

Explain loops available in C with example

- Loops are used to repeat execution of a block of code.
- During looping, a set of statements are executed until some condition for termination is encountered.

Generally, looping process would include the following four steps

- 1) Initialization of a counter
- 2) Test for a termination condition
- 3) Loop body statements
- 4) Increment the counter

while loop

- The simplest of all looping structure is while statement.
- The general format of the while statement is:

```
Initialization;  
while (test condition)  
{  
    body of the loop ;  
    increment or decrement;  
}
```

Test condition is evaluated and if the condition is true then the body of the loop is executed.

- After the execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again.
- This process is repeated till the test condition is true. When it becomes false, the control is transferred out of the loop.

On exit, the program continues with the statements immediately after the body of the loop.

- While loop is also known as entry control loop because first control-statement is executed and if it is true then only body of the loop will be executed.

Example: To print first 10 positive integer numbers

```
void main()
{
    int i;
    i = 1;                \\ initialization of i
    while(i <= 10)        \\ condition checking
    {
        printf("\t%d",i); \\ statement execution
        i++;              \\ increment of control variable
    }
}
```

```
int num,i;
clrscr();
printf("Enter integer number\n");
scanf("%d",&num);
printf("reverse of %d is=",num);
while(num!=0)
{
    i=num%10;
    printf("%d",i);
    num=num/10;
}
getch();
```

(2)check for palindrome.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int num,i;
    int temp,rev=0;
    clrscr();
    printf("Enter integer number\n");
```

```
scanf("%d",&num);
printf("reverse of %d is=",num);
temp=num;
while(num!=0)
{
    i=num%10;
    rev=rev*10+i;
    num=num/10;
}
printf("%d\n",rev);
if(temp==rev)
{
    printf("given no.is palindrome.");
}
else
{
    printf("given no. is not palindrome.");
}
```

```
getch();
```

(Q-3) WAP to generate Fibonacci series up to given number.

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

```
#include<conio.h>
```

```
void main()
```

```
{    int n1,n2,n3,n;
```

```
    int count;
```

```
    clrscr();
```

```
    n1=0;
```

```
    n2=1;
```

```
    printf("how many fibonacci no?\n");
```

```
    scanf("%d",&n);
```

```
    if(n>2)
```

```
    {
```

```
        printf("Fibonacci no.are\n");
```

```
        printf("%3d%3d",n1,n2);
```

```
        count=2;
```

```
        while(count<n)
```

```
        {
```

```
            n3=n1+n2;
```

```
            printf("%3d",n3);
```

```
            n1=n2;
```

```
            n2=n3;
```

```
            count++;
```

```
        }
```

```
    }
```

```
    getch();
```

```
}
```


(Q-1) WAP (1) to reverse given number

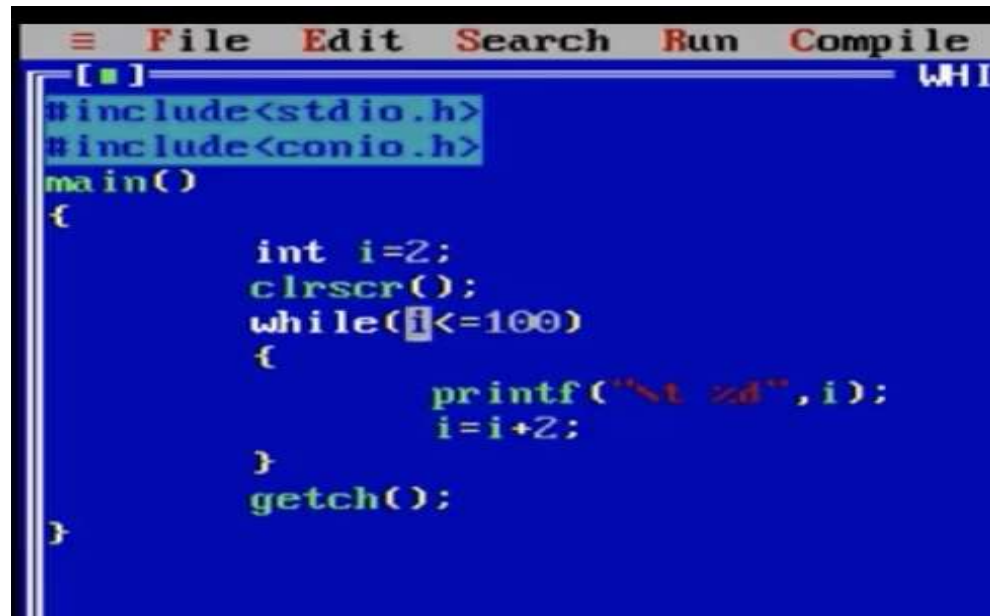
```
#include<stdio.h>
#include<conio.h>
void main()
{
    int num,i;
    clrscr();
    printf("Enter integer number\n");
    scanf("%d",&num);
    printf("reverse of %d is=",num);
    while(num!=0)
    {
        i=num%10;
        printf("%d",i);
        num=num/10;
    }
    getch();
}
```

table

```
01. #include <stdio.h>
02. int main()
03. {
04.     int n, i;
05.
06.     printf("Enter a Number ");
07.     scanf("%d",&n);
08.     i=1;
09.     while(i<=10){
10.
11.         printf("%d * %d = %d \n", n, i, n*i);
12.         ++i;
13.     }
14.
15.     getch();
16.
17. }
```

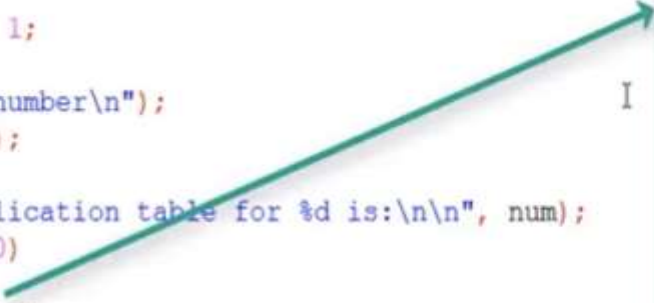
```
main.c x
1  #include<stdio.h>
2
3  int main()          I
4  {
5      int num, count = 2, flag = 1;
6
7      printf("Enter a number\n");
8      scanf("%d", &num);
9
10     while(count < num)
11     {
12         if(num%count == 0)
13         {
14             flag = 0;
15             break;
16         }
17         count++;
18     }
19
20     if(flag) printf("%d is prime number\n", num);
21     else    printf("%d is not prime number\n", num);
22
```

Even/odd



```
File Edit Search Run Compile
[ ]
#include<stdio.h>
#include<conio.h>
main()
{
    int i=2;
    clrscr();
    while(i<=100)
    {
        printf("%d ", i);
        i=i+2;
    }
    getch();
}
```

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int num, count = 1;
6
7      printf("Enter a number\n");
8      scanf("%d", &num);
9
10     printf("\nMultiplication table for %d is:\n\n", num);
11     while(count <= 10)
12     {
13         printf("%d x ", num);
14         count++;
15     }
16     return 0;
17 }
18
```



2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10

```
#include<stdio.h>
int main()
{
    int num,i=1,c=0;
    printf("/*To Check Number Prime or
Not*/\n\nEnter Number : ");
    scanf("%d",&num);
    while(i<=num)
    {
        if(num%i==0)
            c++;
            i++;
    }
    if(c==2)
        printf("\n%d is Prime Number",num);
    else
        printf("\n%d is Not Prime
Number",num);
    return 0;
}
```



Code:

```
#include<stdio.h>
#include<conio.h>
main( )
{   int i,j;
    clrscr( );
    i=1;
    while(i<=5)
    {
        printf("  ");
        j=1;
        while(j<=i)
        {
            printf(" * ");
            j++;
        }
        printf("\n");
        i++;
    }
    getch( );
}
```

O/P:

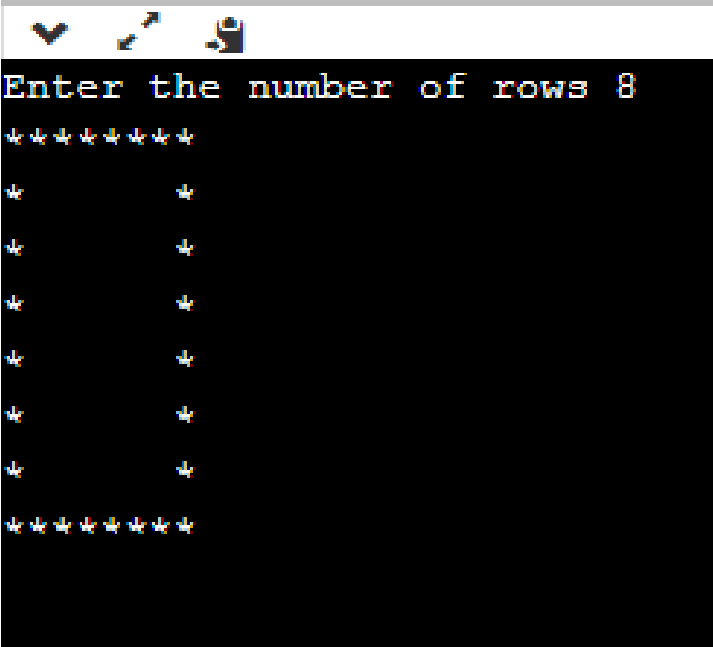
Code:

```
*
*  *
*  *  *
*  *  *  *
*  *  *  *  *
```

```

{
    int n;
    printf("Enter the number of rows");
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=n;j++)
        {
            if(i==1 || i==n || j==1 || j==n)
            {
                printf("*");
            }
            else
            {
                printf(" ");
            }
        }
        printf("\n");
    }
}

```



```

Enter the number of rows 8
*****
*       *
*       *
*       *
*       *
*       *
*       *
*****

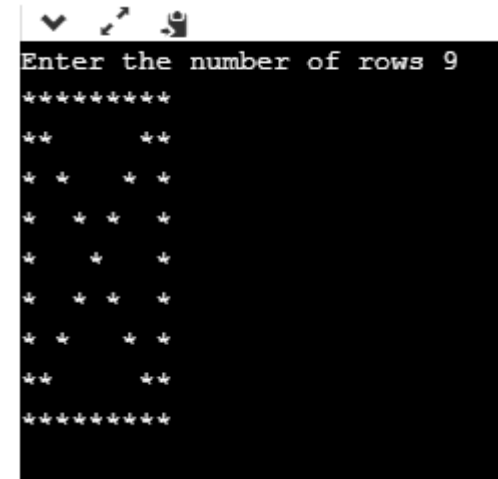
```



```

printf("Enter the number of rows");
scanf("%d",&n);
for(int i=1;i<=n;i++)
{
    for(int j=1;j<=n;j++)
    {
        if(i==1 || i==n || j==1 || j==n-i+1 || i==j || j==n)
        {
            printf("*");
        }
        else
        {
            printf(" ");
        }
    }
    printf("\n");
}

```



```

Enter the number of rows 9
*****
**      **
* *    * *
*  *  *  *
*   *   *
*  *  *  *
* *    * *
**      **
*****

```

various string handling operations

Example: `char s []="Their", s2[]="There";`

Function	Meaning
strlen(s1)	Returns length of the string. <code>l = strlen(s1);</code> it returns 5
strcmp(s1,s2)	Compares two strings. It returns negative value if <code>s1 < s2</code> , positive if <code>s1 > s2</code> and zero if <code>s1 = s2</code> .

strcpy(s1,s2)	<p>Copies 2nd string to 1st string.</p> <p>strcpy(s1,s2) copies the string s2 in to string s1 so s1 is now "There".</p> <p>s2 remains unchanged.</p>
strcat(s1,s2)	<p>Appends 2nd string at the end of 1st string.</p> <p>strcat(s1,s2); a copy of string s2 is appended at the end of string s1. Now s1 becomes "TheirThere"</p>
strchr(s1,c)	<p>Returns a pointer to the first occurrence of a given character in the string s1.</p> <p>printf("%s",strchr(s1,'i'));</p> <p>Output : ir</p>
strstr(s1,s2)	<p>Returns a pointer to the first occurrence of a given string s2 in string s1.</p> <p>printf("%s",strstr(s1,"he"));</p> <p>Output : heir</p>

strrev(s1)	Reverses given string. strrev(s1); makes string s1 to "riehT"
strlwr(s1)	Converts string s1 to lower case. printf("%s", strlwr(s1)); Output : their
strupr(s1)	Converts string s1 to upper case. printf("%s", strupr(s1)); Output : THEIR
strncpy(s1,s2,n)	Copies first n character of string s2 to string s1 s1=""; s2="There"; strncpy(s1,s2,2); printf("%s",s1); Output : Th

strncat(s1,s2,n)	<p>Appends first n character of string s2 at the end of string s1.</p> <pre>strncat(s1,s2,2);</pre> <pre>printf("%s", s1);</pre> <p>Output : TheirTh</p>
strncmp(s1,s2,n)	<p>Compares first n character of string s1 and s2 and returns similar result as strcmp() function.</p> <pre>printf("%d", strcmp(s1,s2,3));</pre> <p>Output : 0</p>
strrchr(s1,c)	<p>Returns the last occurrence of a given character in a string s1.</p> <pre>printf("%s",strrchr(s2,'e'));</pre> <p>Output : ere</p>

(Q-1) WAP to find (a) the length of a string using strlen (),

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

```
#include<conio.h>
```

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

```
void main()
```

```
{
```

```
char s1[30];
```

```
int n;
```

```
clrscr();
```

```
printf("entre string=");
```

```
gets(s1);
```

```
n=strlen(s1);
```

```
printf(" string length=%d",n);
```

```
getch();
```

```
}
```

(b) to reverse a string using strrev ()

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

```
#include<conio.h>
```

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

```
void main()
```

```
{
```

```
char s1[30];
```

```
clrscr();
```

```
printf("entre string=");
```

```
gets(s1);
```

```
strrev(s1);
```

```
printf("reverse string is=%s",s1);
```

```
getch();
```

(c) Copy one string into another string using strcpy (),

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char s1[30];
char s2[30];
clrscr();
printf("entre string 1=");
gets(s1);
printf("entre string 2=");
gets(s2);
strcpy(s2,s1);
printf("string is=%s",s2);
getch();
}
```


(e) convert a string into lower case using strlwr (),

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

```
#include<conio.h>
```

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

```
void main()
```

```
{
```

```
char s1[30],lwr[30];
```

```
clrscr();
```

```
printf("entre string=");
```

```
gets(s1);
```

```
strlwr(s1);
```

```
printf("lwer string is=%s",s1);
```

```
getch();
```

```
}
```

(d) concatenate two string using strcat ()

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char s1[30];
char s2[30];
clrscr();
printf("entre string 1=");
gets(s1);
printf("entre string 2=");
gets(s2);
strcat(s1,s2);
printf("string is=%s",s1);
getch();
}
```

(f) convert a string into uppercase usingstrupr ().

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

```
#include<conio.h>
```

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

```
void main()
```

```
{
```

```
char s1[30],upr[30];
```

```
clrscr();
```

```
printf("Enter string=");
```

```
gets(s1);
```

```
strupr(s1);
```

```
printf("upr is=%s",s1);
```

```
getch();
```

```
}
```

Functions

- A function is a block of code that performs a specific task.
- The functions which are created by programmer are called user-defined functions.
- The functions which are in-built in compiler are known as system functions.
- The functions which are implemented in header libraries are known as library functions.

- It has a unique name and it is reusable i.e. it can be called from any part of a program.
- Parameter or argument passing to function is optional.
- It is optional to return a value to the calling program. Function which is not returning any value from function, their return type is void.

While using function, three things are important

1) Function Declaration

2) Function Definition

3) Function Call

1. Function Declaration

- Like variables, all the functions must be declared before they are used.

- The function declaration is also known as function prototype or function signature. It consists of four parts,

- 1) Function type (return type).

- 2) Function name.

- 3) Parameter list.

- 4) Terminating semicolon

Syntax: <return type> FunctionName

Example: int sum(int , int);

(Argument1, Argument2, Argument3.....);

- In this example, function return type is int,
 - name of function is sum,
- 2 parameters are passed to function and both are integer.

2. Function Definition

- Function Definition is also called function implementation.
- It has mainly two parts.
- **Function header** : It is same as function declaration but with argument name.
- **Function body** : It is actual logic or coding of the function

3. Function call

- Function is invoked from main function or other function that is known as function call.
- Function can be called by simply using a function name followed by a list of actual argument enclosed in parentheses.

Syntax or general structure of a Function

<return type> FunctionName (Argument1,
Argument2, Argument3.....)

{

Statement1;

Statement2;

Statement3;

}

An example of function

```
#include<stdio.h>

int sum(int, int);           \\ Function Declaration or Signature

void main()
{
    int a, b, ans;
    scanf("%d%d", &a, &b);
    ans = sum(a, b);         \\ Function Calling
    printf("Answer = %d", ans);
}

int sum (int x, int y)       \\ Function Definition
{
    int result;
    result = x + y;
    return (result);
}
```

Explain different categories of functions.

Functions can be classified in one of the following category based on whether arguments are present or not, whether a value is returned or not.

- | | |
|--|---------------------------------------|
| 1. Functions with no arguments and no return value | <code>\\ void printline(void)</code> |
| 2. Functions with no arguments and return a value | <code>\\ int printline(void)</code> |
| 3. Functions with arguments and no return value | <code>\\ void printline(int a)</code> |
| 4. Functions with arguments and one return value | <code>\\ int printline(int a)</code> |
| 5. Functions that return multiple values using pointer | <code>\\ void printline(int a)</code> |

1. Function with no argument and no return value.

- When a function has no argument, it does not receive any data from calling function.
- When it does not return a value, the calling function does not receive any data from the called function.
- In fact there is no data transfer between the calling function and called function.

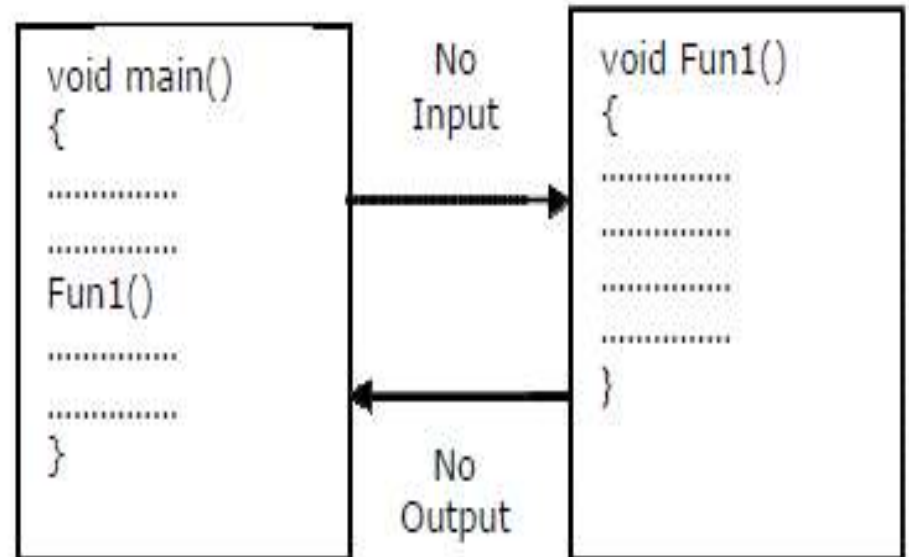
Example:

```
#include<stdio.h>

void printline(void);    // No argument - No return value

void main()
{
    clrscr();
    printline();
    printf("\n GTU \n");
}

void printline(void)
{
    int i;
    for(i=0;i<10;i++)
    {
        printf("-");
    }
}
```



2. Function with no arguments and return a value

- When a function has no argument, it does not receive data from calling function.
- When a function has return value, the calling function receives one data from the called function.

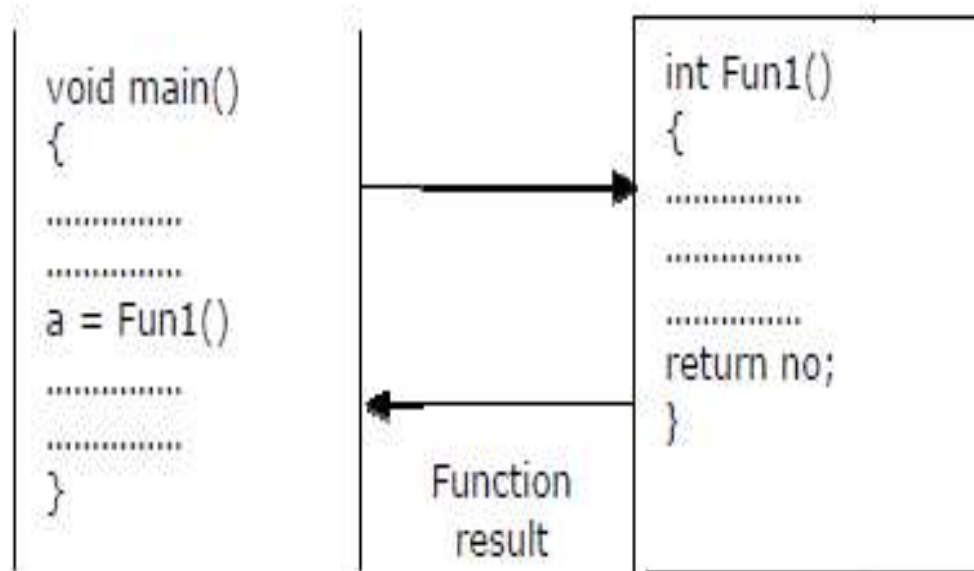
Example:

```
#include<stdio.h>

int get_number(void);           // No argument

void main()
{
    int m;
    m=get_number();
    printf("%d",m);
}

int get_number(void)
{
    int number;
    printf("enter number:");
    scanf("%d",&number);
    return number;              // Return value
}
```



3. Function with arguments and no return values

- When a function has argument, it receives data from calling function.
- When it does not return a value, the calling function does not receive any data from the called function.

```

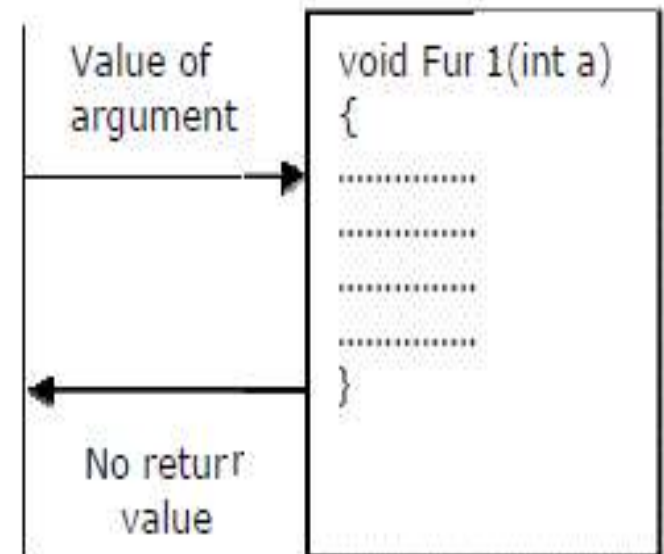
#include<stdio.h>
#include<conio.h>
void sum(int,int);          // Argument
void main()
{
    int no1,no2;
    printf("enter no1,no2:");
    scanf("%d%d",&no1,&no2);
    sum(no1,no2);
}
void sum(int no1,int no2)
{
    if(no1>no2)
        printf("\n no1 is gretest");
    else
        printf("\n no2 is gretest");
} // No return value

```

```

void main()
{
    .....
    Fun1(a)
    .....
}

```



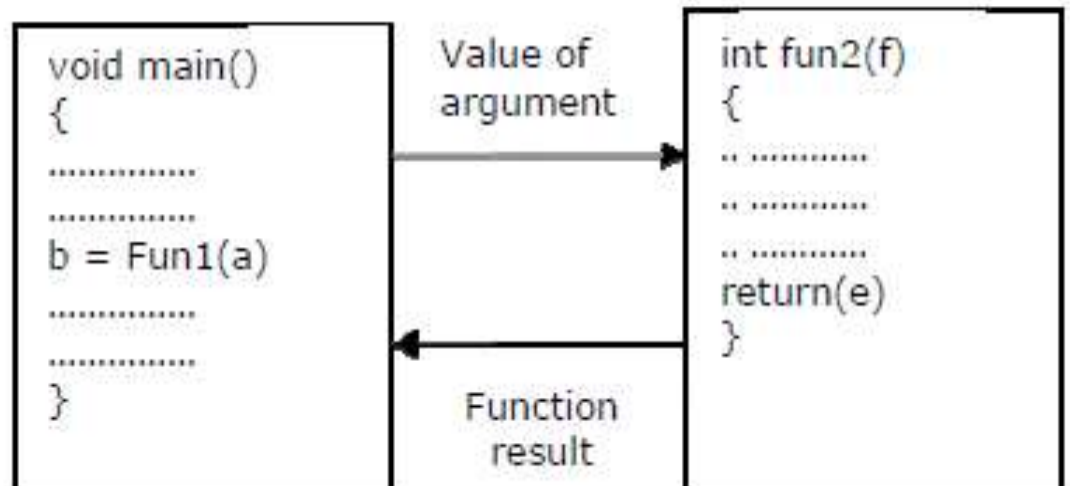
4. Function with arguments and one return value

- When a function has argument, it receives data from calling function.
- When a function has return value, the calling function receives any data from the called function.

```

#include<stdio.h>
#include<conio.h>
int sum(int);      // Argument
void main()
{
    int no,x;
    clrscr();
    printf("enter no:");
    scanf("%d",&no);
    x=sum(no);
    printf("sum=%d",x);
    getch();
}
int sum(int no)
{
    int add=0,i;
    while(no>0)
    {
        i=no%10;

```



```
        add=add+i;  
        no=no/10;  
    }  
    return add;                // Return value  
}
```

5. Function that returns a multiple value

- Function can return either one value or zero value. It cannot return more than one value
- To receive more than one value from function, we have to use pointer.
- So function should be called with reference not with value

Example:

```
#include<stdio.h> void
mathoperation(int void x, int y, int *s, int *d);
main()
{
    int x=20,y=10,s,d;
    mathoperation(x,y,&s,&d);
    printf("s=%d \nd=%d", s,d);
}
void mathoperation(int a, int b, int *sum, int *diff)
{
    *sum = a + b;
    *diff = a - b;
}
```


Recursive function

- Recursive function is a function that calls itself.
- If a function calls itself then it is known as recursion.
- Recursion is thus the process of defining something in terms of itself.
- Suppose we want to calculate the factorial of a given number then in terms of recursion we can write it as

$$n! = n * (n-1)!$$

First we have to find $(n-1)!$ and then multiply it by n .

the $(n-1)!$ is computed as $(n-1)! = (n-1)*(n-2)!$

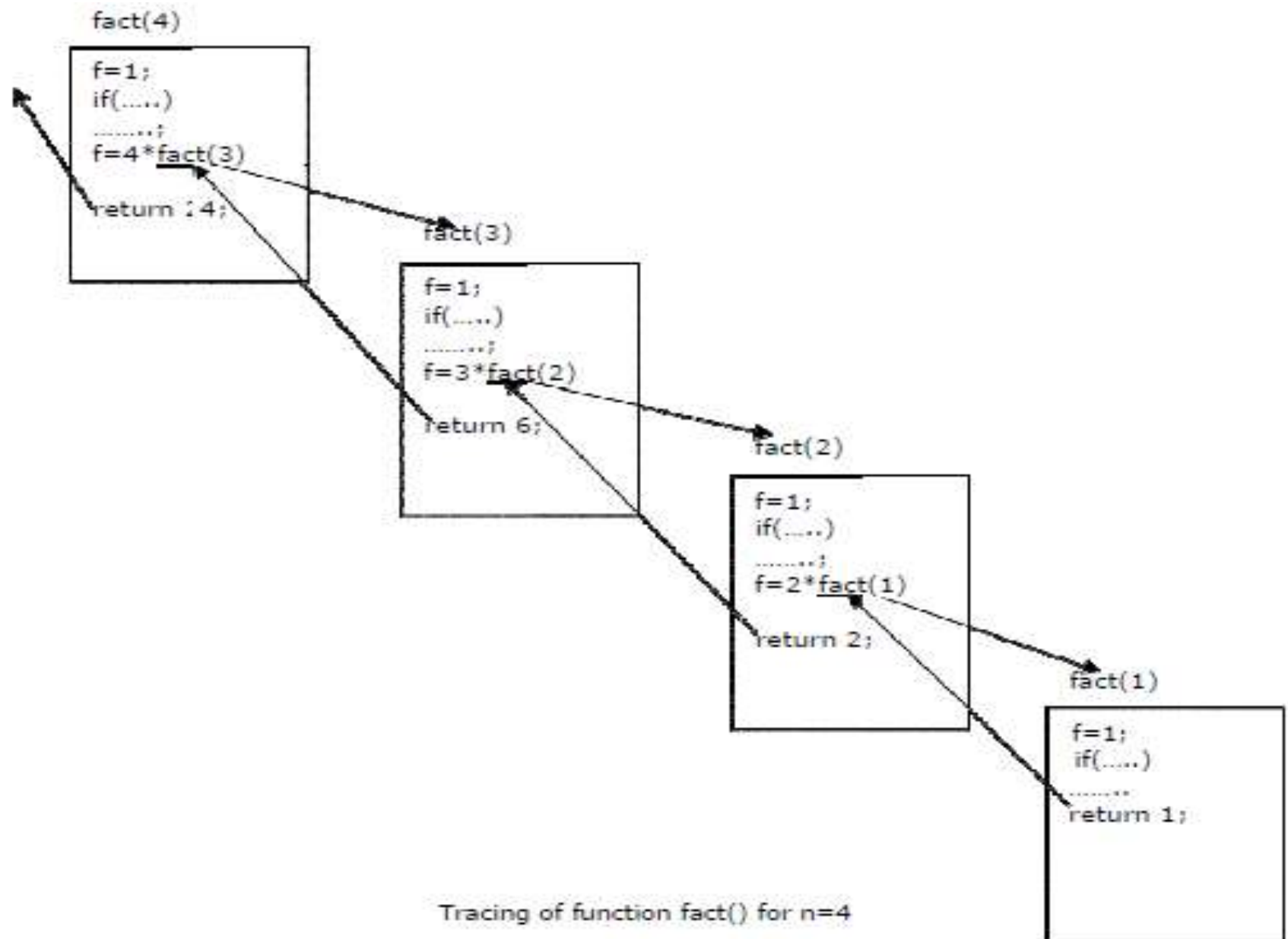
This process ends when finally we need to calculate $1!$ which is 1.

$$\text{Ex: } 4! = 4 * 3!$$

$$= 4 * 3 * 2!$$

$$= 4 * 3 * 2 * 1!$$

$$= 4 * 3 * 2 * 1$$



Tracing of function `fact()` for $n=4$

Example: Find factorial of a given number using recursion.

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main()
{
    int f,n;
    printf("enter number:");
    scanf("%d",&n);
    f=fact(n);
    printf("\n factorial=%d",f);
}
int fact(int n)
{
    int f=1;
    if(n==1)
    {
        return 1;
    }
    else
    {
        f=n*fact(n-1);
        return f;
    }
}
```

Advantages

- Easy solution for recursively defined solution.
- Complex programs can be easily written with less code.

Disadvantages

- Recursive code is difficult to understand and debug.
- Terminating condition is must; otherwise it will go in an infinite loop.
- Execution speed decreases because of function call and return activity many times.

call by value (pass by value)

The parameters can be passed in two ways during function calling,

- Call by value
- Call by reference

Call by value

- In call by value, the values of actual parameters are copied to their corresponding formal parameters.
 - So the original values of the variables of calling function remain unchanged.
- Even if a function tries to change the value of passed parameter, those changes will occur in formal parameter, not in actual parameter.

```
#include<stdio.h>

void swap(int, int);

void main()
{
    int x, y;
    printf("Enter the value of X & Y:");
    scanf("%d%d", &x, &y);
    swap(x, y);
    printf("\n Values inside the main function");
    printf("\n x=%d, y=%d", x, y);
    getch();
}
```

```
void swap(int x,int y)
{
    int temp;
    temp=x;
    x=y;
    y=temp;
    printf("\n Values inside the swap function");
    printf("\n x=%d y=%d", x, y);
}
```

Enter the value of X & Y: 3 5

Values inside the swap function

X=5 y=3

Values inside the main function

X=3 y=5

Call by Reference

- In call by reference, the address of the actual parameters is passed as an argument to the called function.
- So the original content of the calling function can be changed.
- Call by reference is used whenever we want to change the value of local variables through function.

Example:

```
#include<stdio.h>

void swap(int *, int *);

void main()
{
    int x,y;
    printf("Enter the value of X & Y:");
    scanf("%d%d", &x, &y);
    swap(&x, &y);
    printf("\n Value inside the main function");
    printf("\n x=%d y=%d", x, y);
}
```

```
void swap(int *x, int *y)
{
    int temp;
    temp=*x;
    *x=*y;
    *y=temp;
    printf("\n Value inside the swap function");
    printf("\n x=%d y=%d", x, y);
}
```

Output:

Enter the value of X & Y: 3 5

Value inside the swap function

X=5 y=3

Value inside the main function

X=5 y=3

actual argument and formal argument with example.

- Arguments passed to the function during function calling are called actual arguments or parameters.
- Arguments received in the definition of a function are called formal arguments or parameters.

scope, lifetime and visibility of variable?

Scope

The scope of variable can be defined as that a part of a program where the particular variable is accessible.

In what part of the program the variable is accessible is depends on where the variable is declared.

Local variables which are declared inside the body of function cannot be accessed outside the body of function.

- Global variables which are declared outside any function definition can be accessible by all the function in a body.

Lifetime

Lifetime is a time limit during the program execution until which a variable exist in a memory.

In C language, a variable can have automatic, static or dynamic lifetime.

Automatic – A variable with automatic lifetime are created. Every time, their declaration is encountered and destroyed. Also, their blocks are exited.

Static – A variable is created when the declaration is executed for the first time. It is destroyed when the execution stops/terminates.

Dynamic – The variables memory is allocated and deallocated through memory management functions.

Visibility

Visibility is the ability of the program to access a variable from the memory.

If variable is redeclared within its scope of variable, the variable losses the visibility in the scope of variable which is redeclared.

What is pointer? How to declare and initialize it?

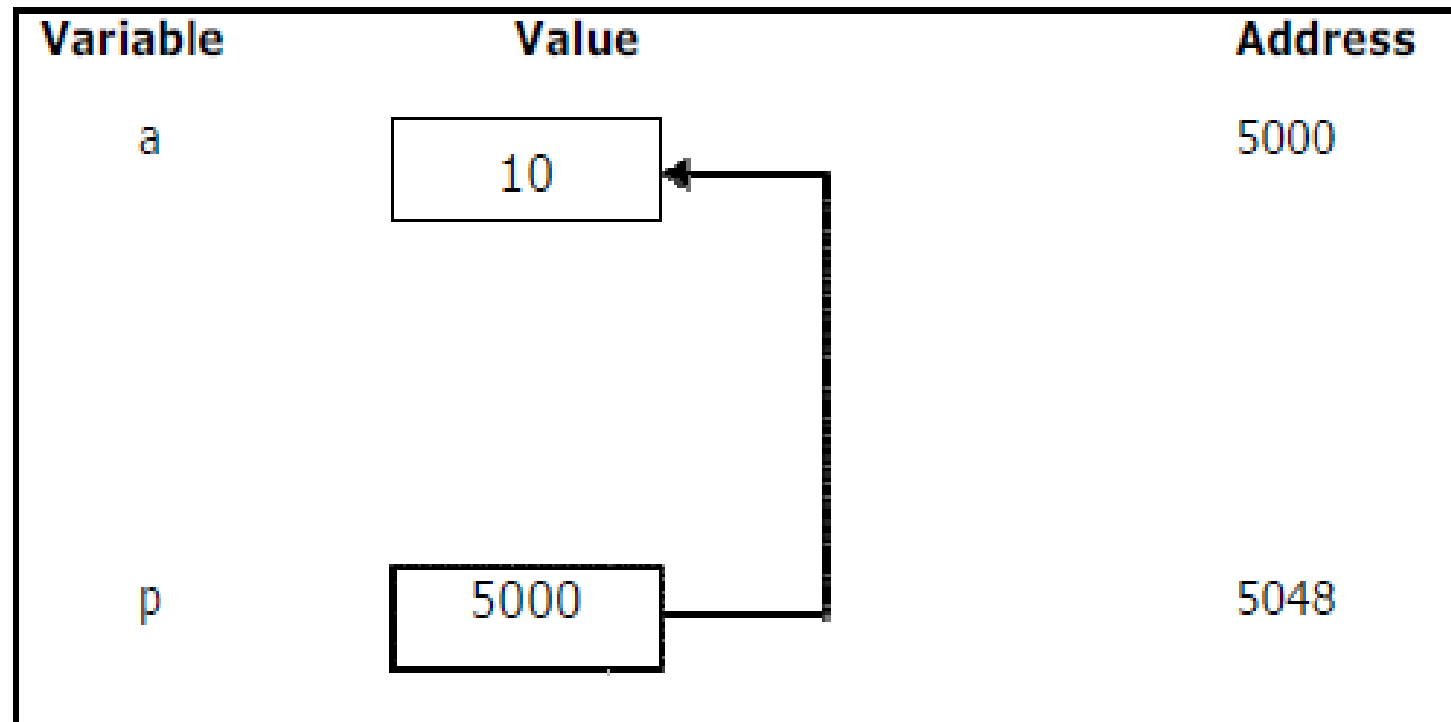
A pointer is a variable that contains address or location of another variable.

- **Pointer is a derived data type in C.**
- Pointers contain memory address as their values, so they can also be used to access and manipulate data stored in memory.

```
void main()
{
int a=10, *p;
p = &a;
\\ Assign memory address of a to pointer variable p
printf(“%d %d %d”, a, *p, p);
}
```

Output: 10 10 5000

- p is integer pointer variable
- & is address of or referencing operator which returns memory address of variable
- * is indirection or dereferencing operator which returns value stored at that memory address
- & operator is the inverse of * operator (x = a is same as x = *(&a))



Declaration of pointer,

Syntax:

Example

:

```
data_type *pt_name;
```

```
int *p, float *q, char *c;
```

- 1) The asterisk (*) tells that the variable pt_name is a pointer variable
- 2) pt_name needs a memory location to store address of another variable
- 3) pt_name points to a variable of type data_type

Initialization of the pointer,

`int a=5, x, *p; // Declares pointer variable p and
regular variable a and x`

`p = &a`

`x = *p;`

`// Initializes p with address of a`

`// p contains address of a and *p gives value stored
at that address.`

How pointer is different from array?

Array	Pointer
Array is a constant pointer.	Pointer variable can be changed.
It refers directly to the elements.	It refers address of the variable.
Memory allocation is in sequence.	Memory allocation is random.
Allocates the memory space which cannot resize or reassigned.	Allocated memory size can be resized.
It is a group of elements.	It is not a group of elements. It is single variable.

Discuss relationship between array and pointer.

```
int a[10], *p;
```

- Array name is constant pointer so a is constant pointer and it always points to the first element of an array.
- $a[0]$ is same as $*(a+0)$, $a[2]$ is same as $*(a+2)$, $a[i]$ is same as $*(a+i)$
- So every program written using array can always be written using pointer.

a:	a[0]
	a[1]
	.
	.
	.
	.
	a[i]
	.
	.
	.
	a[9]

a:	*(a+0)	2000
a+1:	*(a+1)	2002
	.	
	.	
	.	
	.	
a+i:	*(a+i)	2000+i*2
	.	
	.	
	.	
a+9:	*(a+9)	2018

Explain Array of Pointers

As we have an array of char, int, float etc..., same way we can have an array of pointers, individual elements of an array will store the address values. So, array of pointers is a collection of pointers of same type known by single name.

Syntax :

```
data_type *name[size];
```

Example :

```
int *ptr[5]; \\Declares an array of integer pointer of  
size 5
```

```
int mat[5][3]; \\Declares a two dimensional array  
of 5 by 2
```

Now, the array of pointers ptr can be used to point to different rows of matrix as follow

```
for(i=0;i<5;i++)  
{  
    ptr[i]=&mat[i][0]; \\ Can be re-written as  
    ptr[i]=mat[i];  
}
```

Swap value of two variables using pointer. OR

Swap value of two variables using call by reference

```
#include<stdio.h>
void swap(int *,int *);
void main()
{
    int a,b;
    printf("Enter two numbers:");
    scanf("%d%d", &a, &b);
    swap(&a, &b);
    printf("a=%d b=%d", a, b);
    getch();
}
void swap(int *a, int *b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}
```


calculate sum of 10 elements of an array using pointers

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int *p, a[10], sum=0, i;
    p=a;
    printf("Enter elements:");
    for(i=0; i<10; i++)
    {
        scanf("%d", &a[i]);
    }
    for(i=0; i<10; i++)
    {
        sum=sum+*p;
        p++;
    }
    printf("sum=%d", sum);
    getch();
}
```

Write a C program to print address of variable using pointer.

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

```
int main(void)
{
int i=15;
int *p;
p=&i;
printf("\n Address of Variable i = %d",p);
return 0;
}
```

Write a C program to print the address of character and the character of string using the pointer.

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

```
int main(void)
{
    char str[50];
    char *ch;
    printf("\n Enter String : ");
    scanf("%s",&str);
    ch=&str[0];
    while(*ch!='\0')
    {
        printf("\n Position : %u Character : %c",ch,*ch);
        ch++;
    }
    return 0;
}
```

Write C program to access elements using pointer.

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

```
int main(void)
```

```
{
```

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

```
int *p,i=0;
```

```
p=&a[0];
```

```
while(i<10)
```

```
{
```

```
printf("\n Position : %d Value : %d",i+1,*(p+i));
```

```
i++;
```

```
}
```

```
return 0;
```

```
}
```

What is structure

- Structure is a collection of logically related data items of different data types grouped together under a single name.
- Structure is a user defined data type.
- Structure helps to organize complex data in a more meaningful way.

Syntax of Structure:

```
struct structure_name  
{  
    data_type member1;  
    data_type member2;  
    .....  
};
```

- struct is a keyword.
- structure_name is a tag name of a structure.
- member1, member2 are members of structure.

Example:

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

struct book
{
    ch r title[100];
    ch r author[50];
    int pages;
    float price;
};
```

```
void main()
{
    struct book book1;
    printf("enter title, author name, pages and price of
           book");
    scanf("%s",book1.title);
    scanf("%s", book1.author);
    scanf("%d",&book1.pages);
    scanf("%f",&book1.price);
    printf("\n detail of the book");
    printf("%s",book1.title);
    printf("%s",book1.author);
```



```
printf("%d",book1.pages);  
printf("%f",book1.price);  
    getch();  
}
```

- book is structure whose members are title, author, pages and price.
- book1 is a structure variable.

Declaration of structure:

A structure variable declaration is similar to the declaration of variables of any other data type. It includes

the following elements:

- 1) The keyword struct
- 2) The structure tag name
- 3) List of variable names separated by commas
- 4) A terminating semicolon

For example:

```
struct book
{
    ch r title[100];
    ch r author[50];
    int pages;
    float price;
} book1;
struct book book2;
```

We can declare structure variable in two ways

- 1) Just after the structure body like book1
- 2) With struct keyword and structure tag name like
book2

Accessing structure members:

The following syntax is used to access the member of structure.

structure_variable.member_name

- structure_variable is a variable of structure and member_name is the name of variable which is a member of a structure.

- The “.”(dot) operator or ‘period operator’ connects the member name to structure name.

- for ex:

book1.price represents price of book1

We can assign values to the member of the structure variable book1 as below,

```
strcpy(book1.title,"ANSI C");  
strcpy(book1.author,"Balagurusamy");  
book1.pages=250;  
book1.price=120.50;
```

We can also use scanf function to assign value through a keyboard.

```
scanf("%s",book1.title);  
scanf("%d",&book1.pages);
```

Write a C program to read structure elements from keyboard.

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

```
struct book
```

```
{
```

```
int id;
```

```
char name[20];
```

```
float price;
```

```
};
```

```
int main(void)
```

```
{
```

```
struct book b1;
```

```
printf("\n Enter Book Id : ");
```

```
scanf("%d",&b1.id);
```

```
fflush(stdin);
```

```
printf("\n Enter Book Name : ");
```

```
scanf("%[^\n]s",b1.name);
```

```
printf("\n Enter Book Price : ");
```

```
scanf("%f",&b1.price);
```

```
printf("\nBook Id   = %d",b1.id);
```

```
printf("\nBook Name = %s",b1.name);
```

```
printf("\nBook Price = %.2f",b1.price);
```

```
return 0;
```

```
}
```

Define a structure type struct personal that would contain person name, date of joining and salary using this structure to read this information of 5 people and print the same on screen.

*/

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

```
struct person
```

```
{  
    char name[20];  
    char doj[10];  
    float salary;  
}p[5];
```

```
int main(void)
```

```
{  
    int i=0;
```

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

```
    {  
        printf("\n Enter Person Name : ");  
        scanf("%s",p[i].name);  
        printf("\n Enter Person Date of Joining (dd-mm-yyyy) : ");  
        scanf("%s",p[i].doj);  
        printf("\n Enter Person Salary : ");  
        scanf("%f",&p[i].salary);  
    }
```

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

```
    {  
        printf("\n Person %d Detail",i+1);  
        printf("\n Name   = %s",p[i].name);  
        printf("\n DOJ    = %s",p[i].doj);  
        printf("\n Salary = %.2f",p[i].salary);  
    }  
    return 0;  
}
```

Define structure data type called time_struct containing three member's integer hour, integer minute and integer second. Develop a program that would assign values to the individual number and display the time in the following format: 16: 40:51
*/

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

```
struct time_struct  
{  
int hour;  
int minute;  
int second;  
}t;
```

```
int main(void)  
{  
printf("\n Enter Hour : ");  
scanf("%d",&t.hour);  
printf("\n Enter Minute: ");  
scanf("%d",&t.minute);  
printf("\n Enter Second : ");  
scanf("%d",&t.second);
```

```
printf("\n Time %d:%d:%d",t.hour%24,t.minute%60,t.second%60);
```

```
return 0;  
}
```


Define a structure called cricket that will describe the following information:

Player name

Team name

Batting average

Using cricket, declare an array player with 50 elements and write a C program to read the information about all the 50 players and print team wise list containing names of players with their batting average.

*/

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

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

```
struct cricket
```

```
{
```

```
char player_name[20];
```

```
char team_name[20];
```

```
float batting_avg;
```

```
}p[50],t;
```

```
int main(void)
```

```
{
```

```
int i=0,j=0,n=50;
```

```
for(i=0;i<n;i++)
```

```
{
```

```
printf("\n Enter Player Name : ");
```

```
scanf("%s",p[i].player_name);
```

```
printf("\n Enter Team Name : ");
```

```
scanf("%s",p[i].team_name);
```

```
printf("\n Enter Batting Average : ");
```

```
scanf("%f",&p[i].batting_avg);
```

```
}
```

```

//Sorting of Data based on Team
for(i=0;i<n-1;i++)
{
for(j=i;j<n;j++)
{
if(strcmp(p[i].team_name,p[j].team_name)>0)
{
t=p[i];
p[i]=p[j];
p[j]=t;
}
}
}

j=0;
for(i=0;i<n;i++)
{
if(strcmp(p[i].team_name,p[j].team_name)!=0 || i==0)
{
printf("\n Team Name: %s",p[i].team_name);
j=i;
}
printf("\n Player Name    = %s",p[i].player_name);
printf("\n Batting Average = %f",p[i].batting_avg);
}
return 0;
}

```

Design a structure student_record to contain name, branch and total marks obtained. Develop a program to read data for 10 students in a class and print them.

*/

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

```
struct student_record
```

```
{  
    char name[20];  
    char branch[20];  
    int total_marks;  
}p[10];
```

```
int main(void)
```

```
{  
    int i=0,n=10;
```

```
    for(i=0;i<n;i++)
```

```
    {  
        printf("\n Enter Student Name : ");  
        scanf("%s",p[i].name);  
        printf("\n Enter Students Branch : ");  
        scanf("%s",p[i].branch);  
        printf("\n Enter Students Marks : ");  
        scanf("%d",&p[i].total_marks);  
    }
```

```
    for(i=0;i<n;i++)
```

```
    {  
        printf("\n Student %d Detail",i+1);  
        printf("\n Name      = %s",p[i].name);  
        printf("\n Branch    = %s",p[i].branch);  
        printf("\n Total marks = %d",p[i].total_marks);  
    }  
    return 0;  
}
```

What is Union?

- Union is user defined data type just like structure.
- Each member in structure is assigned its own unique storage area where as in Union, all the members share common storage area.
- All members share the common area so only one member can be active at a time.
- Unions are used when all the members are not assigned value at the same time.

Example:

union book

{

ch r title[100];

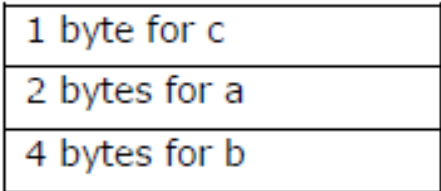
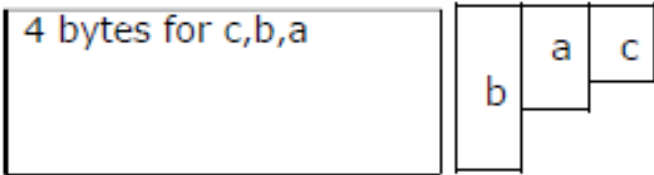
ch r author[50];

int pages;

float price;

};

Difference between Structure and Union

Structure	Union
Each member is assigned its own unique storage area.	All members share the same storage area.
Total memory required by all members is allocated.	Maximum memory required by the member is allocated.
All members are active at a time.	Only one member is active a time.
All members can be initialized.	Only the first member can be initialized.
Requires more memory.	Requires less memory.
<p>Example:</p> <pre>struct SS { int a; float b; char c; };</pre> 	<p>Example:</p> <pre>union UU { int a; float b; char c; };</pre> 
Total bytes = 1 + 2 + 4 = 7 bytes.	4 bytes are there between a,b and c because largest memory occupies by float which is 4 bytes.

Explain nested structure with example

- A structure that contains another structure as a member variable is known as nested structure or structure within a structure.
- Structure which is part of other structure must be declared before the structure in which it is used.

Example:

```
#include<stdio.h>
#include<conio.h>
struct address
{
    char add1[50];
    char add2[50];
    char city[25];
};
struct employee
{
    char name[100];
    struct address a;
    int salary;
};
```

```
void main()
{
    struct employee e;
    printf("enter name,address,city,salary");
    scanf("%s",e.name);
    scanf("%s",e.a.add1);
    scanf("%s",e.a.add2);
    scanf("%s",e.a.city);
    scanf("%d",&e.salary);
    printf("detail of employaee:");
    printf("%s",e.name);
    printf("%s",e.a.add1);
    printf("%s",e.a.add2);
    printf("%s",e.a.city);
    printf("%d",e.salary);
    getch();
}
```


Explain Array of Structure with Example?

- As we hav an array of basic data types, same way we can have an array variable of structure.
- It is better to use array of size 66 instead of 66 variables to store result of 66 student.

Following example shows how an array of structure can be used

```
#include<stdio.h>
#include<conio.h>
struct result
{
    char name[100];
    int rollno;
    float cpi;
};
```

```
void main()
{
    struct result r[66];
    int i;
    printf("enter detail of student :");
    for(i=0;i<66;i++)
    {
        printf("\nenter name,roll no,cpi");
        scanf("%s",r[i].name);
        scanf("%d%f",&r[i].rollno,&r[i].cpi);
    }
    printf("\n detail of student:\n");
    for(i=0;i<66;i++)
    {
        printf("%s",r[i].name);
        printf("\t%d",r[i].rollno);
        printf("\t%f\n",r[i].cpi);
    }
    getch();
}
```

File Management

- In real life, we want to store data permanently so that later on we can retrieve it and reuse it.
 - A file is a collection of bytes stored on a secondary storage device like hard disk, pen drive, and tape.
 - There are two kinds of files that programmers deal with text files and binary files.
- Text file are human readable and it is a stream of plain English characters.
- Binary files are not human readable. It is a stream of processed characters and Ascii symbols.

File Opening Modes

- We want to open file for some purpose like, read file, create new file, append file, read and write file,
etc...
- When we open any file for processing, at that time we have to give file opening mode.
- We can do limited operations only based on mode in which file is opened.
e.g. `fp = fopen("demo.txt","r");` // here file is opened in read only mode.

C has 6 different file opening modes for text files,

1. **r** open for reading only.
2. **w** open for writing (If file exists then it is overwritten)
3. **a** open for appending (If file does not exist then it creates new file)
4. **r+** open for reading and writing, start at beginning
5. **w+** open for reading and writing (overwrite file)
6. **a+** open for reading and writing, at the end (append if file exists)

Write a C program to display file on screen.

```
#include <stdio.h>
void main()
{
    FILE *fp;                // fp is file pointer. FILE is a structure defined in stdio.h
    char ch ;

    fp = fopen("prog.c", "r"); // Open Prog.c file in read only mode.
    c = getc(fp) ;
    while (ch != EOF)         // EOF = End of File. Read file till end
    {
        putchar(ch);
        ch = getc (fp);
        //Reads single character from file and advances position to next character
    }
    fclose(fp);              // Close the file so that others can access it.
}
```

Write a C program to copy a file.

```
#include <stdio.h>
void main()
{
    FILE *p, *q;
    char ch;
    p = fopen("Prog.c", "r");
    q = fopen("Prognew.c", "w");
    ch = getc(p);
    while(ch != EOF)
    {
        putc(ch, q);
        ch = getc(p);
    }
    printf("File is copied successfully. ");
    fclose(p);
    fclose(q);
    return 0;
}
```

Explain file handing functions with example.

C provides a set of functions to do operations on file. These functions are known as file handling functions.

Each function is used for some particular purpose.

fopen() (Open file)

- fopen is used to open a file for operation.
- Two arguments should be supplied to fopen function,
 - File name or full path of file to be opened
 - File opening mode which indicates which type of operations are permitted on file.
- If file is opened successfully, it returns pointer to file else NULL.

Example:`fp = fopen("Prog.c","r");` // File name is prog.c and it is opened for reading only.

fclose()

(Close file)

// File name is prog.c and it is opened for reading only.

- Opened files must be closed when operations are over.
 - The function fclose is used to close the file i.e. indicate that we are finished processing this file.
 - To close a file, we have to supply file pointer to fclose function.
- If file is closed successfully then it returns 0 else EOF.

Example: fclose(fp);

fscanf() (Read formatted data from file)

- The function fscanf() reads data from the given file.
- It works in a manner exactly like scanf(), only difference is we have to pass file pointer to the function.
- If reading is succeeded then it returns the number of variables that are actually assigned values, or EOF if any error occurred.

Example: fscanf(fp, "%d", &sum);

fseek() (Reposition file position indicator)

- Sets the position indicator associated with the file pointer to a new position defined by adding offset to a reference position specified by origin.
- You can use fseek() to move beyond a file, but not before the beginning.
 - fseek() clears the EOF flag associated with that file.
- We have to supply three arguments, file pointer, how many characters, from which location.
 - It returns zero upon success, non-zero on failure.

The origin value should have one of the following values

Name

SEEK_SET

SEEK_CUR

SEEK_END

Explanation

Seek from the start of the file

Seek from the current location

Seek from the end of the file

ftell()

`fseek(fp,9,SEEK_SET); // Moves file position indicator to 9th position from begging.`

(Get current position in file)

It returns the current value of the position indicator of the file.

For binary streams, the value returned corresponds to the number of bytes from the beginning of the file.

Example: `position = ftell (fp);`

rewind() (Set position indicator to the beginning)

- Sets the position indicator associated with file to the beginning of the file.
- A call to rewind is equivalent to: `fseek (fp, 0, SEEK_SET);`
- On file open for update (read+write), a call to rewind allows to switch between reading and writing.

Example:`rewind (fp);`

getc()

(Get character from file)

- getc function returns the next character from file or EOF if the end of file is reached.
- After reading a character, it advances position in file by one character.
 - getc is equivalent to getchar().
 - fgetc is identical to getc.

Example: `ch = getc(fp);`

putw() (Write integer to file)

- putw function writes integer to file and advances indicator to next position.
- It succeeded then returns same integer otherwise EOF is returned.

Example: putw(l, fp);

putc() (Write character to file)

- putc writes a character to the file and advances the position indicator.
- After reading a character, it advances position in file by one character.
- putc is equivalent to putchar().
 - fgetc is identical to putc.

Example: putc(ch, fp);

getw() (Get integer from file)

- getw function returns the next int from the file. If error occurs then EOF is returned.

Example: i = getw(fp);

Static Memory Allocation

- If memory is allocated to variables before execution of program starts then it is called static memory allocation.
- It is fast and saves running time.
- It allocates memory from stack.
- It is preferred when size of an array is known in advance or variables are required during most of the time of execution of program.
- Allocated memory stays from start to end of program.
- The storage space is given symbolic name known as variable and using this variable we can access value.
- e.g.
`int i;`
`float j;`

Dynamic Memory Allocation

- If memory is allocated at runtime (during execution of program) then it is called dynamic memory.
- It is bit slow.
- It allocates memory from heap
- It is preferred when number of variables is not known in advance or very large in size.
- Memory can be allocated at any time and can be released at any time.
- The storage space allocated dynamically has no name and therefore its value can be accessed only through a pointer.
- e.g.
`p = malloc(sizeof(int));`

Explain various functions used in Dynamic Memory Allocation.

malloc()

- malloc() is used to allocate a certain amount bytes of memory during the execution of a program.
- malloc() allocates size_in_bytes bytes of memory from heap, if the allocation succeeds, a pointer to the block of memory is returned else NULL is returned.
- malloc() returns an uninitialized memory for you to use.
- Malloc(can be used to allocate space for complex data types such as structures.
- Syntax:

- Syntax: `ptr_var = (cast_type *)malloc(size_in_bytes);`
- Examl :

```
#include<stdio.h>
int main()
{
    int *p ;
    p = (int *)malloc(sizeof(int));
    *p =25;
    printf("%d", *p);
    free(P);
}
```

- `calloc()`

- `calloc()` is used to allocate a block of memory during the execution of a program, e.g. for an array.

- `calloc()` allocates a region of memory large enough to hold `no_of_blocks` of size `size_of_block` each, if the allocation succeeds then a pointer to the block of memory is returned else `NULL` is returned.

- **Syntax:** ptr var=(cast type *) calloc(no_of_blocks ,size_of_block);
- **Example :**

```
#include<stdio.h>
int main ()
{
    int i,n;
    int *p;
    printf ("Enter how many numbers:");
    scanf ("%d",&n);
    p = (int*) calloc (n, sizeof(int));

    for (i=0; i<n; i++)
    {
        scanf ("%d",p);
        p++;
    }
}
```

- `realloc()`

- `realloc()` reallocates a memory block with a specific new size. If you call `realloc()`, the size of the memory block pointed to by the pointer is changed to the given size in bytes. This way you are able to expand and reduce the amount of memory you want to use.
- It is possible that the function moves the memory block to a new location; then the function returns address of new location. Old memory block is copied to new memory and old memory is released automatically.
- The content will remain unchanged means it is copied to new location.
- If the pointer is `NULL` then the `realloc()` will behave exactly like the `malloc()`. It will assign a new block of a size in bytes and will return a pointer to it.

- Syntax: `ptr_var=* realloc (void * ptr, size_t size);`
- Example

```
#include<stdio.h>
int main()
{
    int *p ;
    p = (int *)malloc(sizeof(int));
    *p =25;
    p = (int *)realloc(p, 2 * sizeof(int));
    printf("%d", *p);
    free(P);
}
```

: free());

When the memory is not needed anymore, you must release it calling the function free.

- Just pass the pointer of the allocated memory to free function and memory is released.

- **Syntax:**

- **Example**

```
void free(void *pointer);
```

```
: free(p);
```


Storage Class

- Storage class decides the scope, lifetime and memory allocation of variable.
- Scope of a variable is the boundary within which a variable can be used.
 - Four storage classes are available in C,
 - o Automatic (auto)
 - o Register (register)
 - o External (extern)
 - o Static (static)

automatic:

- Variables which are declared in function are of automatic storage class.
- Automatic variables are allocated storage in the main memory of the computer.
- Memory is allocated automatically upon entry to a function and freed automatically upon exit from the function.
- The scope of automatic variable is local to the function in which it is declared.
- It is not required to use the keyword **auto** because by **default storage class within a block is auto.**

Example:

```
int a; auto
```

```
int a;
```

Register:

- Automatic variables are allocated storage in the main memory of the computer; However, for most computers, accessing data in memory is considerably slower than processing directly in the CPU.
- Register variables are stored in registers within the CPU where data can be stored and accessed quickly.
- Variables which are used repeatedly or whose access time should be fast may be declared to be of storage class **register**.
- Variables can be declared as a register: `register int var;`

External:

- Automatic and register variables have limited scope and limited lifetimes in which they are declared.
 - Sometimes we need global variables which are accessible throughout the program.
- `extern` keyword defines a global variable that is visible to ALL functions.
- `external` is also used when our program is stored in multiple files instead of single file.
 - Memory for such variables is allocated when the program begins execution, and remains allocated until the program terminates. Memory allocated for an external variable is initialized to zero.
- Declaration for external variable is as follows: `extern int var;`

Static:

- static keyword defines a global variable that is visible to ALL functions in same file.
- Memory allocated for static variable is initialized to zero.
- Static storage class can be specified for automatic as well as external variables such as:
static extern int varx; //static external
static int var; // static automatic
- Static automatic variables continue to exist even after the function terminates.
- The scope of static automatic variables is identical to that of automatic variables.

Explain Input / Output functions.

- C language provides a set of standard built-in functions which will do the work of reading or displaying data or information on the I/O devices during program execution.
- Such I/O functions establish an interactive communication between the program and user.

Formatted Input/Output functions

scanf()

- It is used to read all types of data.
- It cannot read white space between strings.
- It can read multiple data at a time by multiple format specifier in one scanf().
- Examl : scanf(“%d%d”, &a, &b);

printf()

- It is used to display all types of data and messages.
- It can display multiple data at a time by multiple format specifier in one printf().
- Examl : printf(“a=%d b=%d”, a, b);

Unformatted input output functions

gets()

- It is used to read a single string with white spaces.
- It is terminated by enter key or at end of line.
- Examp1 :

```
char str[10];  
gets(str);
```

getchar(), getche(), getch()

- It is used to read single character at a time.
- `getchar()` function requires enter key to terminate input while `getche()` and `getch()` does not require.
- `getch()` function does not display the input character while `getchar()` and `getche()` function display the input character on the screen.

- Examp1 :

```
char ch;
```

```
ch = getchar();
```

```
ch = getche();
```

```
ch = getch();
```


puts()

- It is used to display a string at a time.

- Examp

char

:

```
str[]="Hello";
```

```
puts(str );
```

putchar()

- It is used to display single character at a time.

- Examp : putchar(ch);

Character checking functions.

Character checking functions are available in ctype.h header file.

1. `isdigit(int);` for checking number (0-9)
2. `isalpha(int);` for checking letter (A-Z or a-z)
3. `isalnum(int);` for checking letter (A-Z or a-z) or digit (0-9)
4. `isspace(int);` for checking empty space
5. `islower(int);` for checking letter (a-z)
6. `isupper(int);` for checking letter (A-Z)
7. `ispunct(int);` for checking punctuation symbols (like - : , ; , { , } , ? , . etc.)

Example: Write a program to check the entered character is digit or not

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
    char c;
    clrscr();
    scanf("%c",&c);
    if(isdigit(c))
        printf("True");
    getch();
}
```

do...while loop

- In contrast to while loop, the body of the do...while loop is executed first and then the loop condition is checked.
- The body of the loop is executed at least once because do...while loop tests condition at the bottom of the loop after executing the body.

- do...while loop is also known as **exit control loop** because **first body statements are executed** and then control-statement is executed, thus condition checking happens at exit point.
- The general format of the do...while statement is:

- The general format of the do...while statement is:

initialization;

do

{

statement;

increment or decrement;

}

while(test-condition);

Example: To print first 10 positive integer numbers

```
void main()
{
    int i;
    i = 1;                \\ initialization of i
    do
    {
        printf("\t%d",i);  \\ statement execution
        i++;               \\ increment of control variable
    } while(i <= 10);      \\ condition checking
}
```

for Loop

- *for loop provides a more concise loop control structure.*

- The general form of the for loop is:

for (initialization; test condition; increment)

{

body of the loop;

}

- When the control enters for loop, the variables used in for loop is initialized with the starting value such as $i=0$, $count=0$. Initialization part will be executed exactly one time.
- Then it is checked with the given test condition. If the given condition is satisfied, the control enters into the body of the loop. If condition is not satisfied then it will exit the loop.

- After the completion of the execution of the loop, the control is transferred back to the increment part of the loop. The control variable is incremented using an assignment statement such as `i++`
- If new value of the control variable satisfies the loop condition then the body of the loop is again executed. This process goes on till the control variable fails to satisfy the condition.
- For loop is also **entry control loop because first control-statement is executed and if it is true then only body of the loop will be executed.**

Example: // The following is an example that finds the sum of the first ten positive integer numbers

```
void main()
{
    int i;                //declare variable
    int sum=0;
    for(i=1; i < = 10; i++) // for loop
    {
        sum = sum + i ;    // add the value of i and store it to sum
    }
    printf("%d", sum);
}
```

for Loop

```
for( i=1; i < = 10; i++)  
{  
    sum = sum + i ;  
}
```

while Loop

```
i=1;  
while(i<=10)  
{  
    sum = sum + i;  
    i ++;  
}
```

do...while Loop

```
i=1;  
do  
{  
    sum = sum + i;  
    i ++;  
} while(i<=10);
```

Entry control loop	Exit control loop
Entry control loop checks condition first and then body of the loop will be executed.	Exit control loop first executes body of the loop and checks condition at last.
Body of loop may or may not be executed at all.	Body of loop will be executed at least once because condition is checked at last.
for, while are example of entry control loop.	Do...while is example of exit control loop.

Explain break, continue, goto with example.

- Sometimes it is required to quit the loop as soon as certain condition occurs.
- For example, consider searching a particular number in a set of 100 numbers, as soon as the search number is found it is desirable to terminate the loop.
 - *A break statement is used to jump out of a loop.*
 - *A break statement provides an early exit from for, while, do...while and switch constructs.*
 - *A break causes the innermost enclosing loop or switch to be exited immediately.*

Example : Read and sum numbers till -1 is entered

```
void main()
{
    int i, num=0;

    float sum=0, average;
    printf("Input the marks, -1 to end\n");
    while(1)
    {
        scanf("%d", &i);
        if(i == -1)
            break;
        sum += i;
    }
    printf("%d", sum);
}
```

continue;

- The *continue* statement can be used to skip the rest of the body of an iterative loop.
- The *continue* statement tells the compiler, “SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION”.

Example: Find sum of 5 positive integers. If a negative number is entered then skip it.

```
void main()
{
    int i=1, num, sum=0;
    for (i = 0; i < 5; i++)
    {
        scanf("%d", &num);
        if(num < 0)
            continue;    // starts with the beginning of the loop
        sum+=num;
    }
    printf("The sum of positive numbers entered = %d", sum);
}
```

goto:

- The *goto statement is a jump statement which jumps from one point to another point within a function.*
- The *goto statement is marked by label statement. Label statement can be used anywhere in the function above or below the goto statement.*
- Generally *goto should be avoided because its usage results in less efficient code, complicated logic and difficult to debug.*

Example: Following program prints 10,9,8,7,6,5,4,3,2,1

```
void main ()
{
    int n=10; loop:
    printf("%d,",n);
    n--;
    if (n>0)
        goto loop;
}
```