of the function `showadd()`. The program then declares a pointer to an array of 3 function pointers `ptr`. The pointer `ptr` is assigned the address of the array `arr`. The program then calls the function pointed to by the first element of `ptr`, which is `show()`. The value returned by `show()` is stored in the variable `result1`. The program then calls the function pointed to by the second element of `arr`, which is `showadd()`, and passes it the value stored in `result1` as an argument. The function `show()` returns 65, which is incremented to 66 before being returned. The function `showadd()` adds 90 to its argument and prints the result on the console

Dangling Pointers in C:

Sometimes the programmer fails to initialize the pointer with a valid address, then this type of initialized pointer is known as a dangling pointer in C.

Let's observe the following examples.



In the above figure, we can observe that the **Pointer 3** is a dangling pointer. **Pointer 1** and **Pointer 2** are the pointers that point to the allocated objects, i.e., Object 1 and

Object 2, respectively. **Pointer 3** is a dangling pointer as it points to the de-allocated object.

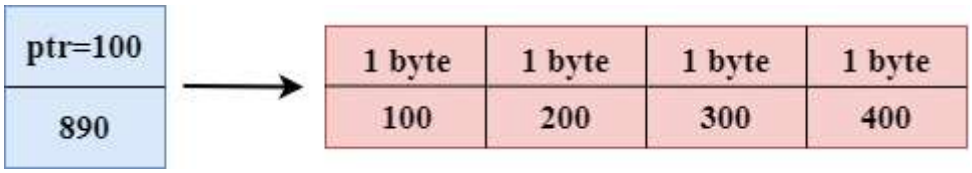
Let's understand the dangling pointer through some C programs.

Using free() function to de-allocate the memory.

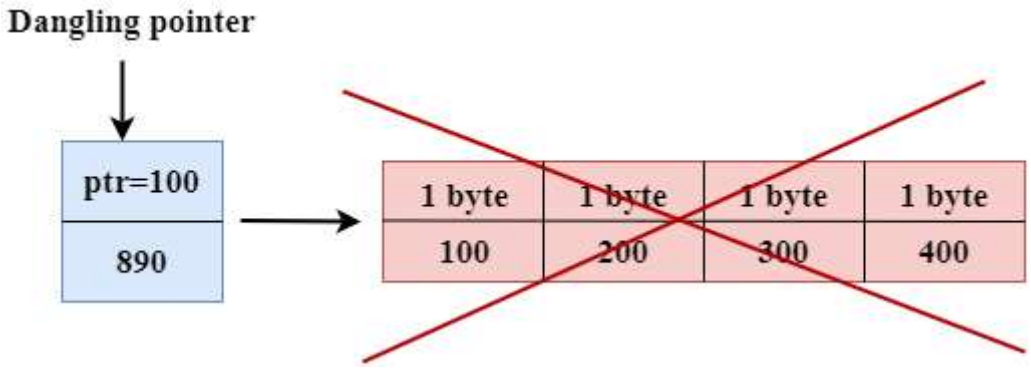
In the above code, we have created two variables, i.e., *ptr and a where 'ptr' is a pointer and 'a' is a integer variable. The *ptr is a pointer variable which is created with the help of **malloc()** function. As we know that malloc() function returns void, so we use int * to convert void pointer into int pointer.

```
#include <stdio.h>

int main()

{

   int *ptr=(int *)malloc(sizeof(int));

   int a=560;

   ptr=&a;

   free(ptr);

   return 0;

}
```

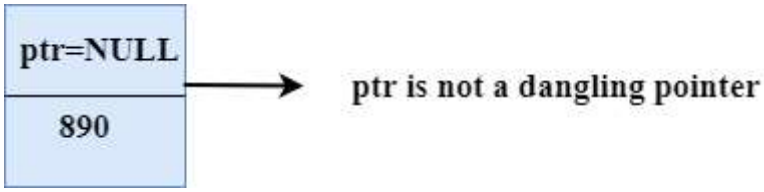
The statement **int *ptr=(int *)malloc(sizeof(int));** will allocate the memory with 4 bytes shown in the below image:



The statement **free(ptr)** de-allocates the memory as shown in the below image with a cross sign, and 'ptr' pointer becomes dangling as it is pointing to the de-allocated memory.



If we assign the NULL value to the 'ptr', then 'ptr' will not point to the deleted memory. Therefore, we can say that ptr is not a dangling pointer, as shown in the below image:



Avoiding Dangling Pointer Errors

The dangling pointer errors can be avoided by initializing the pointer to the NULL value. If we assign the NULL value to the pointer, then the pointer will not point to the de-allocated memory. Assigning NULL value to the pointer means that the pointer is not pointing to any memory location.

.sizeof() operator in C:

It determines the size of the expression or the data type specified in the number of char-sized storage units. The **sizeof()** operator contains a single operand which can be either an expression or a data typecast where the cast is data type enclosed within parenthesis. The data type cannot only be primitive data types such as integer or floating data types, but it can also be pointer data types and compound data types such as unions and structs.

The **sizeof()** operator behaves differently according to the type of the operand.

- **Operand is a data type**
- **Operand is an expression**

When operand is a data type:

Example:

```
#include <stdio.h>
int main()
{
    int x=89;    // variable declaration.
    printf("size of the variable x is %d", sizeof(x));  // Displaying the
size of ?x? variable.
    printf("\nsize of the integer data type is %d",sizeof(int));
//Displaying the size of integer data type.
    printf("\nsize of the character data type is %d",sizeof(char));
//Displaying the size of character data type.

    printf("\nsize of the floating data type is %d",sizeof(float));
//Displaying the size of floating data type.
return 0;
}
```

In the above code, we are printing the size of different data types such as int, char, float with the help of **sizeof()** operator.

Output

```
size of the variable x is 4
size of the integer data type is 4
size of the character data type is 1
size of the floating data type is 4
```

When operand is an expression:

```
#include <stdio.h>

int main()

{
```

```
  double i=78.0; //variable initialization.

  float j=6.78; //variable initialization.

  printf("size of (i+j) expression is : %d",sizeof(i+j)); //Displaying the
size of the expression (i+j).

  return 0;

}
```

In the above code, we have created two variables 'i' and 'j' of type double and float respectively, and then we print the size of the expression by using **sizeof(i+j)** operator.

Output

```
size of (i+j) expression is : 8
```

const Pointer in C:

Constant Pointers:

A constant pointer in C cannot change the address of the variable to which it is pointing, i.e., the address will remain constant. Therefore, we can say that if a constant pointer is pointing to some variable, then it cannot point to any other variable.

Syntax of Constant Pointer

```
<type of pointer> *const <name of pointer>;
```

Declaration of a constant pointer is given below:

```
int *const ptr;
```

Let's understand the constant pointer through an example.

```
#include <stdio.h>

int main()
```

```
int a=1;

int b=2;

int *const ptr;

ptr=&a;

ptr=&b;

printf("Value of ptr is :%d",*ptr);

return 0;
```

Output:

```
pointers_to_array.c: In function 'main':
pointers_to_array.c:7:8: error: assignment of read-only variable 'ptr'
     ptr=&a;
        ^
pointers_to_array.c:8:8: error: assignment of read-only variable 'ptr'
     ptr=&b;
        ^
```

In the above output, we can see a error this is due we are changing the value of ptr from &a to &b which is not possible with a constant pointer.

Pointer to Constant:

A pointer to constant is a pointer through which the value of the variable that the pointer points cannot be changed. The address of these pointers can be changed, but the value of the variable that the pointer points cannot be changed.

Syntax:

```
const <type of pointer>* <name of pointer>
```

Declaration of a pointer to constant is given below:

```
const int* ptr;
```

Let's understand through an example.

Example:

```
#include <stdio.h>

int main()

{

    int a=100;

    int b=200;

    const int* ptr;

    ptr=&a;

    ptr=&b;

    *ptr = 300;

    printf("Value of ptr is :%u",ptr);

    return 0;

}
```

Output:

```
pointers_to_array.c: In function 'main':
pointers_to_array.c:9:10: error: assignment of read-only location '*ptr'
     *ptr = 300;
          ^
```

EXAPLINATION:

In above output we encounter an error this is due to we can change address of ptr but not the value of that variable which is pointed.

Constant Pointer to a Constant:

A constant pointer to a constant is a pointer, which is a combination of the above two pointers. It can neither change the address of the variable to which it is pointing nor it can change the value placed at this address.

Syntax:

```
const <type of pointer>* const <name of the pointer>;
```

Declaration for a constant pointer to a constant is given below:

```
const int* const ptr
```

Let's understand through an example.

```
#include <stdio.h>
int main()
{
    int a=10;
    int b=90;
    const int* const ptr=&a;
   *ptr=12;
    ptr=&b;
    printf("Value of ptr is :%d",*ptr);
    return 0;
}
```

Output:

```
pointers_to_array.c: In function 'main':
pointers_to_array.c:7:8: error: assignment of read-only location '*ptr'
    *ptr=12;
        ^
pointers_to_array.c:8:8: error: assignment of read-only variable 'ptr'
     ptr=&b;
        ^
```

Explanation:

Above error due to in constant pointer to constant we cannot cange the value of variable and cannot change the address of pointer.

void pointer in C:

Till then we know that we can only point pointer to same datatype which we declared.To overcome this problem we use void in place of data_type infront of pointer declaration,so that that pointer can point to any datatype such as int, float, char etc.

Syntax:

```
void *pointer_name;
```

→**Note:** Void pointer cannot be dereferenced .

Example:

```
#include <stdio.h>
int main()
{
    int a=10;
    void *ptr ;
    ptr = &a;
    printf("%d", *ptr);
    return 0;
}
```

Output:

```
tempCodeRunnerFile.c: In function 'main':
tempCodeRunnerFile.c:7:18: warning: dereferencing 'void *' pointer
     printf("%d", *ptr);
                  ^~~~
tempCodeRunnerFile.c:7:5: error: invalid use of void expression
     printf("%d", *ptr);
```

Explanation:

As we discussed in note void pointer cannot be referenced so it wil give compile time error.

To avoid this see the code below:

```
#include <stdio.h>
int main()
{
    int a=10;
    void *ptr ;
    ptr = &a;
    printf("%d", *(int*)ptr);//here we want to write *(int*) before ptr to
get value of ptr.
//And to get address of ptr we need to keep (int*) before ptr.
```

```
    return 0;
}
```

Output:
10

- **Arithmetic operation on void pointers**

We cannot apply the arithmetic operations on void pointers in C directly. We need to apply the proper typecasting so that we can perform the arithmetic operations on the void pointers.

Example:

```
#include <stdio.h>

int main()

{

    float arr[3] = {8.9, 2.8, 6.7};

    void *ptr;

    ptr = &arr;

    for (int i = 0 ; i < 3; ++i){

        printf("%f", *ptr);

        ptr = ptr + 1;

      }

    return 0;

}
```

Above code gives a compile time error due to improper typecasting.
To correct that we need to write as given beow.

```c
#include <stdio.h>
int main()
{
    float arr[3] = {8.9, 2.8, 6.7};
    void *ptr;
    ptr = &arr;
    for (int i = 0 ; i < 3; ++i){
        printf("%f\n", *((float*)ptr+i));


    }
    return 0;
}
```

Output:

```
8.900000
2.800000
6.700000
```

→**Usage of void pointers**:We use void pointers because of its reusability. Void pointers can store the object of any type, and we can retrieve the object of any type by using the indirection operator with proper typecasting.

C dereference pointer:

The dereference operator is also known as an indirection operator, which is represented by (*). When **indirection operator (*)** is used with the pointer variable, then it is known as **dereferencing a pointer.** When we dereference a pointer, then the value of the variable pointed by this pointer will be returned.

Example:

```c
#include <stdio.h>
int main()
{
    int a = 5;
    int *ptr;
    ptr = &a;
    printf("%d", *ptr);//here we use * operator to get the value stored in
that pointer.
    return 0;
}
```

Output:
5

Null Pointer:

Examples of Null Pointer

int *ptr=(int *)0;
float *ptr=(float *)0;
char *ptr=(char *)0;
double *ptr=(double *)0;
char *ptr='\0';
int *ptr=NULL;

→why do we use this?

```c
#include <stdio.h>
int main()
{
    int *ptr;
   printf("Address: %d", ptr); // printing the value of ptr.
   printf("Value: %d", *ptr); // dereferencing the illegal pointer
   return 0;
}
```

From above code see that we are not assigned any address to pointer so when we print address it will give a garbage value or leads to crash.To avoid this we use **Null pointer.**

```c
#include <stdio.h>
int main()
{
    int *ptr = NULL;
   printf("Address: %d", ptr); // printing the value of ptr.
   printf("Value: %d", *ptr); // dereferencing the illegal pointer
   return 0;
}
```

This is correct method.

When we use the malloc() function.

In the above code, we use the library function, i.e., **malloc()**. As we know, that malloc() function allocates the memory; if malloc() function is not able to allocate the memory, then it returns the **NULL** pointer. Therefore, it is necessary to add the condition which

will check whether the value of a pointer is null or not, if the value of a pointer is not null means that the **memory is allocated**.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *ptr;
    ptr=(int*)malloc(4*sizeof(int));
    if(ptr==NULL)
    {
        printf("Memory is not allocated");
    }
    else
    {
        printf("Memory is allocated");
    }
    return 0;
}
```

Output:

```
Memory is allocated
```

C Function Pointer:

We know that every function we create has some memory address as which it stores.
To get the address of that function we use **Function pointers.**

Declaration of a function pointer

Syntax:

```
return_type (*ptr_name)(type1, type2…);
```

For example:

```
int (*ip) (int);
```

In the above declaration, *ip is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

> Note: Declaration of a function is necessary before assigning the address of a function to the function pointer.

Calling a function through a function pointer:

```c
#include <stdio.h>

int sum(int a, int b);



int main(){

    int result;

    int a,b;

    int (*ptr)(int,int);



    printf("Enter the values of a and b:\n");

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



    ptr = sum;

    result = (*ptr)(a,b);

    printf("The sum a+b is: %d", result);



}

int sum(int a, int b){

    int c = a + b;
```

```
    return c;



}
```

Output:

Enter the values of a and b:

67

5

The sum a+b is: 72

Passing a function's address as an argument to other function:

Let's understand through an example.

```
#include <stdio.h>
void func1(void (*ptr)());
void func2();
int main()
{
    func1(func2);
     return 0;
}
void func1(void (*ptr)())
{
    printf("Function1 is called");
    (*ptr)();
}
void func2()
{
    printf("\nFunction2 is called");
}
```

Output:

```
Function1 is called
Function2 is called
```

Array of Function Pointers:

Let's understand through an example.

```
#include <stdio.h>

float add(float,int);

float sub(float,int);

float mul(float,int);

float div(float,int);

int main()

{

    float x;              // variable declaration.

    int y;

    float (*fp[4]) (float,int);        // function pointer declaration.

    fp[0]=add;              // assigning addresses to the elements of an
array of a function   pointer.

    fp[1]=sub;

    fp[2]=mul;

    fp[3]=div;

    printf("Enter the values of x and y :");

    scanf("%f %d",&x,&y);

  float r=(*fp[0]) (x,y);        // Calling add() function.

    printf("\nSum of two values is : %f",r);

     r=(*fp[1]) (x,y);             // Calling sub() function.

    printf("\nDifference of two values is : %f",r);
```

```
      r=(*fp[2]) (x,y);             // Calliung sub() function.

    printf("\nMultiplication of two values is : %f",r);

     r=(*fp[3]) (x,y);            // Calling div() function.

    printf("\nDivision of two values is : %f",r);

    return 0;

}



float add(float x,int y)

{

    float a=x+y;

    return a;

}

float sub(float x,int y)

{

    float a=x-y;

    return a;

}

float mul(float x,int y)

{

    float a=x*y;

    return a;

}

float div(float x,int y)
```

```
{

    float a=x/y;

    return a;

}
```

Output:

Enter the values of x and y :3

7

Sum of two values is : 10.000000

Difference of two values is : -4.000000

Multiplication of two values is : 21.000000

Division of two values is : 0.428571

Function pointer as argument in C:

Syntax:

```
(type)  (*pointer_name)(parameter);
```

Lets see a brief example:

```
#include <stdio.h>



void display(void (*p)(int));

void print_numbers(int num);
```

```c
void display(void (*p)(int))

{

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

    {

        p(i);

    }

}



void print_numbers(int num)

{

    printf("%d", num);

}



int main()

{

    void (*p)(int); // void function pointer declaration



    printf("The values are: ");

    p = print_numbers; // Assign the function print_numbers to the
function pointer p

    display(p);

    return 0;

}
```

Output:

```
The values are: 12345
```

Explaination:

Certainly! Let's break down the code step by step and explain each part:

Step 1: Function Prototypes

```c
#include <stdio.h>

void display(void (*p)(int));
void print_numbers(int num);
```

Here, function prototypes for the `display` and `print_numbers` functions are declared. This informs the compiler about the existence and signature of these functions.

Step 2: Function Definition for `display`

```c
void display(void (*p)(int))
{
    for (int i = 1; i <= 5; i++)
    {
        p(i);
```

```
    }
}
```

This is the definition of the `display` function. It takes a function pointer `p` as an argument. The function pointer `p` points to a function that takes an integer argument and returns `void`. In the `display` function, a loop is executed from `i = 1` to `i <= 5`. During each iteration, the function pointed to by `p` is called, passing the current value of `i` as an argument.

Step 3: Function Definition for `print_numbers`

```c
void print_numbers(int num)
{
    printf("%d", num);
}
```

This is the definition of the `print_numbers` function. It takes an integer argument `num` and uses `printf` to print the value of `num` as a decimal integer.

Step 4: Main Function

```c
int main()
{
```

```
    void (*p)(int); // void function pointer declaration


    printf("The values are: ");
    p = print_numbers; // Assign the function print_numbers to the function pointer p
    display(p);
    return 0;
}
```

This is the `main` function, the entry point of the program. It declares a function pointer `p` that points to a function taking an integer argument and returning `void`. The statement `p = print_numbers;` assigns the address of the `print_numbers` function to the function pointer `p`.

The program then prints the string "The values are: " using `printf`. Next, it calls the `display` function and passes the function pointer `p` as an argument. This causes the `display` function to execute the loop five times, calling the `print_numbers` function each time with a different value of `i`.

Finally, the program returns 0 to indicate successful program execution.

Overall, this code demonstrates the use of function pointers in the C programming language. The `display` function acts as a generic function that can call any function passed to it as an argument, and the `print_numbers` function is the specific function that is called by `display`.

Here in Function pointer as argument topic now we learn a new method called qsort() .

qsort():

Certainly! Here's an explanation of `qsort` in the C language:

`qsort` is a standard library function in C that is used to sort arrays. It stands for "quick sort," which is a well-known sorting algorithm. The `qsort` function provides a convenient way to sort elements in an array based on a user-defined comparison function.

The syntax of the `qsort` function is as follows:

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

Parameters:
- `base`: Pointer to the start of the array to be sorted.
- `nmemb`: Number of elements in the array.
- `size`: Size in bytes of each element in the array.
- `compar`: Pointer to the comparison function that defines the sorting order.

The comparison function:
- The comparison function takes two `const void*` arguments, which represent the elements being compared.
- It returns an integer value that indicates the relationship between the two elements.
  - A negative value if the first element should be placed before the second.
  - Zero if the elements are considered equal in terms of sorting order.
  - A positive value if the first element should be placed after the second.

The `qsort` function works by repeatedly partitioning the array into two subarrays based on the comparison function. It then recursively sorts the subarrays until the entire array is sorted. This algorithm has an average time complexity of O(n log n).

Note: When using `qsort`, it is essential to ensure that the comparison function provided follows the expected format and correctly handles the types of elements being sorted. Care must be taken to avoid potential issues such as undefined behavior or incorrect sorting results.

Feel free to copy the above explanation to your documentation.

Now we see a example:

```
#include <stdio.h>
```

```c
#include <stdlib.h>



int compare(const void *x_void, const void *y_void){

    int x = *(int*)x_void;

    int y = *(int*)y_void;

    return y - x;//here y - x for ascending order.

                 //if we keep x - y it's for descending order.

}



int main(){

    int arr[] = {1, 5, 6, 2, 3, 7, 8, 10, 4, 9};

    int len_arr = sizeof(arr)/sizeof(int);

    qsort(arr, len_arr, sizeof(int), compare);

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

        printf("arr[%d] = %d\n", i, arr[i]);

    }







    return 0;

}
```

Output:

```
arr[0] = 10

arr[1] = 9
```

```
arr[2] = 8

arr[3] = 7

arr[4] = 6

arr[5] = 5

arr[6] = 4

arr[7] = 3

arr[8] = 2

arr[9] = 1
```

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 in a string since it is the only way to identify where the string ends.

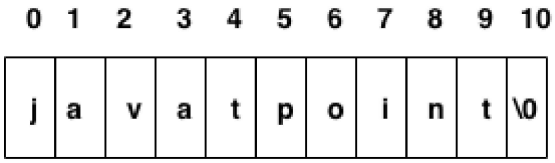
There are two ways to declare a string in c language.

1. By char array
2. By string literal

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

1. `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.



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.

The difference between char array string and string literal is we need to append '\0' at last in char array while in string literal it is done by the compiler and value cannot be reassigned in string literal where as in char array string.

Example:

```
#include <stdio.h>

#include <string.h>

int main(){

    char arr1[4] = {'k', 'd', 'n', '\0'};

    char arr2[4] = "kdn";

    printf("The char array value is:%s\n", arr1);

    printf("The string literal value is:%s\n", arr2);

    return 0;

}
```

Output:

```
The char array value is:kdn

The string literal value is:kdn
```

Pointers with strings:

Example:

```c
#include <stdio.h>
#include <string.h>

int main(){
    char p[] = "K.D.N.C";
    char *ptr;
    ptr = p;
    printf("%s\n", ptr);
    return 0;
}
```

Output:

```
K.D.N.C
```

C gets():

gets() function used to take string input and stores in char array and adds '\0' at the the end of array to make that as a string.It also space separated strings as input and stops when it triggers **enter.**

Declaration

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

Example:

```c
#include <stdio.h>
#include <string.h>

int main(){
    char str[100];
    printf("Enter the string you want: ");
    gets(str);
```

```
    printf("%s", str);
    return 0;
}
```

Output:

Enter the string you want: MAD_DRAX
MAD_DRAX

Note: 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() makes sure that not more than the maximum limit of characters are read.

fgets() in C:

Syntax:

```
char *fgets(char *str, int num, FILE *stream);
```

Where:

- **str**: Pointer to an array of chars where the string read is stored.
- **num**: Maximum number of characters to be read (including the final null-character).
- **stream**: Pointer to a FILE object that identifies the stream where characters are read from.

Example:

```
#include <stdio.h>
#include <string.h>

int main(){
    char str[20];
    printf("Enter the string you want: ");
    fgets(str, 20, stdin);
    printf("%s", str);
    return 0;
}
```

Output:

Enter the string you want: Hello MAD_DRAX i am a hacker
Hello MAD_DRAX i am *//here only 10 characters will be printed.*

puts() in C:

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[])
```

Example:

```
#include <stdio.h>
#include <string.h>

int main(){
    char str[20];
    printf("Enter your name: ");
    gets(str);
    printf("Your name is : ");
    puts(str);
    return 0;
}
```

Output:

Enter your name : K.D.N.C
Your name is : K.D.N.C

C String Functions:

There are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination, source)	copies the contents of source string to destination string.

3)	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase.

strlen():

Example:

```
#include <stdio.h>
#include <string.h>

int main(){
    char str[20];
    printf("Enter your name: ");
    gets(str);

    printf("The length of the string is : %d", strlen(str));
    return 0;
}
```

Output:

Enter your name: MAD_DRAX
The length of the string is : 8

strcpy():

Example:

```
#include <stdio.h>
#include <string.h>
```

```c
int main(){
    char str1[20];
    char str2[20];
    printf("Enter your name: ");
    gets(str1);
    strcpy(str2, str1);
    printf("The value that copied from str1 to str2 is : %s", str1);
    return 0;
}
```

Output:

Enter your name: MAD_DRAX
The value that copied from str1 to str2 is : MAD_DRAX

strcat():

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.
Here concatenation means combination of two strings.

Example:

```c
#include <stdio.h>
#include <string.h>

int main(){
    char str1[20] = {'H', 'e', 'l', 'l', 'o', '\0'};
    char str2[20] = {'D','R', 'A', 'X', '\0' };
    strcat(str1, str2);
    printf("The combined string is  : %s", str1);
    return 0;
}
```

Output:

The combined string is : HelloDRAX

strcmp():

It compares the two strings .If the two strings are same it returns 0.

Example:

```c
#include <stdio.h>
#include <string.h>

int main(){
```

```
    char str1[20];
    printf("Enter the string 1: ");
    gets(str1);

    char str2[20];
    printf("Enter the string 2: ");
    gets(str2);

    if (strcmp(str1, str2) == 0)
    {
        printf("str1 and str2 are equall");
    }

    else
    {
        printf("str1 and str2 are not equall");
    }
    return 0;
}
```

Output:

Enter the string 1: hello
Enter the string 2: hello
str1 and str2 are equall

Another try of this code.

Enter the string 1: hello
Enter the string 2: hell
str1 and str2 are not equall

strrev():

Example:

```
#include <stdio.h>
#include <string.h>

int main(){
    char str[20];
    printf("Enter your name: ");
    gets(str);
    strrev(str);
```

```
    printf("Your name is : ");
    puts(str);
    return 0;
}
```

Output:

Enter your name: k.d.n.c
Your name is : c.n.d.k

strlwr():

Example:

```
#include <stdio.h>
#include <string.h>

int main(){
    char str1[20];
    printf("Enter the string 1: ");
    gets(str1);

    printf("\nThe string after using strlwr is: %s", strlwr(str1));
    return 0;
}
```

Output:

Enter the string 1: Hello

The string after using strlwr is: hello

strupr():

Example:

```
#include <stdio.h>
#include <string.h>

int main(){
    char str1[20];
    printf("Enter the string 1: ");
    gets(str1);

    printf("\nThe string after using strupr is: %s", strupr(str1));
    return 0;
}
```

Output:
Enter the string 1: hello

The string after using strupr is: HELLO

strstr():

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.

Syntax:

```
char *strstr(const char *string, const char *match)
```

String strstr() parameters:

string: It represents the full string from where substring will be searched.

match: It represents the substring to be searched in the full string.

Example:

```
#include <stdio.h>

#include <string.h>



int main(){

    char str1[100];

    char *sub;

    printf("Enter the string 1: ");

    gets(str1);

    sub = strstr(str1, "k.d.n.c");

    printf("\nSubstring is: %s", sub);
```

```
    return 0;

}
```

Output:

Enter the string 1: My name is k.d.n.c and i am learning strstr in c from string methods

Substring is: k.d.n.c and i am learning strstr in c from string methods

File Handling in C

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

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

- *Creation of the new file*
- *Opening an existing file*
- *Reading from the file*
- *Writing to the file*
- *Deleting the file*

Functions for file handling:

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

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position

8	*fputw()*	*writes an integer to file*
9	*fgetw()*	*reads an integer from file*
10	*ftell()*	*returns current position*
11	*rewind()*	*sets the file pointer to the beginning of the file*

Opening File: fopen():

Syntax:

```
FILE* fopen(const char *file_name, const char *access_mode);
```

Parameters

file_name: name of the file when present in the same directory as the source file. Otherwise, full path.

access_mode: Specifies for what operation the file is being opened.

Return Value

If the file is opened successfully, returns a file pointer to it.

If the file is not opened, then returns NULL.

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

Mode	Description
r	opens a text file in read mode

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

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

```
#include<stdio.h>

void main( )

{

FILE *fp ;

char ch ;

fp = fopen("Sample_File.txt","r") ;

while ( 1 )

{

ch = fgetc ( fp ) ;

if ( ch == EOF )

break ;

printf("%c",ch) ;

}

fclose (fp ) ;

}
```

Output:

The text in sample_file.txt wiil appear here.

Closing File: fclose():

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

Syntax:

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

=>Thank's For Reading<-

Warning:- These information gathered and referred from many sites.

Ended 01 july 2023

Kommi Druthendra Nad Chowdary