UNIT – III

UNIT – III SYLLABUS

Chapter 1: Functions: - Basics of Functions, Types of Functions, Parameter Passing Mechanism, Recursion.

Chapter 2: Pointers

Chapter 3: Strings and Standard Functions

Chapter 4: Dynamic Memory Allocation and Linked List: Dynamic Memory Allocation, Memory Models, And Memory Allocation Functions.

Chapter 5: Storage Class: -Automatic Variables, External Variables, Static Variables, Register Variables.

Programming Exercises for Unit - III:

1. Functions - Insertion sort, Linear search.
2. Recursive functions to find factorial & GCD (Greatest Common Divisor),
3. String operations using pointers and pointer arithmetic and dynamic memory allocation.
4. Swapping two variable values.
5. Sorting a list of names using array of pointers.

UNIT – III

CHAPTER 1: FUNCTIONS

- ✓ A function is a sub-program (or) block of statements which is used to perform a particular task.
- ✓ Function is executed when we call it.

Function Definition

Syntax: return_type function_Name (Parameter/Parameter List)

```
{
        local declaration;
        Statement1;
        Statement2;          /*Body of the function definition*/
        return (value);
}
```

TYPES OF FUNCTION:

Based on parameters presented and return values to the calling function; functions are categorized into 4 types.

1. **without parameters and without return values**
2. **with parameters but without return values**
3. **With parameters and with return values**
4. **Without parameters but with return values**

1. without parameters and without return values

Calling function	Analysis	Called function

Void main()
{
……....
abc();
……..
}
	No parameterss are passed.
	No values are sent back.
		abc()
		{
		……..
		……..
		}

- Data is neither passed through the calling function nor sent back from the called function.
- There is no data transfer between calling and called function.
- The function is executed and nothing is obtained.
- In such functions are used to perform any operation independently. They read data values and print result in same block

UNIT – III

2. **<u>with parameters but without return values</u>**

Calling function	Analysis	Called function

Void main()
{
…….
abc(x);
……..
}
	Parameter(s) are passed.
	No values are sent back.
		abc(x)
		{
		……..
		……..
		}

- In the above function, parameters are passed through the calling function. The called function operates on the values but no result is sent back.
- Such functions are partly dependent on calling function. The result is used in called function an it doesnot return valu to main().

3. **<u>With parameters and return values</u>**

Calling function	Analysis	Called function

Void main()
{
int z;
…….
z = abc(x);
……
}
	Parameter(s) are passed.
	Values are sent back.
		abc(y)
		{
		……..
		y++;
		…….
		return (y);
		}

- In the above example, the copy of actual parameter is passed to the formal parameters ie., the value of x is copied to y.
- The return statement returns the incremented value of y. The return value is collected by Z.

Here, the data is transfer between calling and called functions.

Here, both calling and called functions are dependent to each other.

4. **<u>Without parameters and but with return values</u>**

Calling function	Analysis	Called function

Void main()
{
int z;
……....
z = abc( );
……..
}
	Parameter(s) are passed.
	Values are sent back.
		abc( )
		{
		int y = 5;
		……..
		y++;
		…….
		return (y);
		}

- In the above function no parameters passed through the calling function but the called function return value.
- The called function is independent.
- Here, both calling and called functions are partly communicated to each other.

UNIT – III

Example program:

```
#include<stdio.h>
    void main()
    {
            void add(), div(int x,int y);
            int sub();
            int mul(int x,int y);
            int s,m;
            clrscr();
            /* with out par and with out ret val*/
            add();
            /* with out par but with ret val*/
            s = sub();
            printf("\nsub is :%d",s);
            /* with par and with ret val*/
            m = mul(5,3);
            printf("\nmul is %d", m);
            /* with par but with out ret val*/
            div(4,2);
            getch();

    }
    void div(int x,int y)
    {
            printf("\ndiv of x anf y is :%d", (x/y));
    }
    void add()
    {
            int x=10,y=20,z;
            z=x+y;
            printf("\nsum of x and y is :%d",z);
    }
    int sub()
    {
            int x=20, y=5,z;
            z=x-y;
            return z;
    }
    int mul(int x,int y)
    {
            return (x*y);
    }
```

UNIT – III

PARAMETER PASSING MECHANISM:

We can pass the parameters in two ways.

1. **Call by value**
2. **Call by reference**

1. **Call by value:**

- The values of actual parameters are passed to the formal parameters and operation is done on the formal parameters.
- Any change in formal parameters made does not affect the actual parameters because formal parameters are just copy of actual parameters.

Example:

```
#include<stdio.h>
void main()
{
    int a=10,b=20;
    void swaping(int , int);
    clrscr();
    printf("Before swaping");
    printf("\na= %d \t b = %d",a,b);
    swaping(a, b);
    printf("\nAfter swaping");
    printf("\na= %d \t b = %d", a , b);
    getch();

}
void swaping(int  a , int  b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}
```

Output:

```
Before Swaping
10    20
After Swaping
10    20
```

2. **Call by reference:**

- Here, instead of passing values addresses (reference) passed. Function operates on address rather than values.
- Any change in formal parameters made does affect the actual parameters because the formal parametres are pointer to the actual patameters.

Example:

```
#include<stdio.h>
void main()
{
 int a=10,b=20;
 void swaping(int*,int*);
 clrscr();
 printf("Before swaping");
 printf("\na= %d \t b = %d",a,b);
 swaping(&a,&b);
 printf("\nAfter swaping");
 printf("\na= %d \t b = %d",a,b);
 getch();

}
void swaping(int *a,int *b)
{
 int *temp;
 *temp=*a;
 *a=*b;
 *b=*temp;
}
```

Output:
Before Swaping
10 20
After Swaping
20 10

RECURSION

- The function is call repeatedly by itself is called recursion.
- Recursion is one of the applications to the stack.

Types of recursions: Recursions are two types

i) **Direct recursion:** the direct recursion function call to itself till the condition is true. In this type of recursion, only one function is involved.

```
int num()
{
    ............
    ............
    num();
}
```

ii) Indirect recursion: In indirect recursion, a function call to another function and called function is called to calling function. In this type of recursion, two functions are call to each other.

```
int num()
{
    ............
    ............
    sum();
}
int sum()
{
    ................
    int num();
}
```

Two essential conditions are satisfied by the recursion

- First, every time a function calls itself directly or indirectly, the function should have condition to stop the recursion. Otherwise, an infinite loop is generated.
- Second, some people think recursion is very needless luxury in the programming language.

Example program: WAC to find factorial of given number using recursion

```
#include<stdio.h>
#include<conio.h>
int fact( int );
void main( )
{
        int f, n;
        clrscr();
        printf("\nEnter any number  :");
        scanf("%d", n);
        f = fact( n );
       printf("\nfactorial of given number  :%d", f);
}
int fact(int x)
{
      if( x == 1)
       {
         return (1);
      }
      return (x * fact( x - 1));
}
```

Output:-

```
        Enter any number  : 5
        factorial of given number  : 120
```

UNIT – III

CHAPTER 2: POINTERS

Pointer: a pointer is a memory variable that stores a memory address.

- The pointer can have name like variable but the pointer is always denoted by asterisk (*) operator.
- The size of pointer variable is either 16 bit or 32 bit that's depends upon compiler.

Pointer Declaration:

Syntax:

<data type> * PointerVariableName

(Or)

<data type>* PointerVariableName

Example : int *x;

Here, 'x' is an integer type pointer and it holds the address of any integer variable.

Initializing Pointer:

Syntax:

<datatype> *< PointerVariableName > = & (VariableName);

(Or)

<datatype> *< PointerVariableName > ;

< PointerVariableName > = & (VariableName);

Example:

```
int a = 10;
int * x = &a;
```

```
int a = 10
int * x;
x = &a;
```

Ex: write a program to demonstrate pointer

```
void  main()
{
        int  x = 10;
        int  *ptr;
        clrscr( );
        ptr = &x;
        printf("\naddress of x = %u", &x);
        printf("\naddress of x = %u", ptr);
        printf("\nvalue of x = %d",  x);
        printf("\nvalue of x = %d", *ptr);
        printf("\nvalue of x = %d", *(&x));
}
```

Output:-

```
address of x = 6453
address of x = 6453
value of x = 10
 value of x =10
value of x =10
```

Note: The indirection operator (*) is used in different ways with pointer.

1. **Declaration:** pointer is always denoted by asterisk (*) operator.
2. **Dereference:** The operator '*' is also called as dereferencing operator. It means, the value at that address stored by the pointer is retrived.

- Normal variable provides direct access to their own values, where as a pointer provides indeirect access to the values of a variable whose address stored.
- '**&**' is the address operator and represent the address of the variable. %u is used with on printf() for printing the address of a variable.

Features of pointers:

1. Pointers save the memory space.
2. Execution time with pointer is faster because data is manipulated with the address ie., direct access to memory location.
3. The memory is accessed efficiently with the pointers. The pointer assigns thememory space and also release it.
4. Pointers are used with the data structure.pointers are used in 2-D and multidimentional array.
5. We can access elements of any type of array irrespective of its range.
6. Pointers are used for file handling.
7. Pointers are used to allocate memory dynamically.

void pointers:

- Pointer can also be declared as void pointer.
- The void pointer cannot be dereferenced without explicit type conversion. Because of void the compiler cannot determine the size of the object that the pointer points to.
- When a pointer is declared as void two bytes are allocated to it.
- Void variables cannot be declared because memory is not allotted to them.

Example:

```
int a;
void  *x = &a;
*(int *) x = 10;
```

Example: write a program to demonstrate void pointer

```
void  main( )
{
        int  x ;
        char  ch;
        float  f;
        void  *ptr ;

        *ptr = &x;
        *(int *)ptr = 20;
        printf("\nvalue of x = %d", *ptr);

        ptr = &ch;
        *(char *)ptr = 'a';
        printf("\nvalue of ch = %c", *ptr);

        ptr = &f;
        *(float *)ptr = 20.5;
        printf("\nvalue of f = %f", *ptr);
}
```

Output:-

```
value of x = 20
value of ch = a
value of f = 20.5
```

Wild pointers:

- When a pointer points to an unallocated memory location or destroyed variable location, such a pointer is called wild pointer.
- Wild pointer generates garbage memory location.
- The pointer becomes wild due to the following reasons.
    1. Pointer declared but not initialized.
    2. Pointer alternation.
    3. Accessing the destroyed data.

Example: write a program to demonstrate void pointer

```
void  main()
{
        int  *x ;
        printf("\n%u", x);
}
```

Output:- 75432

Constant pointer:

- The address of a pointer cannot be modified during the execution, such a pointer is called constant pointer.

  Example int * const x = 10;

  In the above example, it is not possible to modify the address of 'x';

ARITHMETIC OPERATIONS WITH POINTERS:

i. **Addition:-** Addition of two variables through pointer can be possible. But addition of two addresses (pointess) not posible.

ii. **Subtraction:-** Subtraction of two variables through pointer can be possible. But subtraction of two addresses (pointess) not posible.

iii. **Multiplication:-** Multiplication of two variables through pointer can be possible. But multipication of two addresses (pointess) or multiplication of address with constant not posible.

Similarly, division and mod operations can be done.

- Increment, decrement, prefix and postfix operations can be performed using pointers
- If you increment or decrement a pointer variable, then the address is modified based on data type.

Data type	Initial address	Operation		Address after operations		Required byte
int i = 2	4046	++	--	4048	4044	2
char c = 'x'	4053	++	--	4054	4052	1
float f = 2.2	4058	++	--	4062	4062	4
long l=2	4060	++	--	4064	4064	4

Program for arithmetic operation using pointers

```
void main()
{
    int a =25, b = 10, *p, *q ;
    p = &a ;
    q = &b ;
```

```
    printf("\n Addition of a + b =%d", *p + b);
    printf("\n Subtraction of a - b =%d", *p - b);
    printf("\n Multiplication of a * b =%d", *p * *q);
}
```

Output:-

Addition of a + b = 35

Subtraction of a - b = 15

Multiplication of a * b = 250

POINTERS AND ARRAY:

- Array name itself is a pointer. In an array, the base address is stored in array name with that we can get the array elements.
- Array elements are stored in contiguous continuous memory locations.

Example:

```
Void main( )
{
        int  i,  x[] = {1, 2, 3, 4, 5};
        for(i=0; i<5; i++)
        {
            printf("%d\t", x[i]);
        }
}
```

Output:- 1 2 3 4 5

- The elements in an array are displayed in different ways.
    1. **i [p]**
    2. **i [x]**
    3. ***(x + i)**
    4. ***(i + x)**
- The result of all of them would be the same.

Example:

```
void main()
{
        int  x[] = {1, 2, 3, 4, 5}, i;
```

```
        printf("values of an array\n");
        for(i=0; i<5; i++)
        {
            printf("x[ i ] = ");
            printf("%d\t", x [ i ]);
            printf("%d\t", i[ x ]);
            printf("%d\t", *(x + i));
            printf("%d\t", *(i + x));
            printf("%u\t", &(i + x));
            printf("\n");
        }
}
```

Output:-

values of an array and address

x [0] = 1	1	1	1	4668
x [1] = 2	2	2	2	4670
x [2] = 3	3	3	3	4672
x [3] =4	4	4	4	4674
x [4] =5	5	5	5	4676

POINTERS AND TWO DIMENSIONAL ARRAYS:

- A matrix can be represented in two dimensional elements of an array. Here, first argument is row nad second argument is column number.
- To display the base address of a two dimensional array using a pointer by '&' operator.
- The '&' operator is prefix with an array name followed by element number, otherwise the compiler show an error.

Example:

```
void main()
{
        int  x[3][3] = {{1, 2, 3},{ 4, 5,6},{7,8,9}};
        int *ptr, i;
        printf("values of an array\n");
        ptr = &x[0][0];
        for(i=0; i<5; i++, j++)
        {
            printf("%d   [%u]",*ptr, ptr);
```

```
            ptr++;
        }
    }
```

Output:-

values of an array and address

1 [5072]	2 [5074]	3 [5076]
4 [5078]	5 [5080]	6 [5082]
7 [5084]	8 [5086]	9 [5088]

POINTERS AND STRINGS:

- In pointers an string we can assign the string directly or other string to pointer.
- Example : char ch[10] = "mahesh";

  char *ptr = ch;

  (Or)

  char *name = "mahesh";

```
Void main()
{
        char  *name = "mahesh";
        printf("Your name is :%s", *name);
}
```

Output:-

Your name is: mahesh

ARRAY OF POINTERS:

- Array of pointer is nothing but collection of similar types of addresses and all addresses shares common name.
- We are storing the address of variables for which we have to declare the array as pointer.
- **Syntax:** <datatype> *PointerName[size];
- **Example :** int *x[10];

  Here 'x' pointer stores the 10 integer type variable value addresses.

Example :

```
Void main()
{
        int  *ptr[5];
```

```
        int arr[5 ] = {3, 5, 6}, k;
        for(k=0; k<3; k++)
        {
                ptr[k] = &arr + k;
        }
        printf("element and address\n");
        for(k=0; k<3; k++)
        {
                printf("%d\t", *ptr[k]);
                printf("%u\n", ptr[k]);
        }
}
```

Output:-

```
element and address
3       3487
5       3489
6       3501
```

POINTERS TO POINTERS:

Pointer is a memory variable which is used to store the address of another variable.

The pointer variable containing the address of another pointer is called the pointer to pointer.

Example:

```
void main( )
{
        int a=2, *p, **q;
        p = &a;
        q = &p;
        printf("Value of a= %d and address = %u", a, &a);
        printf("Through *p value of a= %d and address = %u", *p, p);
        printf("Through **q value of a= %d and address = %u", **q, *q);
}
```

Output:-

```
Value of a= 2 and address = 4056
Through *p value of a= 2 and address = 4056
Through **q value of a= 2 and address = 4056
```

UNIT – III

CHAPTER 3:STRINGS AND STANDARD FUNCTIONS

- In c language, a sequence of characters, digits and symbols enclosed with in double quotation is clled a string.
- The string is always declared as character array and its elements are stored in contiguous memory locations.
- Every string is terminated with '\0' (NULL) character.
  Example: char name [ ] = { 'I', 'N', 'D', 'I', 'A', '\0' } ;

Declaration of string:

Syntax:

char StringName[size];

Example: char name[20];

Initialization of string: Assigning a group of characters to string

Syntax:

char StringName[size] = sequance of characters;

Example:

char name[ ] = " india " ;

- The compiler automatically inserts the NULL (\0) character at the end of the string.
  char name[6] = { 'I', 'N', 'D', 'I', 'A'} ;
- The compiler does not automatically insert the NULL (\0) character at the end of the string
  So it having the garbage value at the end of the string.

Ex program:

```
void main()
{
        char name1[ ] = { 'I', 'N', 'D', 'I', 'A'} ;
        char name2[ ] = "INDIA";
        char name3[ ] = { 'I', 'N', 'D', 'I', 'A', '\0'} ;
        printf("name 1 =%s\n",name1);
        printf("name 2 =%s\n",name2);
        printf("name 3 =%s\n",name3);
}
```

output:-

```
INDIAindia

INDIA
INDIA
```

UNIT – III

String Standard Functions:

C compiler provides some string handling functions and all functions are in under **string.h** header file

S.No	Functions	Description
1	strlen()	Determine the length of the string.
2	strcpy()	Copies the string from the source to destination.
3	strncpy()	Copies character of a string to another string up to the specified length.
4	strcmp()	Compare character of two strings(function differentiates from small and capital letters).
5	stricmp()	Compare character of two strings(function does not differentiates from small and capital letters)
6	strncmp()	Compare character of two strings up to the specified length.
7	strnicmp()	Compare character of two strings up to the specified length. Ignore case.
8	strlwr()	Converts uppercase character of string to lowercase.
9	strupr()	Converts lowercase character of string to uppercase.
10	strdup()	Duplicate a string.
11	strchr()	Determines the first occurrence of a given character in a string.
12	strrchr()	Determines the last occurrence of a given character in a string.
13	strsrt()	Determines the first occurrence of a given string in another string.
14	strcat()	Appends source string to the destination string.
15	strncat()	Appends source the string to the destination string up to specified length.
16	strrev()	Reversing all characters of a string.
17	strset()	Sets all characters of a string with a given argument or symbol.
18	strnset()	Sets specified number of characters of a string with a given argument or symbol..
19	strspn()	Find up to what length two strings are identical.
20	strpbrk()	Search the first occurrence of the character in a given string and then display the string starting from the character.

UNIT – III

1. **<u>strlen( ) :</u>** strlen( ) function counts the number of characters in a given string. This function return an integer value of count.

   Syntax:

   strlen(char *string)

   Example program:

```
void main()
{
        char text[20];
        int len;
        printf("type text below.\n");
        gets(text);
        len = strlen(text);
        printf("length of string = %d",len);
 }
```

   Output:-

```
type text below.
Hello
length of string = 5
```

2. **<u>strcpy( ):</u>** This function copies the content of source string to destination string. This function overwrite the contenet if any in the array.

   Syntax:

   strcpy(char *destination, char *source)

   Example program:

```
void main()
{
        char  s[20], d[20];
        printf("type text below.\n");
        gets(s);
        strcpy(d, s);
        printf("original string = %s",s);
        printf("duplicate string = %s",d);
 }
```

   Output:-

```
type text below.
Hello
original string = Hello
duplicate string = Hello
```

UNIT – III

3. **strncpy( ):** This function copies the string up to specified length of character from source to destination whereas strcpy() function copies the whole content of one string to another string.
   - This function overwrite the contenet if any in the array.

   Syntax:

   strncpy(char *destination, char *source, int n)

   Example program:

```
void main()
{
        char  s[20], d[20];
        int n;
        printf("Enter source string :");
        gets(s);
        printf("Enter destination string :");
        gets(d);
        printf("No. of characters to replace :");
        scanf("%d", &n);
        strncpy(d, s, n);
        printf("source string = %s",s);
        printf("destination string = %s",d);
 }
```

   Output:-

```
Enter source string : wonderful
Enter destination string : beautiful
No. of characters to replace : 6
source string = wonderful
destination string = wonderful
```

4. **strcmp( ):** This function is compare two strings with differentiate between lowercase and uppercase.
   - If two strings are same then it return zero (0) otherwise return non-zero value.

   Syntax :

   strcmp(char *s1, char *s2)

   Example program:

```
void main()
{
        char  s[20], d[20];
        int diff;
        printf("Enter source string :");
        gets(s);
        printf("Enter destination string :");
```

```
        gets(d);
        diff = strcmp(s, d);
         if(diff == 0)
                printf(" string are identical");
         else
                printf(" string are not identical");
 }
```

Output:-

```
Enter source string : wonderful
Enter destination string : wonderful
string are identical
```

5. **stricmp( ):** This function is compare two strings with out differentiate between lowercase and uppercase.

- If two strings are same then it return zero (0) otherwise return non-zero value.

Syntax:

stricmp(char *s1, char *s2)

Example program:

```
void main()
{
        char  s[20], d[20];
        int diff;
        printf("Enter source string :");
        gets(s);
        printf("Enter destination string :");
        gets(d);
        diff = stricmp(s, d);
        if(diff == 0)
                printf(" string are identical");
        else
                printf(" string are not identical");
 }
```

Output:-

```
Enter source string : woNdeRful
Enter destination string : WonDerFul
string are identical
```

6. **strncmp( ):** This function is compare two strings up to specified length with differentiate between lowercase and uppercase.

- If two strings are same then it return zero (0) otherwise return non-zero value.

Syntax:

strncmp(char *s1, char *s2, int n)

Example program:

```
void main()
{
        char  s[20], d[20];
        int diff,n;
        printf("Enter source string :");
        gets(s);
        printf("Enter destination string :");
        gets(d);
        printf("Enter length to compare string :");
        scanf("%d", &n);
        diff = strncmp(s, d, n);
         if(diff == 0)
                printf(" string are identical up to %d characters", n);
         else
                printf(" string are not identical");
 }
```

Output:-

```
Enter source string : hello
Enter destination string : hello man
Enter length to compare string : 2
string are identical up to 2 characters
```

7. **strnicmp( ):** This compare two strings up to specified length without case sensitive.

- The two strings are same then it return zero (0) otherwise return non-zero value.

Syntax:

strnicmp(char *s1, char *s2, int n)

Example program:

```
void main()
{
        char  s[20], d[20];
        int diff,n;
        clrscr();
        printf("Enter source string :");
```

```
        gets(s);
        printf("Enter destination string :");
        gets(d);
        printf("Enter length to compare string :");
        scanf("%d", &n);
        diff = strnicmp(s, d, n);
         if(diff == 0)
                printf(" strings are identical up to %d characters", n);
         else
                printf(" strings are not identical");
 }
```

Output:-

```
Enter source string : Gud luck
Enter destination string : gud boy
Enter length to compare string : 3
strings are identical up to 3 characters
```

8. **<u>strlwr( ):</u>** This function can be used to convert any string to lowercase.

Syntax

strlwr(char *string)

Example program:

```
void main()
{
        char  upr[20];
        printf("Enter source string :");
        gets(upr);
        printf(" After strlwr() string is:  %s", strlwr(upr));
 }
```

Output:-

```
Enter source string : ABCDE
    After strlwr() string is:  abcde
```

9. **<u>strupr( ):</u>** This function can be used to convert any string to uppercase.

Syntax:

strupr(char *string)

Example program:

```
void main()
{
        char  lwr[20];
        clrscr();
        printf("Enter source string :");
        gets(lwr);
        printf(" After strupr() string is:  %s", strupr(upr));
 }
```

Output:-

```
Enter source string : abcde
After strlwr() string is:  ABCDE
```

10. **strdup( ):** This function is used for duplicating a given string at allocated memory which is pointed by the pointer variable.
    - strdup( ) return the address of duplicate string.

Syntax

ptr = strdup(char *str)

where, ptr is pointer variable and str is string.

Example program:

```
void main()
{
        char  str[20], *ptr;
        printf("Enter source string :");
        gets(str);
        ptr = strdup(str);
        printf(" original string is:  %s", str);
        printf(" duplicate string is:  %s", ptr);
 }
```

Output:-

```
Enter source string : abcde
original string is: abcde
duplicate string is: abcde
```

11. strchr( ): This function return the pointer position to the first occurrence of the character is given string.

Syntax:

strchr(char *string, char ch)

Example program:

```
void main()
{
        char  str[20], *ptr, ch;
        printf("Enter source string :");
        gets(str);
        printf("Enter character to find :");
        ch = getchar();
        ptr = strchr(str, ch);
        if(ptr)
                printf(" character  %c  found in string.", ch);
        else
                printf(" character  %c  not found in string.", ch);
}
```

Output:-

```
Enter source string : welcome to college
Enter character to find : o
character  o  found in string.
```

12. strrchr( ): This function return the pointer position to the last occurrence of the character in the given string. The syntax is as follow

strrchr(char *string, char ch)

Example program:

```
void main()
{
        char  str[20], *ptr, ch;
        printf("Enter source string :");
        gets(str);
        printf("Enter character to find :");
        ch = getrchar();
        ptr = strrchr(str, ch);
        if(ptr)
                printf(" character  %c  found in string.", ch);
        else
                printf(" character  %c  not found in string.", ch);
}
```

Output:-

```
Enter source string : welcome to college
Enter character to find : o
character  o  found in string.
```

13. <u>strstr( ):</u> This function finds the destination string in the source string.

This function returns the pointer position to the source string where destination string starts.

Syntax:

strstr(char *s, char *d)

Example program:

```
void main()
{
        char  s[20], d[20], *ptr;
        printf("Enter source string :");
        gets(s);
        printf("Enter destination string :");
        gets(d);
        ptr = strstr(s, d);
        if(ptr)
                printf(" string is found.");
        else
                printf("string is not found.");
}
```

Output:-

```
Enter source string: welcome to college
Enter destination string: to
    string is found.
```

14. <u>strcat( ):</u> This function appends the target string to source string.

Syntax

strcat(char *source, char *target)

Example program:

```
void main()
{
        char  s[20], t[20];
        printf("Enter source string :");
        gets(s);
        printf("Enter target string :");
        gets(t);
```

```
        strcat(s, "  ");
        strcat(s, t);
        printf("%s", s);
}
```

Output:-

Enter source string: I am
Enter target string: an Indian
I am an Indian

15. <u>strncat( ):</u> This function appends the target string to source string.

Syntax:

strncat(char *source, char *target, int n)

Example program:

```
void main()
{
        char  s[20], t[20];
         int n;
        printf("Enter source string :");
        gets(s);
        printf("Enter target string :");
        gets(t);
        printf("Enter number of characters to add :");
        scanf("%d", &n);
        strcat(s,"  ");
        strncat(s, t, n);
        printf("%s", s);
}
```

Output:-

Enter source string: may i
Enter target string: come in?
may I come

16. <u>strrev( ):</u> This function simply reverse the given string.

Syntax:

strrev(char *string)

Example program:

```
void main()
{
        char  s[20];
        printf("Enter source string :");
        gets(s);
        printf("Reverse string\n");
        strrev(s);
        puts(s);
 }
```

Output:-

```
Enter source string: abcdef
Reverse string
fedcba
```

17. <u>strset( ):</u> This function replace every character of a string with the symbol given by the the programmer.

Syntax:

strset(char *string, char symbol)

Example program:

```
void main()
{
        char  s[20], symbol;
        printf("Enter source string :");
        gets(s);
        printf("Enter symbol :");
        symbol = getchar();
        printf("before set :%s", s);
        strset(s, symbol);
        printf("after set :%s", s);
 }
```

Output:-

```
Enter source string: abcdef
Enter symbol : @
before set : abcdef
after set : @@@@@@
```

18. <u>strnset( ):</u> This function replace specified length of character of a string with the symbol given by the programmer.

Syntax: **strset(char *string, char symbol, int n)**

Example program:

```
void main()
{
        char  s[20], symbol;
        int n;
        printf("Enter source string :");
        gets(s);
        printf("Enter symbol :");
        symbol = getchar();
        printf("Enter number of characters to be replace :");
        scanf("%d", &n);
        printf("before set :%s", s);
        strset(s, symbol, n);
        printf("after set :%s", s);
 }
```

Output:-

```
Enter source string: abcdef
Enter symbol : @
Enter number of characters to be replace : 4
before set : abcdef
after set : @@@@ef
```

19. **strspn():** This function return the position of the string from where the sourcestring is not matching the targe t string.

Syntax

strspn(char *source, char *target)

Example program:

```
void main()
{
        char  s[20], t[20];
         int len;
        printf("Enter source string :");
        gets(s);
        printf("Enter target string :");
        gets(t);
        printf("Enter number of characters to add :");
        scanf("%d", &n);
        len = strspn(s, t);
        printf("After %d characters is not match", len);
```

```
}
```

Output:-

```
Enter source string: good morning
Enter target string: good bye
After 5 characters is not match
```

20. **strpbrk( ):** this function search the first occurrence of character in a given string, and then display the string from that charater.

This function returns pointer position to the first occurrence of character.

Syntax

strpbrk(char *source, char *target)

Example program:

```
void main()
{
        char  s[20], ch, *ptr;
        printf("Enter source string :");
        gets(s);
        printf("Enter character to be search:");
        ch = getchar();
        ptr = strpbrk(s, ch);
        printf("string from given character  :%s", ptr);
}
```

Output:-

```
Enter source string: good morning
Enter character to be search: d
    string from given character: d morning
```

String Conversion functions:

1. **atof( ):** This function used to convert the given string to double.

Syntax:

double atof (const char *string)

Example program:

```
#include <stdlib.h>
void main()
{
    double  d;
```

```
        d = atof("99.1245");
        printf("%g", d);
}
```

Output:- 99.1245

2. atoi( ): This function used to convert the given string to int.

syntax:

int atof (const char *string)

Example program:

```
#include <stdlib.h>
void main()
{
        int d;
        d = atoi("99.1245");
        printf("%d", d);
}
```

Output:- 99

3. atoi( ): This function used to convert the given string to long.

Syntax:

long atof (const char *string)

Example program:

```
#include <stdlib.h>
void main()
{
        long d;
        d = atoi("99.1245");
        printf("%ld", d);
}
```

Output:- 99

4. strtod( ): This function used to separates char and float data from the given string. The syntax is as follow

double strtod (char *string1, char *string2)

Example program:

```
#include <stdlib.h>
void main()
{
        const char *string1 = "12.2% is rate of interest";
```

```
        char *string2;
        double  d;
        d = strtod(string1, &string2);
        printf("%g", d);
        printf("\n%s", string2);
}
```

Output:-

```
12.2
% is rate of interest
```

5. <u>strtol( ):</u> This function used to separates char and long int data from the given string. The syntax is as follow

double strtol (char *string1, char *string2)

Example program:

```
#include <stdlib.h>
void  main()
{
        const char *string1 = "12.2% is rate of interest";
        char *string2;
        long int  d;
        clrscr();
        d = strtol(string1, &string2);
        printf("%g", d);
        printf("\n%s", string2);
}
```

Output:-

```
12
% is rate of interest
```

MEMORY FUNCTIONS:

1. memcpy(): This function used to copies n number of characters from one string to another.

Example program:

```
void  main()
{
        char  *string1 = "mahesh and ntr";
        char   string2[20];
        clrscr();
        memcpy(string2, string1, 20);
        printf("\n%s", string2);
```

```
}
```

Output:-

mahesh and ntr

2. memove(): This function used to moves a specified range of char from one place to another.

Example program:

```
void  main()
{
        char  string[] = "good very good";
        clrscr();
        printf("\nbefore :%s", string);
        memove(string, & string[5], 10);
        printf("\nafter :%s", string);
}
```

Output:-

before :good very good
after :very good good

3. memcmp(): This function used to compares the content of the memory.

Example program:

```
void  main()
{
        char  string1[] = "a";
        char  string2[] = "A";
        int diff;
        diff = memcmp(string1,  string2,2);
        printf("\n%d", diff);
}
```

Output:- 32

4. memchr(): This function used to search for the first occurrence of the given character.

Example program:

```
void  main()
{
        char  *string1 = "c is easy";
        printf("\n%s", memchr(string1, 'e' ,10));
}
```

Output:- easy

UNIT – III

CHAPTER4: DYNAMIC MEMORY ALLOCATION

- The process of allocating memory during program execution is called dynamic memory allocation.
- All the dynamic memory allocation functions are defined in **stdlib.h** and **alloc.h**
- C language offers 4 dynamic memory allocation functions. They are,

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

1. malloc( ):

- malloc () function is used to allocate block of memory during the execution of the program.
- malloc () reserves bytes of determined size and return the base address to pointer variable.
- malloc () function returns null pointer if it couldn't able to allocate requested amount of memory.

Syntax:

Pointer = (datatype *) malloc (size);

Example: char *x;
x = (char *) malloc (20);

example program for malloc() function in c:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    char *mem_allocation;
    mem_allocation = malloc( 20 * sizeof(char) );
    if( mem_allocation == NULL )
    {
        printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
        strcpy( mem_allocation,"welcome");
    }
    printf("Dynamically allocated memory content : " %s\n", mem_allocation );
    free(mem_allocation);
}
```

Output: Dynamically allocated memory content : welcome

2. calloc( ):

- malloc () function is used to allocate multiple block of memory during the execution of the program.

Syntax:

Pointer = (datatype *) malloc (total_size , block size);

Example: int *x;

x = (int *) malloc (20, 2);

example program for calloc() function in c:

```
#include <string.h>
#include <stdlib.h>
int main()
{
    char *mem_allocation;
    mem_allocation = calloc( 20 ,  sizeof(char) );
    if( mem_allocation== NULL )
    {
      printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
       strcpy( mem_allocation,"welcome");
    }
       printf("Dynamically allocated memory content   : " %s\n", mem_allocation );
       free(mem_allocation);
}
```

Output: Dynamically allocated memory content : welcome

3. realloc( ):

- realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.
- If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

4. free( ):

- free( ) function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.

Syntax :

free(pointer);

Example: int *x;

x = (int *) malloc (20, 2);

```
free(x);
```

example program for realloc() and free() functions in c:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
   char  *mem_allocation;
   mem_allocation = malloc( 20 * sizeof(char) );
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
     strcpy( mem_allocation,"welcome");
   }
   printf("Dynamically allocated memory content  : " %s\n", mem_allocation );
   mem_allocation= realloc (mem_allocation,100*sizeof(char));
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
      strcpy( mem_allocation, "space is extended upto 100 characters");
   }
   printf("Resized memory : %s\n", mem_allocation );
   free(mem_allocation);
}
```

<u>Output</u>

```
Dynamically allocated memory content : welcome
Resized memory : space is extended upto 100 characters
```

UNIT – III

DIFFERENCE BETWEEN MALLOC() AND CALLOC() FUNCTIONS IN C:

malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
int *ptr; ptr = malloc(20 * sizeof(int)); For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr; Ptr = calloc(20, 20 * sizeof(int)); For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
malloc () doesn't initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
type cast must be done since this function returns void pointer int *ptr; ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr; ptr = (int*)calloc(20, 20 * sizeof(int));

UNIT – III

CHAPTER 5: STORAGE CLASSES

Based on variable declaration, the storage classes are categorized in 4 ways

1. **AUTOMATIC VARIABLES**
2. **EXTERNAL VARIABLES**
3. **STATIC VARIABLES**
4. **REGISTER VARIABLES**

1. AUTOMATIC VARIABLES:

- The automatic variables are declared by using **auto** keyword.
- The auto variables are defined in local declaration. A local variable without storage class
- having default auto variable.
- The storage area of a variable is stack. Stack is a temporary storage.
- The initial value of variable is garbage values.
- The scope of a variable is within the function or block only.
- The life time of a variable is up to the function or block is in active state. After that the variables are vanish.

Example: write a program to show the working of the auto variable

```
void main( )
{
        auto int v = 10;
        fun2( );
        printf("\n V = %d", v);
}
void fun1( )
{
        auto int v = 20;
        printf("\n V = %d", v);
}
void fun2( )
{
        auto int v = 30;
        fun1( );
        printf("\n V = %d", v);
}
```

Output: V =20
V =30
V =10

UNIT – III

2. EXTERNAL VARIABLES:

- The external variables are declared by using **extern** keyword.
- The external variables are defined in before main( ), use of extern keyword is optional.
- The external variables are declared in global declaration or other place of program. And also declared in another source file.
- The storage area of a variable is permanent storage i.e., variables are active until the program is running.
- The initial value of external variable is cannot be done because its values in another source file.
- The scope of a variable is entire program.
- The life time of a variable is active until the program is running.

Note: - If both external and auto variables are declared with same name then the first priority given to the auto variable and external variables are hidden.

3. STATIC VARIABLES:

- The static variables are declared by using **static** keyword.
- The static variables are defined in both local and global declaration, depending upon the where you are declared.
- The storage area of a static variable is permanent storage i.e., variables are active until the program is running.
- The initial value of static variable is NULL (zero) value. The static variable is initialized only once.
- The scope of a variable is depends on declaration. If variable is declared inside the function
- then the scope is static local, if variable is declared outside the function then the scope is static global.
- The life time of a variable is active until the program is running.

Note: - The value of static variable is continues at each call and last change made in the value of static variable remains throughout the program execution.

Example: write a program to show the working of the auto variable

```
void main( )
{
        fun1( );
        fun1( );
        fun1( );
        printf("\n V = %d", v);
}
void fun1( )
{
        int  static  m;
        m++;
        printf("\n m = %d", m);
}
```

Output: m = 1

m=2
m=3

4. REGISTER VARIABLES:

- The register variables are declared by using **register** keyword. It indicates the variables are stored in some CPU registers.
- The registers are access the memory very fast. But the memory size is limited.
- Once the memory limit is reached, then the register variables are converted into non-register variable as auto variables.
- We cannot use register for all type of data variables.
- For example the register size of 8086 microcomputer is 16-bit. So we can store only char and int. It not support for float and double like that.

```
void main( )
{
        register int m = 1;
        clrscr();
        for( ;m<= 5; m++)
            printf("\n m = %d", m);
}
```

Output: 1 2 3 4 5