Chapter 7 : Arrays

- **Definition of Array :**

  An array is a sequenced collection of related data items that share a common name. In other words, An array can be is used to represent a list of numbers or a list of names.

- **An array is a derived data type.**

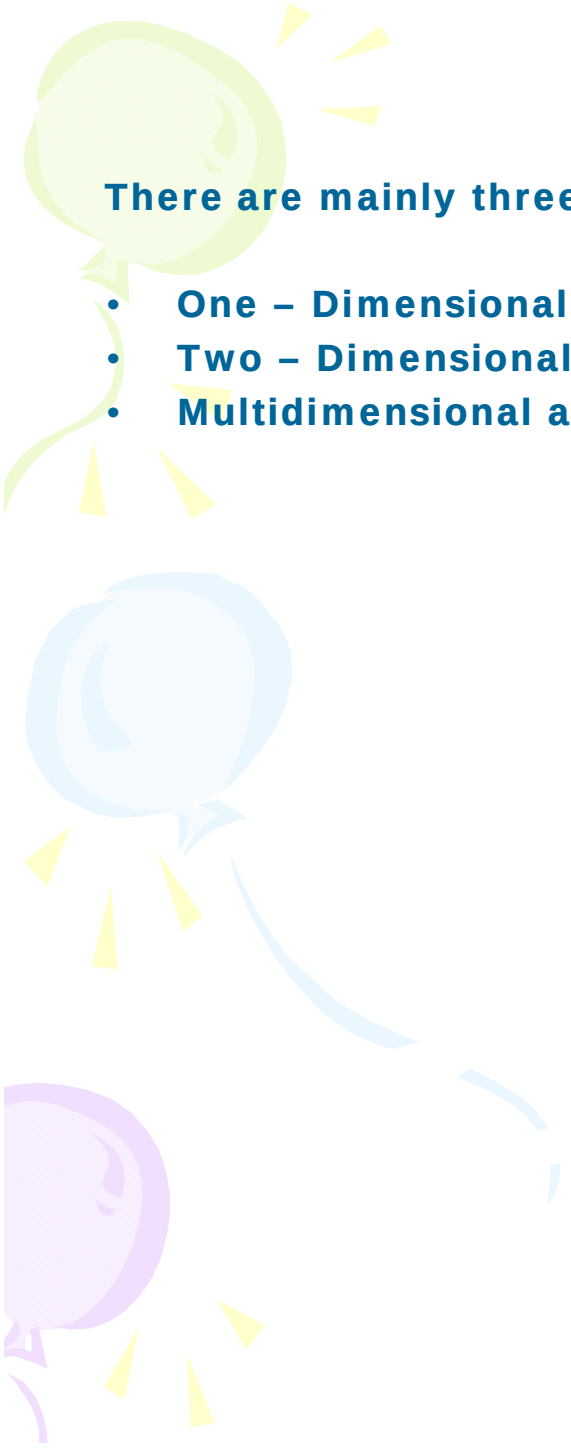
- **Examples: list of students, list of temperatures, list of employees, list of products etc.**

- **when we use array name salary to represent a set of salaries of a group of employee. We can refer to the individual salaries by writing a number called index or subscript in bracket after the array name**

  for example : salary[10];

  It represents the salary of 1o employee.

- **The complete set of values is refereed as array and individual values are called elements.**



There are mainly three types of the array:

- **One – Dimensional arrays**
- **Two – Dimensional arrays**
- **Multidimensional arrays**

❑ One – Dimensional Arrays:

A list of items can be given one variable name using only one subscript and such a variable is called a single subscripted variable or a one dimensional array.

Syntax:

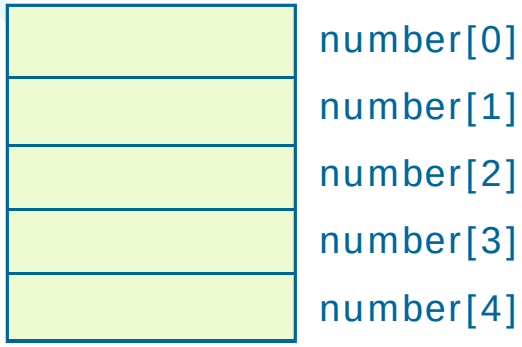
type variable-name[size];

The type specifies the data type of the element that will be contained in the array, such as int, float or char.

Ex: If we want to represent a set of five numbers say, (35,40,20,57,19) by an array variable number, then we may declare the variable number as follows.

int number[5];

And the computer reserves five storage location as shown below:



The value to the array element can be assigned as follow:

```
number[0] = 35 ;
number[1] = 40 ;
number[2] = 20 ;
number[3] = 57 ;
number[4] = 19 ;
```

This would case the array number to store the value as shown below:

35	number[0]
40	number[1]
20	number[2]
57	number[3]
19	number[4]

Same way,

float height[50];

Declares the height to be an array containing 50 real elements. Any subscript to 0 to 49 are valid.

- **Any references to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.**

- **The size should be either integer constant or symbolic constant.**

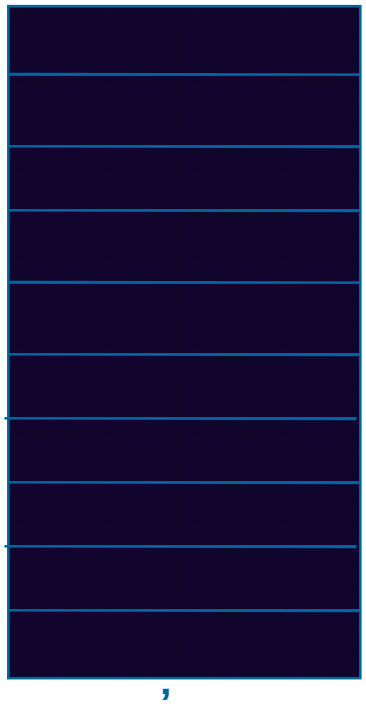
The C language treats character string simply as array of characters. The size in a character string represents the maximum number of characters that the string can hold.

char name[10] ;

Declare name as a character array variable that can hold a maximum of 10 characters. Suppose we read the following string constant into the string variable name.

“WELL DONE”

Each character of the string is treated as an element of the array name and is stored in the memory as follows:



- Character string terminates with an additional null character. Thus name[10] holds the null character '\0'. When declaring character array, we must allow one extra space for the null character.

INITIALIZATION OF ONE – DIMENSIONAL ARRAYS:

After an array is declared, its element must be initialized. Otherwise, they will contain “garbage”. An array can be initialized at either of the following stages:

At compile time
At run time

Compile Time Initialization:

Syntax:

Type array-name[size] = {list of values};

The value in the list are separated by commas.
Ex: int number[3] = {0,0,0} ;

Will declare the variable number as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically

float total[5] = { 0.0 , 15.75 , -10};

Will initialize the first three elements to 0.0, 15.75 and -10.0 and the remaining two elements to zero

The size may be omitted. In such cases, the compiler allocate enough spaces for all initialized elements.

Ex: int counter[] = {1,1,1,1}

Counter array contain four element with initial value 1.

Character array may be initialized similar manner.

char name[] = {'R', 'A', 'M', '\0'};

Is equivalent to :

char name[] = "RAM";

If we have more than the declared size, the compiler will produce an error. That is the statement.

int number[3] = {10, 20, 30, 40} ; will not work. It is illegal in C.

Run Time Initialization :

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays.

Ex:

```
for(i=0; i<100 ; i++)
{
        if (i < 50 )
                sum[i] = 0.0;
        else
                sum[i] = 1.0;
}
```

In the above case the first 50 elements are initialized to zero while the remaining elements are initialized to 1.0 are run time.

We can also use a read function such as scanf() to initialize an array.

Ex:

```
int  x[3];
scanf (“%d  %d  %d”, &x[0], &x[1], &x[2] ) ;
```

Will initialize array elements with the values entered through the kwyboard.

```
/* program:-Write a C program to arrange the accepted numbers in
   ascending order and descending order: */

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

void main()
{
   int i,j,ans,n,temp=0, no[20];
   clrscr();
   printf("Enter the size of the list:=");
   scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter the no:-");
                scanf("%d",&no[i]);
        }
   printf("Enter the your choice ascending=1&descending=2:=");
   scanf("%d",&ans);
```

```
for(i=0;i<n;i++)
{
        for(j=i;j<n;j++)
        {
                if(ans==1)
                {
                        if(no[i]>no[j])
                        {
                            temp=no[i];
                            no[i]=no[j];
                            no[j]=temp;
                           }
                }
                else

                if(no[i]<no[j])
                {
                        temp=no[i];
                       no[i]=no[j];
                        no[j]=temp;
                }
        }
  printf("%d",no[i]);
  printf("\n");
   }
 getch();
}
```

TWO – DIMENSIONAL ARRAYS:

If you want to store data into table or matrix form at that time you can use the two dimensional array. Consider the following data table, which shows the value of sales of three items by four sales man.

Ex:

	ITEM 1	ITEM2	ITEM 3
SALES MAN 1	310	275	365
SALES MAN 2	210	190	325
SALES MAN 3	405	235	240
SALES MAN 4	260	300	380

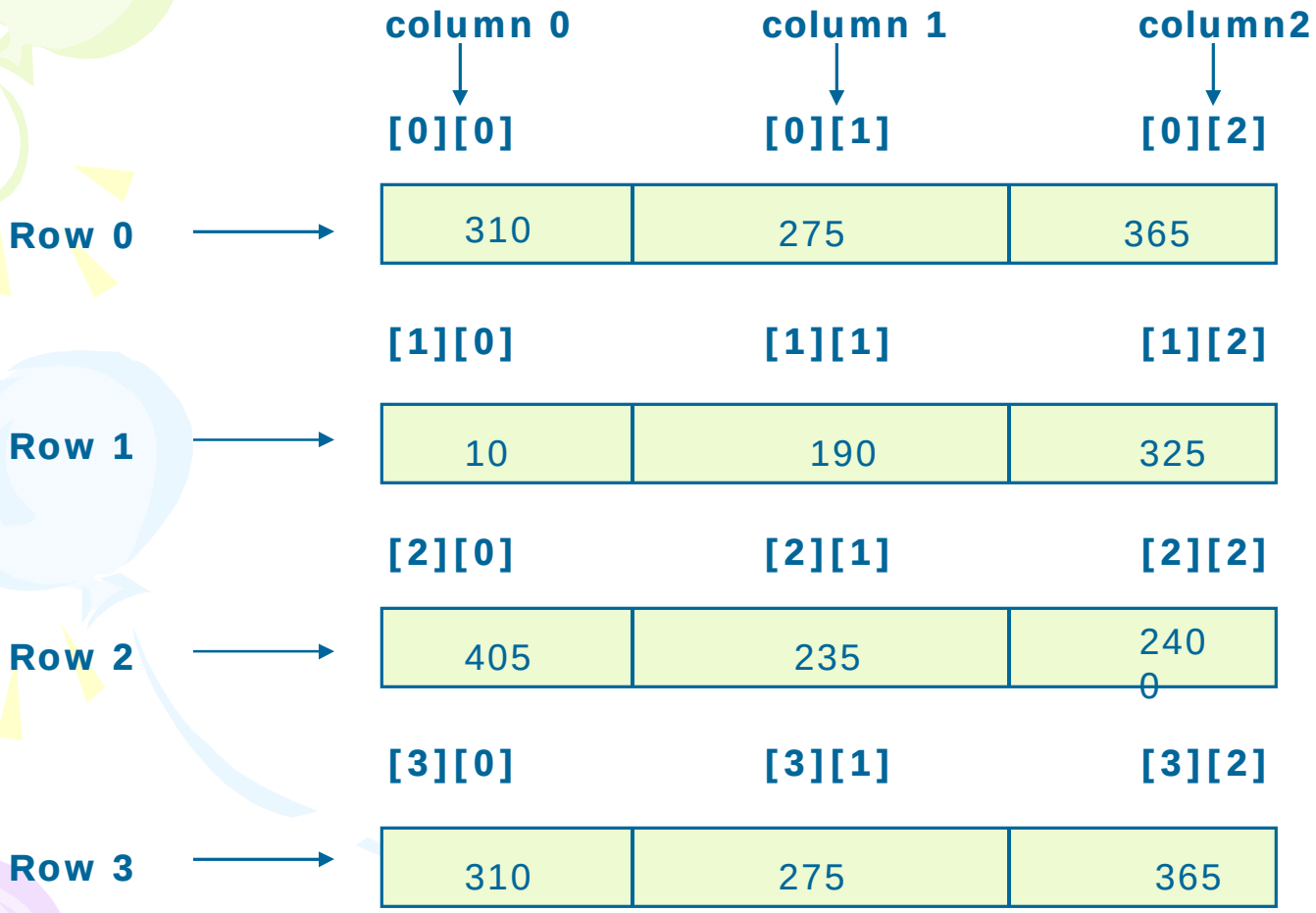
The table discussed above can be define as :

v[4][3] where 4 Rows and 3 columns.

Declaration of two dimensional array:

Syntax : **data_type array-name[rowsize][colsize];**

Memory representation of the two dimensional array for v[4][3] .



INITIALIZING TWO DIMENSIONAL ARRAY:

Ex: **int table[2][3] = {0,0,0,1,1,1};**

Initialize elements of the first row to 0 and second row to 1. The initialization is done row by row.

The above statement can be equivalent written as :

```
int table[2][3] = {{0,0,0},{1,1,1}};
```

We can also initialize a two dimensional array in the form of a matrix as shown below:

```
int table[2][3] =    {
                         {0,0,0},
                         {1,1,1}
                     } ;
```

When the array is completely initialized with all values, we need not specify the size of the first dimension. That is, the statement is permitted.

```
int table[][3] =     {
                         {0,0,0} ,
                         {1,1,1)
                     };
```

If the value is missing in an initializes, they are automatically set to zero.

Ex: int table[2][3] ={

{1,1} ,

{2}

} ;

Will initialize the first two elements of the first row to one, the first element of the second row to two and all other elements to zero.

When all the elements are initialized to zero, the following short cut method may be used.

int m[3][5] = { 0, 0 ,0};

or

int m[3][5] = { {0} , {0} ,{0} };

```
//program30 : addition of two matrix
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j,n,m,a[10][10],b[10][10],m1,n1;
    clrscr();

    printf("Enter No of Rows for first matrix:");
    scanf("%d",&n);

    printf("Enter No of columns for first Matrix:");
    scanf("%d",&m);

    printf("Enter No of Rows for second matrix:");
    scanf("%d",&n1);

    printf("Enter No of columns for second Matrix:");
    scanf("%d",&m1);
```

```
if(m!=m1 || n!=n1)
      printf("Sum not Possible.");

else
{
        for(i=0;i<n;i++)
        {
                for(j=0;j<m;j++)
                {
                        printf("Give value of a[%d][%d]:",i+1,j+1);
                        scanf("%d",&a[i][j]);
                }
        }

        for(i=0;i<n;i++)
        {
                for(j=0;j<m;j++)
                {
                        printf("Give value of b[%d][%d]:",i+1,j+1);
                         scanf("%d",&b[i][j]);
                }
        }
```

```
printf("First Matrix:\n");
        for(i=0;i<n;i++)
        {
                for(j=0;j<m;j++)
                {
                        printf("%d ",a[i][j]);
                }
      printf("\n");

        }
printf("Second Matrix\n");
        for(i=0;i<n1;i++)
        {
                for(j=0;j<m1;j++)
                {
                        printf("%d ",b[i][j]);
                 }
      printf("\n");
        }
    printf("After Sum New Matrix\n");

        for(i=0;i<n1;i++)
        {
                for(j=0;j<m1;j++)
                {
                        printf("%d ",a[i][j]+b[i][j]);
                }
      printf("\n");
        }
   }
   getch();
}
```

MULTIDIMENSIONAL – ARRAY :

C allows array of three of more dimensions.

Syntax:

type array_name [s1] [s2] [s3]……..[sn] ;

Where s_i is the size of the i^{th} dimensions.

Some ex: int survey [3] [5] [12] ;
float table [5] [4] [5] [3] ;

Survey is a three dimensional array while table is four dimensional array.

Survey contain 180 integer value. Table contain 300 element for floating type. In this first index denotes year, second city and third month.

The array survey may represent a survey data of rainfall during the last three years from January to December in five cities.

Ex:

int survey[2][3][10]

The array survey Represent a survey of rainfall in the month of October during the second year in city-3.

YEAR 1

Month / City	1	2	.....	12
1				
⋮				
5				

YEAR 2

Month / City	1	2	.....	12
1				
⋮				
5				

YEAR 3

Dynamic Array:

In C it is possible to allocate memory at the run time. This feature is known as Dynamic Memory Allocation and the arrays created at run time are called dynamic arrays.

Dynamic arrays are created using what are known as pointer variable and memory management function like malloc, calloc and realloc.

The process of allocating memory at compile time is known as static memory allocation and the array that receive static memory allocation is called the static array.

Chapter : 8 Handling of character String

What is a String? :

- String is a group of character array.

- Any group of character defined between double quotation mark is a constant string.

  Ex: printf(“Hello How are you”);

  output : Hello How are you

- If you want to included double quotes with string than also it is possible.

  Ex: printf(“\”Hello How are you”\”);

  output : “Hello How are you”

- The common operations performed on character strings are listed below :

1) Reading and writing strings
2) Combining two strings together
3) Copying one string to another
4) Comparing two strings
5) Extracting a portion of a string

❑ Declaring and Initialising String Variable :

- ✓ A String Variable is always declared as an array.

  Syntax : char string_name[size];

- ✓ Where size determines the number of character in the string.

- ✓ size should be maximum number of the character + 1.
  because When the compiler assigns the character string to character array. It automatically supplies null character (“\0”) at the end of the string.

  Ex: char city[10];
  char name[30];

❑ Initialization of string variable :

- ✓ char array may be initialized when they are declared.
- ✓ you can initialized char array with two form:
- ✓ Ex:-char city[9] = “New York”;

  char city[9] = {‘N’, ‘e’, ‘w’, ‘ ’, ‘Y’, ‘o’, ‘r’, ‘k’,’\0’ };

  (**Note:** when we initialize character array by listing its elements, we must supply explicitly the null terminator)
- ✓ C also permit us to initialize character array without specifying the number of elements.
- ✓ In such cases, the size of the array will be determined automatically, based on the number of the elements initialized.
- ✓ Ex: char string[] = {‘G’,’O’,’O’,’D’,’\0’};
- ✓ It defines the string as five elements array.

❑ Reading String from terminal :

- ✓ You can read the string using scanf function with %s format specification.

  Ex: char address[15];

  scanf("%s",address);

  (& sign is not included before address)

- ✓ The problem with the scanf function is that it terminates its input on the first white space it finds.

- ✓ Ex: If input string is New York in address that address array included only New after new string will be terminated.

- ✓ If we want to to read the entire line "New York" then we may use two character array of appropriate sizes. That is,

  scanf("%s %s",adr1,adr2);

  Where adr1 assign "New" and adr2 assigns "York".

- **Program for Reading a series of words from a terminal using scanf() :**

```
#include<stdio.h>
 void main()
{
   char word1[40],word2[40];
   printf("enter text-→\n");
   scanf("%s %s", word1,word2);
   printf("word1=%s\n  word2=%s", word1,word2);
   getch();
}
```

```
Output : enter text-→
         kunal
         dcs
         word1=kunal
         word2=dcs
```

❑ Using getchar() and gets() functions :

- ✓ getchar() is used to get single character from the console

```
Ex: char ch;
    ch=getchar();
```

- ✓ Both functions are in the <stdio.h> header file.
- ✓ gets() is used to get string from the console.

Syntax : gets(str);

```
Ex: char line[20];
    gets(line);
    printf("%s",line);
```

❑ Reading a line of Text :

Program to read a line of text from terminal :

```
#include<stdio.h>
void main()
{
  char line[81],character;
  int c=0;
  printf("Enter string-→\n");
  do
  {
      character = getchar();
      line[c] = character;
      c++;
  } while(character != '\n');

  c=c-1;
  line[c] = '\0' ;
  printf("\n %s \n",line);
getch();
}
```

Output : Enter string-→
welcome
welcome

❑ Copying one string to another string :

- ✓ We can copy one string to another string using character by character basis.
- ✓ We can't assign one string to another string directly.

Ex: string2 = "ABC";
string1 = string2;

It is not possible in C.

- **program to copying one string to another string :**

```
#include<stdio.h>
void main()
{
   char str1[80],str2[80];
   int i;
   printf("Enter the String\n");
   scanf("%s",str2);

   for(i=0; str2[i] != '\0'; i++)
      str1[i] = str2[i];

   str1[i] = '\0';
   printf("%s\n",str1);
   printf("No.of characters=%d\n",i);
getch();
}
```

❑ Writing string to screen :

We have used printf() function with %s format to print the string to the screen.

Ex: printf("%s",name);

Used to display the entire contents of the array name.

❑ Features of the %s format specification :

(1) When the field width is less than the length of the string than the entire string is printed.

(2) The integer value on the right side of the decimal point specifies the number of character to be printed.
Ex: 10.4%s specifies 4 character printed .

(3) When the number of character to be printed is specified by zero, nothing is printed.
Ex: 10.0%s

(4) The minus sign in the specification causes the string to be printed left side. Ex: -10.4%s 4 character printed left side.

(5) the specification %.ns prints the first n characters of the string.

- Another nice features for printing output is:
  printf("%*.*s\n",w,d,string);
  prints the first d characters of the string in the field width of the w.

- **Write the program for writing string using %s format :**

```
void main()
{
    char country[15] = “United Kingdom”
    printf(“%15s \n”,country);
    printf(“%5s \n”,country);
    printf(“%15.6s \n”,country);
    printf(“%-15.6s \n”,country);
    printf(“%15.0s \n”,country);
    printf(“%0.3s \n”,country);
getch();
}
```

Output :

```
 United Kingdom
United Kingdom
          United
United

Uni
```

- **Write the program to print the output:**

```
void main()
{
   int c , d;
   char string[] = “Cprogramming”;
   for(c=0;c<=11;c++)
   {
        d=c+1;
        printf(“%-12.*s \n”,d,string);
   }
    printf(“\n\n”);
   for(c=11;c>=0;c--)
   {
        d=c+1;
        printf(“%-12.*s \n”,d,string);
   }
 getch();
}
```

- **OUT PUT :**

```
C
Cp
Cpr
Cpro
Cprog
……..……..
Cprogramming

Cprogramming
Cprogrammin
Cprogrammi
Cprogramm
Cprogram
Cprogra
Cprogr
……..
….
C
```

Using : putchar() and puts() functions :

- ✓ putchar() is used to put or print single character on the console

```
Ex: char ch='A';
    putchar(ch);
```

- ✓ This statement is equivalent to below printf() statement.

```
Printf("%c",ch);
```

- ✓ Both functions are in the <stdio.h> header file.

- ✓ puts() is used to put string on the console.

Syntax : puts(str);

```
Ex: char line[20];
    gets(line);
    puts(line);    or    printf("%s",line);
```

❑ Arithmetic operation on characters :

- ✓ To write a character in integer representation , we may write it as an integer.

  Ex: int x;

  x = ‘a’;

  printf(“%d\n”,x); it will display number 97 on the screen.

- ✓ It is also possible to perform arithmetic operation on the character constants and variable.

  Ex: x = ‘z’ – 1

- ✓ ASCII value of z is 122 so x will be store 121.
- ✓ The C library supports a function that converts a string of digits into their integer value.

  Syntax: x=atoi(string);

  Ex. number=“1998”;

  year=atoi(number);

❑ String Handling function :

There are mainly four types of string handling function:

1. strcat() – Concatenation of two string
2. strcmp() – Compares two string
3. strcpy() – copies one string over another string
4. strlen() – finds the length of the string.

❑ Strcat() : -

- ✓ Using this function you can concate two string:
  syntax : strcat(string1,strin2);

  where string1 and string2 both are character array.

- ✓ string2 is appended to string1 by removing null character at the end of string1.
  Ex: str1 = “Very \0”;
  str2 = “Good\0”;
  str3 = “Bad\0”;

- ✓ Strcat(str1,str2); - output : Very Good\0

- ✓ It also allows to concat three string together
  Ex: strcat(strcat(str1,str2),str3);
  output : Very GoodBad

❑ strcmp() :

- ✓ This function compares two string. If both are equal then it return 0 value and if they are not equal then it will return numeric difference between the first nonmatching characters in the strings.

- ✓ Ex: strcmp(“their”,”there”);
- ✓ will return a value of -9 which is the numeric difference between ASCII “i” and ASCII ”r”.

- **strcpy () :**

✓ using this function we can copy contents of one string to contents of another string.

syntax : strcpy(str1,str2);

- **strlen() :**

✓ This function is used to find the length of the string.

n = strlen(string);

where n is a integer variable.

- **Examples of string handling functions :**

```
#include<stdio.h>
#include<string.h>
 void main()
{
  char s1[20],s2[20],s3[20];
  int x,l1,l2,l3;
  printf("Enter two string constants\n");
  scanf("%s %s",s1,s2);
  x=strcmp(s1,s2);
 if(x != 0)
  {
   printf("Strings are not equal \n");
   strcat(s1,s2);
   }
  else
  printf("strings are equal\n");
  strcpy(s3,s1);
  l1 = strlen(s1); l2=strlen(s2) ; l3 = strlen(s3);
  printf("\n s1 = %s \t length=%d characters\n",s1,l1);
  printf("\n s2 = %s \t length=%d characters\n",s2,l2);
  printf("\n s3 = %s \t length=%d characters\n",s3,l3);
getch();
 }
```

❑ Table of string :

- ✓ When you want to store a set of strings in table form than it is possible using two dimensional character array.

  Ex: student[30][10];

- ✓ A character array student[30][15] may be used to store a list of 30 names, each of the length not more than 15 characters.

- **Write a program that would sort a list of names in alphabetical order.**

```
#define ITEMS 5
#define MAXCHAR 20
void main()
{
 char string[ITEMS][MAXCHAR],dummy[MAXCHAR];
 int i=0,j=0;
 printf("Enter names:");
 while(i<ITEMS)
   scanf("%s",string[i++]);
 for(i=1;i<ITEMS;i++)
 {
     for(j=1;j<=ITEMS-i;j++)
     {
        if(strcmp(string[j-1],string[j])>0)
         {
            strcpy(dummy,string[j-1]);
            strcpy(string[j-1],string[j]);
            strcpy(string[j],dummy);
         }
}
Printf("\n Alphabetical order");
for(i=0;i<ITEMS;i++)
   printf("%s",string[i]);
getch();
}
```

- **Write the program to count the number of characters,words and spaces in line :**

```
#include<stdio.h>
void main()
{
char line[81],ctr;
int i,c,end=0,characters=0,words=0,lines=0;
while(end==0)
{
  c=0;
   while((ctr=getchar()) != '\n')
        line[c++] = ctr;
   line[c] = '\0';
   if(line[0]=='\0')
      break;
   else
    {
      words++;
       for(i=0;line[i] != '\0';i++)
            if(line[i] == ' ' || line[i] == '\t')
              words++;
   }
```

```
/* counting lines and characters */
        lines = lines +1;
       characters = characters + strlen(line);
   }
     printf("Number of lines = %d\n",lines);
     printf("Number of words = %d\n",words);
     printf("Number of character = %d",characters);
 getch();
}
```

Chapter - 9 User Defined Function

- C Functions can be classified into two category :
  1. Library function.
  2. User defined function.

- Difference between library and user defined function : library function is inbuilt function (not required to be written by us) while user defined function is developed by user at the time of writing a program.

- Ex : printf() and scanf() are library function while main() is a user defined function.

❑ Need for user defined function :

- ✓ main is a user defined function. It is possible to write entire code in main function but it leads a number of problems.
- ✓ The program becomes to large and complex and so the task of maintaining, debugging and testing is so become difficult.
- ✓ If a program is divided into function parts then each parts may be individually coded and finally combined into single units. These subprogram is called function.
- ✓ It is easy to understand, debug and test the program code.

- **Benefits of function:**

1) It facilitates top - down modular programming.

2) The length of source code is reduced by using function at appropriate place.

3) It is easy to locate and isolate faulty function for further investigation.

4) A function can be used by many other programs.

❑ Elements Of User defined function :

✓ Function has three elements :

1)function definition :

The function definition is an independent program module that is specially written to implement the requirements of the functions.

2)function call :

In order to use the function, we need to invoke it at a required place in the program. This is known as the function call. The program that calls the function is referred to as the calling program or calling function.

3)function declaration :

The calling program should declare any function that is to be used later in the program. This is known as the function declaration or function prototype.

❑ Definition of functions :

✓ A function definition also known as function implementation shall include the following elements :

1) function name : it should be valid identifier name.
2) function type : it specifies the return type value(int or float etc.) of function.
3) list of parameters : the parameter list declares the variable that will receive the data sent by the calling program.
4) local variable declarations : it specify the local variable that is needed by the function.
5) function statements : it performs the task of the function.
6) return statement : it returns the value evaluated by the function.

✓ These six elements are grouped into two parts :

1) function header (first three elements)
2) function body (second three elements)

- **Forms (Syntax) of function :**

```
function_return_type function_name(argument list)
{
  local variable declaration;
  executable statemets;
  ……..……..
  ……..……..
  return statement;
}
```

- **Example :**

```
int sum(int a, int b)
{
  int c=a+b;
  return(c);
}
```

❑ Program of UDF:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  void printline();          // prototype
  printline();      // calling function
  printf("This is a function program\n");
  printline();
  getch();
}
  void printline() //called function
  {
     printf("\n");
     printf("==================");
     printf("\n");
  }
```

❑ Category of the function :

There are five categories of the function :

1. Functions with no arguments and no return value
2. Functions with arguments and no return value.
3. Functions with arguments and one return value
4. Functions with no arguments but return a value
5. Functions that return multiple value

- **Function with no arguments and no return value :**

This function can not contain any arguments and any return value.

Ex:

```
#include<stdio.h>
#include<conio.h>
void mul();
 void main()
{
    mul();
    getch();
}

void mul()
{
     int j,k,mul;
     scanf("%d",&j);
     scanf("%d",&k);
     mul = j * k;
     printf("multiplication of two number=%d",mul);
}
```

- **Function with arguments and no return value :**

This function contains arguments but does not return value.

```
#include<stdio.h>
#include<conio.h>
void mul(int,int);
void main()
{
   int j,k;
  clrscr();
  scanf("%d %d",&j,&k);
  mul(j,k);
  getch();
}
void mul(int l,int m)
{
   int mul;
   mul = l * m;
   printf("multiplication of two number=%d",mul);
}
```

- **Function with arguments and return value :**
  This function contains arguments and also return value.

```
#include<stdio.h>
#include<conio.h>
 int mul(int,int);

void main()
{

  int j,k,multi;
  clrscr();
  scanf("%d %d",&j,&k);
  multi=mul(j,k);
  printf("%d",multi);
  getch();
}
int mul(int l,int m)
{
    int mul;
    mul = l * m;
    return(mul);
}
```

❑ Function with no arguments and return value :

This function does not contain arguments and return value.

```
#include<stdio.h>
#include<conio.h>
 int getnumber(void);

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

int getnumber(void)
{
    int number;
    printf("enter number→");
    scanf("%d",&number);
    return(number);
}
```

❑ Function that return multiple value :

This function returns multiple values.

```
#include<stdio.h>
#include<conio.h>
 void mathoperation(int x, int y,int *s,int *d);

void main()
{
  int x=20,y=10,s,d;
  mathoperation(x,y,&s,&d);
  printf("s=%d\n d=%d\n",s,d);
  getch();
}

void mathoperation(int a, int b,int *sum,int *diff)
 {
   *sum=a+b;
   *diff=a-b;
 }
```

❑ Recursion :

✓ Recursion is a process where a function calls itself.

Ex:
```
main()
{
 printf("This is an example of recursion  ");
 main();
}
```

output :

This is an example of recursion

This is an example of recursion

……..

Execution will be continue indefinitely.

- **Ex: Write the program to find the factorial of the given no. using the recursion :**

```
#include<stdio.h>
#include<conio.h>
int factorial(int);
void main()
{
    int no,fact;
    printf("Enter the number :");
    scanf("%d",&no);
    fact=factorial(no);
    printf("Factorial of the given no is:");
    printf("%d",fact);
    getch();
}
int factorial(int no)
{
    int fact;
    if(no==1)
        return(1);
    else
    {
        fact=no*factorial(no-1);
        return(fact);
    }
}
```

- **Output :**

  Enter No : 3

  Factorial of the given no is: 6

- **Analysis :**

  fact=3*factorial(2)

  = 3*2*factorial(1)

  = 3*2*1

  =6

❑ **Function with arrays :**

✓ Like the values of simple variables, it is also possible to pass the values of an array to a function.

Syntax : functionname(arrayname,size)

Ex : float largest(float a[],int n);

✓ Where largest is a function name, a is a array name and n is a size of an array.

- **Example of function with array for Find the maximum no:**

```
#include<stdio.h>
#include<conio.h>
float largest(float a[],int n);
Void main()
 {
    float value[4]={2.5,-4.75,1.2,3.67}
    printf("%f\n",largest(value,4));
    getch();
 }
float largest(float a[],int n)
 {
    int i ;
    float max;
    max = a[0];
     for(i=1;i<n;i++)
      {
         if(max < a[i])
                  max = a[i];
      }
   return(max);
  }
```

❑ Nesting Of function :

✓ C permits nesting of functions freely. Main() can call fun1(),which call fun2() and so on.... There is no limit of functions to be nested deeply.

- **Example :**

```
#include<stdio.h>
#include<conio.h>
void disp1();
void disp2();
void main()
 {
    clrscr();
    disp1();
    getch();
 }
 void disp1()
{
   printf("hello ");
   disp2();
}
void disp2()
{
   printf("DCS");
 }
```

❑ The scope, visibility and lifetime of variables :

- ✓ In C there are four types of storage classes :
  1. Automatic variables
  2. External variables
  3. Static variables.
  4. Register variables

❖ Automatic variables :

- ✓ Automatic variables declared inside a function in which they are to be utilized. so, automatic variables are private to that function.
- ✓ Automatic variables are referred as a local variable or internal variable.
- ✓ So you can declare this variable in more than one function without any confusion.
- ✓ Ex: main()

```
{
  int number ;
}
```

You can also declared explicitly with auto keyword

Ex: main()

```
{ auto int number ;
}
```

- **External variable :**
- ✓ variables that are both active and alive throughout the entire program is known as external variables. It is also called global variable.
- ✓ unlike local variables global variables can be accessed by any function in the program.
- ✓ Ex:

```
int number;
main()
{
  number=10;
}
fun1()
{
  number=10;
}
fun2()
{
   number=10;
}
```

❖ Static variables:

- ✓ Static variables declared using static keyword.
  Ex: static int number;

- ✓ Static variables are initialized only once during the execution of the program. Static variables are generally used to retain values between function calls.

Ex:

```
void stat(void);
void main()
{
        int i;
        for(i=1;i<=3;i++)
           stat();
}
void stat(void)
{
        static int x=0;
        x++ ;
        printf("%d\n",x);
}
```

- ✓ Output: 1
  2
  3

❖ Register Variables :

- ✓ Register Variables are generally used to store variable in machine's register. so you can access that variable very fast than store in the memory.
- ✓ you can create your register variable using register keyword.
  Ex: register int x;
- ✓ register variables are generally used in looping.

Chapter-10. Structures and Unions

❑ Structure :

- ✓ we can not use an array if we want to represent a collection of data items of different types using a single name.
- ✓ C supports a constructed data type known as structure.
- ✓ Structure is nothing but group of data items of different type with a single name.

❖ **Syntax:-**

```
struct structure name
{
   data type member name;
    ---
};
```

❖ **Define structure :**

```
struct book
{
   char title[20];
   char author[15];
   int pages;
   float prices;
};
```

❖ Declaring structure variables:

- ✓ 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

- ✓ Example :

```
struct book b1,b2,b3;
```

- ✓ You can also defined structure variable at the time of defining the structure.

```
 struct book
{
   char title[20];
   char author[15];
   int pages;
   float prices;
}b1,b2,b3;
```

❖ Accessing structure members:

- ✓ We can access and assign values to the members of the structure.
- ✓ The link between a member and a variable is established using the member operator '.' which is also known as dot operator or period operator.
- ✓ For Example b1.prices=120.50;

❖ Initialization of structure variable :

- First method to initialize structure :

```
main()
{
    struct stud
    {
        int weight;
       float height;
    };
   struct stud stud1 = {60,80.5};
   struct stud stud2 = {53,170};
}
```

- **Second method to initialise structure :**

```
  struct stud
{
    int weight;
   float height;
} stud1={60,80.5};
```

❑ Program for structure :

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

void main()
{
  struct stud
 {
   int rno;
   char name[50];
   char add[50];
   char city[50];
   int ph;
 }s;
 clrscr();
```

```
    printf("Give Roll No:");
    scanf("%d",&s.rno);
    printf("Give Name:");
    scanf("%s",s.name);
    printf("Give Address:");
    scanf("%s",s.add);
    printf("Give City:");
    scanf("%s",s.city);
    printf("Give Phone No:");
    scanf("%d",&s.ph);


    printf("Roll No:%d\n",s.rno);
    printf("Name:%s\n",s.name);
    printf("Address:%s\n",s.add);
    printf("City:%s\n",s.city);
    printf("Phone No:%d\n",s.ph);

  getch();
}
```

❑ Copying and Comparing Structure :

- ✓ two variables of the same structure type can be copied the same way as ordinary variable.
- ✓ If person1 and person2 belong to the same structure then the following statements are valid.

  Ex: person1=person2;

  person2=person1;
- ✓ C does not permit any logical operations on structure variables. In case, we need to compare them we may do so by comparing members individually.

  Ex: person1==person2; is not valid statement.
- ✓ stud1.rno==stud2.rno; is valid statement

❑ Arrays of structure :

- ✓ we may declare an array of structures, each element of the array representing a structure variable.
- ✓ Example :

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int i;
 struct stud
 {
  int rno;
  char name[50];
 }s[3];
 clrscr();
```

```
for(i=0;i<3;i++)
  {
    printf("Give Roll No:");
    scanf("%d",&s[i].rno);
    printf("Give Name:");
    scanf("%s",s[i].name);
}
 for(i=0;i<3;i++)
 {
    printf("Roll No:%d\n",s[i].rno);
    printf("Name:%s\n",s[i].name);
}
 getch();
}
```

❑ Array within structure :

✓ C permits the use of arrays as structure members.

✓ Example :

```
void main()
 {
   struct marks
   {
      int sub[3];
      int total ;
   };
  struct marks student[3] ={45,67,81,0,75,53,69,0,57,36,71,0};

  int i,j;
  for(i=0;i<=2;i++)
  {
     for(j=0;j<=2;j++)
     {
        student[i].total = student[i].total +student[i].sub[j];
     }
  }
```

```
    printf("display student total\n\n")
   for(i=0;i<=2;i++)
       printf("student[%d]     %d",i+1,student[i].total)
    getch();
}
```

❑ Structure within structure :

- ✓ Structure within a structure means nesting of structures.
- ✓ Ex: the following structure defined to store information about the salary of employees.

```
struct salary
{
    char name[20] ;
    char department[20];
    int basic_pay;
    int houserent_allowance;
    int city_allowance ;
    int medical_allowance;
} employee;
```

- ✓ **Now we can group all the items related to allowance together and declare them under a structure.**

```
struct salary
{
      char name[20] ;
      char department[20];
      int basic_pay;
      struct
     {
      int house_rent;
      int city ;
      int medical;
     } allowance;
 } employee;
```

❑ Union :

- ✓ Unions are concepts borrowed from structures therefore follow the same syntax as structures.
- ✓ Using unary operator sizeof() you can know the size of structure.

difference between structure and unions :

- ✓ In structures each member has its own storage locations, whereas all members of the unions use the same storage location. This implies that a union may contain many members at a time but it can handle only one member at a time.
- ✓ In unions the compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.

Ex: Union Example

```
#include<stdio.h>
#include<conio.h>
union item
{
 int m;
 float x;
 }code;
void main()
{
 code.m=10;
 code.x = 15.5;
 printf("%d",code.m);
 printf("%f",code.x);
}
/* Output : 10 15.5 */
```

Chapter-11. POINTERS

What is a pointer ?

- A Pointer is a derived Data Type.
- A pointer is nothing but a variable that contains the address which is a location of the another variable in memory.

Benefits of using the pointer :

1. Pointer are more efficient in handling array and data tables.
2. Pointer reduces the length and complexity of the program.
3. The use of pointer array to character string results in saving of data storage space in memory.
4. Pointer can be used to return multiple values from functions via functions arguments.
5. Pointers permit references to functions.
6. Pointers allow C to support dynamic memory management.
7. Pointers provide an efficient tool for manipulating Dynamic Data Structure such as Linked List , Queue, Stack and Tree.
8. They increase the execution speed.

❑ Accessing The Address of a Variable:

- ✓ The actual location of a variable in the memory is system dependent and therefore the address of a variable is not known to us immediately.
- ✓ We can access the address of a variable using & Operator.

```
int quantity=179;
int *p;
p=&quantity;
```

quantity	variable
179	value
5000	Address

- ✓ p=&quantity; will assign the address the 5000 to the variable p.
- ✓ The & Operator can be used only with simple variable or array element.

❑ Declaring and initializing pointers :

- ✓ syntax : data_type *pt_name ;

- ✓ this tells the compiler three things about the variable pt_name.

  1. * tells the compiler that the variable pt_name is a pointer variable.
  2. pt_name needs a memory location.
  3. pt_name points to a variable of type data_type.

- ✓ Ex: int quantity=179 ;
  int *p;
  p=&quantity;

quantity	variable
179	value
5000	Address

- ✓ p=&quantity; using this statement you can store the address of the variable quantity to the pointer variable p.

- ✓ Note : you can know the address of variable using %u format specification.

- ✓ You can't assign an absolute address to a pointer variable directly.
  Ex: p = 5000; It is not possible

- **Accessing variables using pointers.**

```
void main()
{
        int x,y, *ptr;
        x=10;
        ptr=&x;
        y = *ptr;
        printf("Value of x is %d\n\n",x);
        printf("%d is stored at address %u\n", x, &x);
        printf("%d is stored at address %u\n", *&x, &x);
        printf("%d is stored at address %u\n", *ptr, ptr);
        printf("%d is stored at address %u\n",ptr,&ptr);
        printf("%d is stored at address %u\n", y, &y);

        *ptr=25;
        printf("\n Now x = %d\n",x)
        getch();
}
```

Output:

```
Value of X is 10
10 is stored at address 4104
10 is stored at address 4104
10 is stored at address 4104
4104 is stored at address 4106
10 is stored at address 4108

Now x = 25
```

❑ Pointers expressions:

- ✓ Pointer variables can be used in expressions. For e.g. If P1 and P2 are declared and initialized pointers then the following statements are valid.
- ✓ Y=*p1 * *p2;
- ✓ Sum=sum+*p1;

```
void main()
{
        int a,b,*p1,*p2, x, y, z ;
        a =12;  b = 4;
        p1=&a;   p2 = &b;
        x = *p1 * *p2 – 6;
        y = 3  * - *p2 / *p1 + 10;
        printf("Address of a = %u\n",p1);
        printf("Address of b=%u\n",p2);
        printf("\n");
        Printf("a=%d, b=%d\n",a , b);
        printf("x=%d , y=%d\n",x, y);
        *p2 = *p2 + 3 ; *p1 = *p2 – 5 ; z= *p1 * *p2 – 6 ;
        printf(\n a = %d , b = %d ", a ,b);
        printf("z = %d\n", z);
        getch();
}
```

Out put :-

```
Address of a = 4020
Address of b =4016
a = 12, b=4
x=42, y=9
a=2, b=7, z=8
```

❑ Pointer increments and scale factor :

- ✓ p1++ will cause the pointer p1 points to the next value of its type.

  Ex :- if p1 is an integer pointer with the initial value say 2800,then after the operations p1=p1+1,the value of p1 will be 2802,not 2801

- ✓ when we increment pointer its value is increased by the length of the data type that it points to. This length is called scale factor.

❑ Pointer and Arrays :-

- ✓ When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in memory location.

- ✓ Base address is the location of the first element of the array
  `int x[5] = {1,2,3,4,5};`

- ✓ Suppose the base address of x is 1000 and assuming that Each integer requires 2 bytes, the five elements stored as Follows :

Element	x[0]	x[1]	x[2]	x[3]	x[4]
Value	1	2	3	4	5
Address	1000	1002	1004	1006	1008

So, base address is,
`x = &x[0] = 1000`

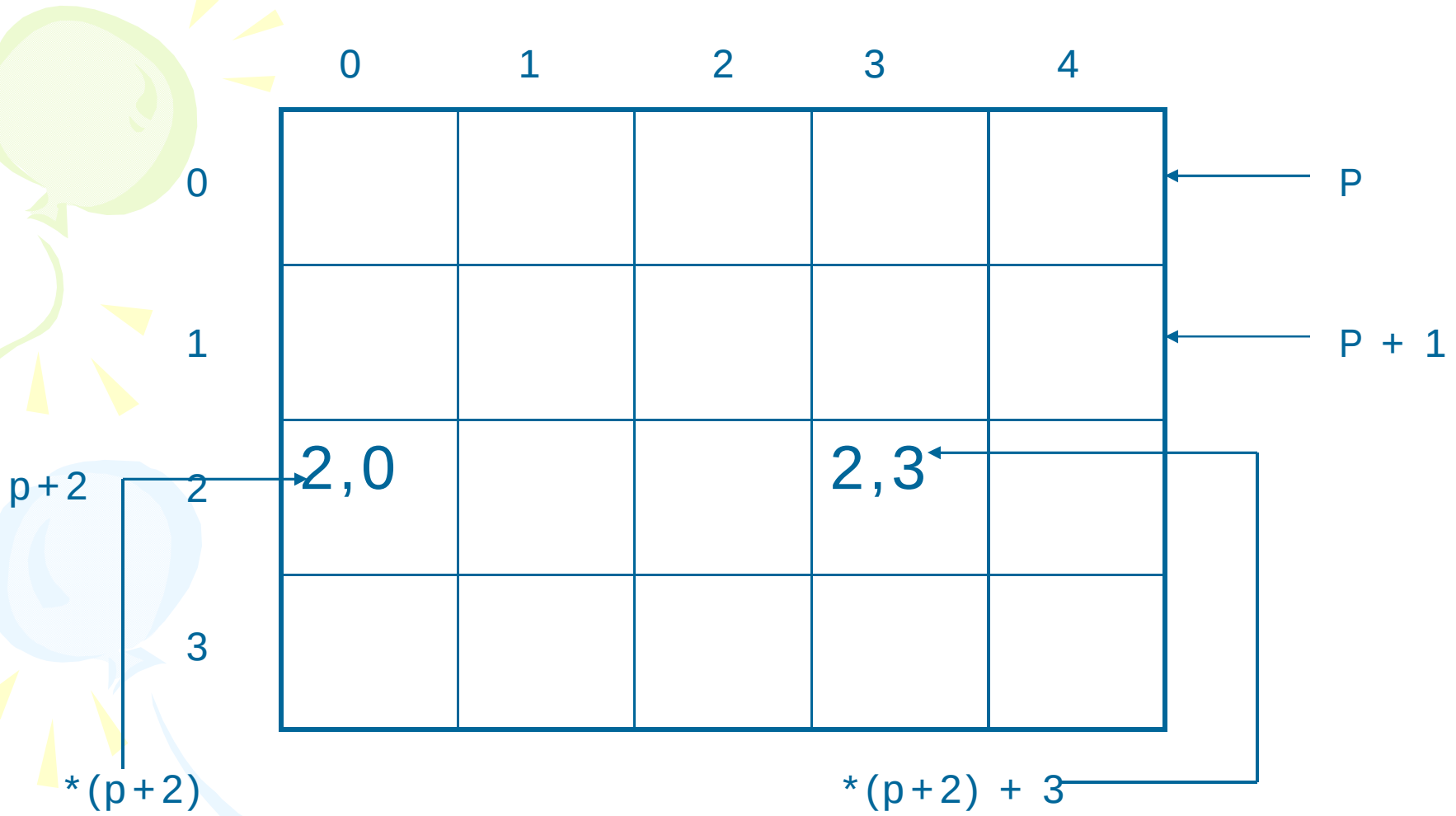
- If we declare p as integer pointer, then we can make the pointer p to point to the array x by the following statements.

  p = x or p = &x[0]

  Now we can access every element of x using p++ to move from one element to another.

  p = &x[0] (=1000)
  p+1 = &x[1] (=1002)
  p+2 = &x[2] (=1004)
  p+3 = &x[3] (=1006)
  p+4 = &x[4] (=1008)

```
/***PRGRAMM FOR SUM OF N NUMBER ELEMENT *****/
#include<stdio.h>
main()
{
   int x[10],i,*p,n,sum=0;
   printf("enter the N:");
   scanf("%d",&n);
   printf("enter the the data:\n");
   for(i=0;i<n;i++)
   {
   scanf("%d",&x[i]);
   }
   p = x;
   for(i=0;i<n;i++)
   {
   printf("\n%d",*p);
   sum=sum + *p;
   p++;
   }
   printf("\nsum=%d",sum);
   getch();
}
```



P = pointer to first row

P+I = pointer to ith row

***(p+i) = pointer to first element in the ith row**

***(p+i)+j = pointer to jth elements**

***(*(p+i)+j)=value stored in the cell(i,j)**

```
/*** PROGRAMM FOR SUM OF N NUBER ELEMENT*****/
#include<stdio.h>
main()
{
    int x[10][10],i,j,n,sum=0;
    clrscr();
    printf("enter the N:");
    scanf("%d",&n);
    printf("enter the the data:\n");
    for(i=0;i<n;i++)
    {
       for(j=0;j<n;j++)
       {
       scanf("%d",&x[i][j]);
       }
    }
  for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
    printf("\n%d",*(*(x+i)+j));
    sum=sum + *(*(x+i)+j);
    }
}
    printf("\nsum=%d",sum);
     getch();
 }
```

❑ Pointer and character string:

- ✓ Strings are treated like character arrays and therefore, they are declared and initialized as follows:
- ✓ `char str[5] = “good”;`
- ✓ C supports an alternative method to create string using pointer variable of type char.

  `char *str = “good”;`
- ✓ The pointer str now points to the first character of the string “good”.

❑ Array of Pointers:

- ✓ One important use of pointer is handling of a table of string. following is the array of string.

  `char name[3][25];`
- ✓ This say that table containing three names ,each with a maximum length of 25 characters. So total storage requirement for the name table are 75 bytes

- We know that individual string may be unequal length. therefore instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length.

```
char *name[3] = {
                    "New Zealand",
                        "Australia",
                        "India"
                    }
```

- Where name to be an array of three pointers to a character, each pointer points to a particular name.

```
name[0]="New Zealand",
name[1]="Australia",
name[2]="India"
```

- This declaration allocates only 28 bytes.

- The character arrays with the rows of varying length are called **ragged array**.

/****PROGRAMM FOR PRINT STRING USING POINTER ****/

```
#include<stdio.h>
main()
{
   char name[10],*ptr;
   clrscr();
   printf("enter the name  => ");
   scanf("%s",name);
   ptr=name;
   printf("\n");
   while(*ptr != '\0')
   {
   printf("%c",*ptr);
   ptr++;
   }
   getch();
}
```

Note : if we want to directly assign string constant then we have to make name as characte pointer.

char *name ; name = “Delhi”

❑ Pointers As Function Arguments:

- ✓ When we pass addresses to a function, the parameter receiving the addresses should be pointers.

- ✓ The process of calling a function using pointer to pass the address of variable is known as call by reference. The function which is called by 'reference' can change the value of the variable used in the call.
- ✓ The process of passing the actual value of variable is known as call by value.
- ✓ Ex:

```
main()
{
        int x=20;;
        change(&x);
        printf("%d\n",x);
}
change(int *p)
{
        *p = *p + 10;
}
```

- ✓ Thus call by reference provides the mechanism by which the function can change the stored value.

Ex: /***PROGRAMM FOR INCHANGING THE VALUE*****/

```
#include<stdio.h>
main()
{
int a,b;
printf("enter the number");
scanf("%d %d",&a,&b);
printf("\t\t\nA=%d B=%d\n\n",a,b);
swap(&a,&b);
printf("\t\t\nA=%d B=%d",a,b);
getch();
}
swap(int *p ,int *q)
{
int t;
t=*p;
*p=*q;
*q=t;
}
```

Call by Value :

If data is passed by value, the data is copied from the variable used in for example main() to a variable used by the function. So if the data passed is modified inside the function, the value is only changed in the variable used inside the function.

Call by value example:

```
#include <stdio.h>

void call_by_value(int x)
{
    printf("Inside call_by_value x = %d before adding 10.\n", x);
    x += 10;
    printf("Inside call_by_value x = %d after adding 10.\n", x);
}
```

```
int main()
{
   int a=10;
   printf("a = %d before function call_by_value.\n", a);
   call_by_value(a);
   printf("a = %d after function call_by_value.\n", a);
   return 0;
}
```

The output of this call by value code example will look like this:

```
a = 10 before function call_by_value.
Inside call_by_value x = 10 before adding 10.
Inside call_by_value x = 20 after adding 10.
a = 10 after function call_by_value.
```

❑ Call by Reference :

If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in main().

Call by reference example:

```
#include <stdio.h>

void call_by_reference(int *y)
{
    printf("Inside call_by_reference y = %d before adding 10.\n",
    *y);
   (*y) += 10;
   printf("Inside call_by_reference y = %d after adding 10.\n", *y);
}
```

```
int main()
 {
    int b=10;
    printf("b = %d before function call_by_reference.\n", b);
    call_by_reference(&b);
    printf("b = %d after function call_by_reference.\n", b);
return 0;
}
```

The output of this call by reference code example will look like this:

```
b = 10 before function call_by_reference.
Inside call_by_reference y = 10 before adding 10.
Inside call_by_reference y = 20 after adding 10.
b = 20 after function call_by_reference.
```

❑ POINTERS AND STRUCTURES:

- ✓ We know that name of an array stands for address of its zeroth element. The same thing is true of the names of arrays of structure variable.
- ✓ Ex:

```
struct inventory
{
  char name[30];
  int number;
  float price;
} product[2],*ptr;
```

- ✓ here product as an array of two elements and ptr as pointer to data objects of the type struct inventory.

```
ptr = product ;
```

- ✓ would assign the address of the zeroth element of product to ptr. That is the pointer ptr points to the product[0].It's members can be accessed using the following notation.

- ptr ->name
  ptr->number
  ptr->price

  -> : this symbol is called arrow operator and is made up of minus sign and greater than sign.

- When pointer ptr is incremented by one. It is made to point to the next record i.e. product[1].

- We can also use notation (*ptr).number to access the member number. Parentheses around *ptr are necessary because the member operator "." has a higher precedence than the operator *.
- Note : precedence is necessary in structure.
  ex: ++ptr -> count : increment count not ptr
  (++ptr)->count :- increment ptr first and then link count

❑ pointer to structure variable :

✓ **Example :**

```
struct invent
{
        char *name[20];
        int number;
        float price;
} product[3] , *ptr ;

void main()
{
clrscr();
printf("Enter Data for Structure members\n");
for(ptr=product ; ptr < product + 3 ; ptr++)
scanf("%s %d %f",ptr->name,&ptr->number,&ptr->price);

ptr=product;

for(ptr=product ; ptr < product + 3 ; ptr++)
printf("%s %d %f",ptr->name,ptr->number,ptr->price);
getch()
}
```

Chapter-12. File Management in C

- ✓ Using the functions such as scanf() and printf() to read and write data. These are console oriented I/O functions which always use the terminal as the target place.

- ✓ This works fine as long as the data is small. In many real-life problem involve large volume of data and in this situation console oriented operations pose two major problems
  1. It is more time consuming to handle large volumes of data.
  2. The entire data is lost when either the program is terminated.

- ✓ So, it's a more flexible that data can be stored on the disk and read whenever necessary, without destroying the data. This is possible using the concept of file to store data.

- ✓ A file is a place on the disk where a group of related data is stored.

❑ Basic file operations are :

1) naming a file
2) opening a file
3) reading data from file
4) writing data to a file
5) closing a file

❑ DEFINING AND OPENING A FILE :

✓ General format for defining and opening a file:

syntax :

```
FILE *fp ;
fp = fopen("filename","mode");
```

✓ where fp is a pointer variable which points to the data type FILE. FILE is a structure that is defined in the I/O library.

✓ The second statement open the file named filename and assign an identifier to the file type pointer fp.

- **There are three types of mode :**

  r : open the file for reading only

  w: open the file for writing only

  a : open the file for appending data to it.

✓ **When trying to open a file, there are three possibilities:**

1) In write mode : a file with specified name is created if the file does not exist. the contents are deleted if the file are already exist.

2) In append mode : when the purpose is "appending" the file is opened with the current content safe. A file with the specified file name is created if file does not exist.

3) In read mode : if file is exist then open file in read mode otherwise an error occurs.

```
Ex:  FILE *p1 , *p2 ;
     p1 = fopen("data","r");
     p2 = fopen("results","w");
```

✓ **Many recent compilers include additional modes of file operations:**

✓ **r+ : The existing file is open for both reading and writing**

✓ **w+: Same as w except both for both reading and writing**

✓ **a+: Same as a except both for both reading and writing**

❑ Closing a file :

- ✓ Using this function you can close the file.
- ✓ syntax : fclose(file_pointer);

  Ex: fclose(*p1);

  fclose(*p2);

❑ INPUT OUTPUT OPERATION IN FILE :

❖ getc and putc Functions :

- ✓ Using getc function you can read a character from a file that has been opened in read mode.
- ✓ syntax : c = getc(fp);

- ✓ where fp is a file pointer which points to a file which is opened in a read mode
- ✓ And store the character to the variable c. Reading should be terminated when EOF is encountered.

- ✓ using putc function you can write character to a file.

  syntax : putc(c,fp1)
- ✓ where fp1 is a function pointer which points to a file which is opened in write mode to store the character and c is a character variable which stores the character which is write to a file.

- **Example :**

```
#include<stdio.h>
void main()
{
  FILE *f1;
  char c;

  f1 = fopen("INPUT","w");
  while((c=getchar()) != EOF)
        putc(c,f1);
  fclose(f1);

  f1 = fopen("INPUT","r");
  while((c=getc(f1)) != EOF)
   printf("%c",c);
  fclose(f1);
  getch();
}
```

Note :You terminate the entering string by pressing ctrl + Z.

- **getw and putw functions**
  - ✓ They are used to read the integer value from file and to write the integer value to the file.
  - ✓ Syntax : int n = getw(fp);
    putw(n,fp);

```
#include<stdio.h>
Void main()
{
   FILE *f1;
   int no , i;

   f1 = fopen("INPUT","w");
   for(i=0 ; i < 10 ; i++)
  {
        scanf("%d",&no);
        if(no == -1)
            break;
        putw(no,f1);
  }
  fclose(f1);

   f1 = fopen("INPUT","r");
   while((no=getw(f1)) != EOF)
     printf("%d",no);
   fclose(f1);
  getch();
}
```

```
/****PROGRAMM FOR TO READ AND WRITE FROM FILE***********/
#include<stdio.h>
main()
{
FILE *f,*f1,*f2;
int c,i;
clrscr();
printf("ENTER  THE DATA  ====  >");

   f=fopen("input","w");
    for(i=0;i<50;i++)
    {
    scanf("%d",&c);
    if(c==999)
    break;
    putw(c,f);
    }
   fclose(f);

    f=fopen("input","r");
    f1=fopen("even","w");
    f2=fopen("odd","w");
```

```
    while(!feof(f))
    {
      c=getw(f);
      printf("%d",c);
      if(c%2==0)
        putw(c,f1);
      else
        putw(c,f2);
    }
   fclose(f);
   fclose(f1);
   fclose(f2);

  f1=fopen("odd","r");
  f2=fopen("even","r");
  printf("GIVE DATA IS ODD\n");
    while(!feof(f1))
    {
      c=getw(f1);
      printf("%d",c);
    }
    printf("GIVE DATA IS EVEN\n");
    while(!feof(f2))
    {
       c=getw(f2);
       printf("%d",c);
    }
    fclose(f1);
    fclose(f2);
    getch();
}
```

❑ The fprintf and fscanf function :

- ✓ The functions fprintf and fscanf are identical to the printf and scanf function.
- ✓ fscanf and fprintf functions are generally used to perform input and output operation.
- ✓ syntax for fprintf :

  fprintf(fp,“control string”,list)

- ✓ where fp is a file pointer associated with the file that has been opened for writing. The control string contains output specifications for the items in the list. The list may include variables , constants and strings.

  Ex: fprintf(f1,”%s %d %f”, name,age,7.5);

- ✓ fscanf functions are generally used to read the data from the file.
- ✓ syntax for fscanf :

  fscanf(fp,“control string”,list)

- ✓ where fp is a file pointer associated with the file that has been opened for reading. The control string contains input specifications for the items in the list. The list may include variables , constants and strings.

  Ex: fscanf(f2,”%s %d”, item , &quantity);

❑ Error handling during I/O operation :

✓ It is possible that an error may occur during I/O operation on a file. typical error situation include :

1. Trying to read beyond the end – of – file mark.
2. Trying to use a file that has not been opened.
3. Trying to perform an operation on a file, when the file opened for another type of operation.
4. Opening a file with an invalid filename.
5. Attempting to write to a write – protected file.
6. Device overflow

✓ This type of errors are handles during the I/O operations.

✓ The feof function can be used to test for an end of file condition.

```
if(feof(fp))
    printf("End of data \n");
```

- ✓ Where fp is file pointer it pass as a arguments to the feof function.

- ✓ The ferror function is used to check that an error has occurred or not during the reading processing.

  Ex: if(ferror(fp) != 0)

  printf("An error has occurred\n");

- ✓ When the file cannot be opened for some reason, then the function returns a null pointer. This facility can be used to test whether a file has been opened or not.

  Ex: if(fp == NULL)

  printf("File could not be opened");

- **Example:**

```
#include<stdio.h>
void main()
{
      char *filename;
      FILE *fp1,*fp2;
      int  i , number;
      fp1 = fopen(“TEST”,”w”);
      for(i=0;i<=100;i+=10)
          putw(i , fp1);
      fclose(fp1);
      printf(“Input file name \n”);
      open_file:
      scanf(“%s”,filename);
      if((fp2 = fopen(filename,”r”)) == NULL)
      {
         printf(“can not open the file Try again\n ”);
         goto open_file;
      }
```

```
    else
    for(i=1;i<=20;i++)
    {
          number = getw(fp2);
          if(feof(fp2))
          {
             printf("\n Range out of data.\n");
             break;
          }
          else
           printf(" %d \n", number);
    }
    fclose(fp2);
}
```

❑ Random access to a files :

✓ Using ftell, rewind and fseek functions we can randomly access the files.

- **ftell function:**

✓ Using ftell function we can know the current position of the file.
syntax : n = ftell(fp);

✓ where fp = filepointer , n would give the relative offset (in bytes) of the current position. This means that n bytes have already been read (or written)

- **rewind function:**

✓ using this function we can resets the position of the pointer to the start of the file.
syntax : rewind(fp);
n=ftell(fp);

✓ where fp = file pointer. When second statements are executed it would assign 0 to n because file position has been set to the start of the file by rewind.

✓ This functions helps us in reading a file more than once , without having close and open the file

Note : whenever file is open for reading or writing , rewind is done automatically

- **fseek function:**

✓ fseek function is used to move the file position to a desired location.

syntax : fseek(fileptr,offset,position);

✓ where fileptr is a pointer to the file concerned , offset is a number or variable of type of long.

✓ The offset specifies the number of positions to be moved from the location specified by position.

✓ The position take one of the following three form :

value	meanings
0	Beginning of the file
1	current position
2	End of the file

✓ The offset may be positive, meaning move forwards or negative meanings move backwards.

- ✓ the following examples illustrate the operation of the fseek function :

Statement	Meaning
fseek(fp,0L,0)	Go to the Beginning
fseek(fp,0L,1)	stay at current position
fseek(fp,0L,2)	Go to the end of the file.
fseek(fp,m,0)	Move to (m+1)th byte in the file
fseek(fp,m,1)	Go forward by m bytes
fseek(fp,-m,1)	Go backward m bytes from current position.
fseek(fp,-m,2)	Go backward by m bytes from the end.

- ✓ When operation is successful, fseek returns a 0 otherwise returns -1.

- **Example:**

```
#include<stdio.h>
void main()
{
        FILE *fp;
        long n;
        char c;
        fp = fopen("RANDOM","w");
        while((c=getchar()) != EOF)
            putc(c,fp);
        printf("No.of character entered=%ld",ftell(fp));
        fclose(fp);

        fp = fopen("RANDOM","r")
        n = 0L;
        while(feof(fp)==0)
        {
        fseek(fp,n,0)   /* position to (n+1)th character */
        printf("Position of %c is %ld\n",getc(fp),ftell(fp));
        n=n+5L;
        }
getch();
}
```

Output :

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ^Z
No. of character entered = 26
Position of A is  0
Position of F is  5
Position of K is 10
Position of P is 15
Position of U is 20
Position of Z is 25
Position of    is 30
```

When printing current position 30 the loop is terminating.

❑ Command line arguments :

- ✓ Command line arguments is a parameter supplied to a program when the program is invoked.

- ✓ Execution of the c program is starting from the main function. main function also contains arguments like another normal function.

- ✓ main function can take two arguments : argc and argv
- ✓ variable argc counts the number of arguments on the command line.

- ✓ argv is an argument vector and represents an array of character pointers that point to the command line arguments. The size of this array will be equal to the value of argc.

- Ex: if we want to execute a program to copy the contents of a file named X_FILE to another one named Y_FILE,then we may use command line like,

```
c:> PROGRAM   X_FILE    Y_FILE
```

- Where PROGRAM is a filename where program execution code is stored.

so, argc ----> 3 arguments
argv[0] ------> program
argv[1] ------> X_FILE
argv[2] ------>Y_FILE

- To access the command line arguments , we must declare main function and its parameters.

```
main(int argc, char *argv[])
{
……….
……….
}
```

❑ **Example:**

```
#include<stdio.h>
#include<conio.h>
void main(int argc , char *argv[])
{
   FILE *fp;
   int i;
   char word[15];
   clrscr();
   fp=fopen(argv[1],"w");
   printf("No.of arguments =%d\n",argc);
   for(i=2;i<argc;i++)
      fprintf(fp,"%s",argv[i]);   /* write contents to file */
   fclose(fp);
```

```
    /* writing contents of the file to the screen */
    printf("contents of the file to the screen\n");
    fp = fopen(argv[1],"r");
    for(i=2;i<argc;i++)
    {
        fscanf(fp,"%s",word);
        printf("%s",word);
    }
    fclose(fp);
    printf("\n\n");
    getch();
}
```

Output:

```
C:> F12_7 TEXT AAAAAA BBBBBB CCCCCC
No. of arguments =5
contents of the file to the screen
AAAAAA BBBBBB CCCCCC
```

Chapter 13 Dynamic Memory Allocation

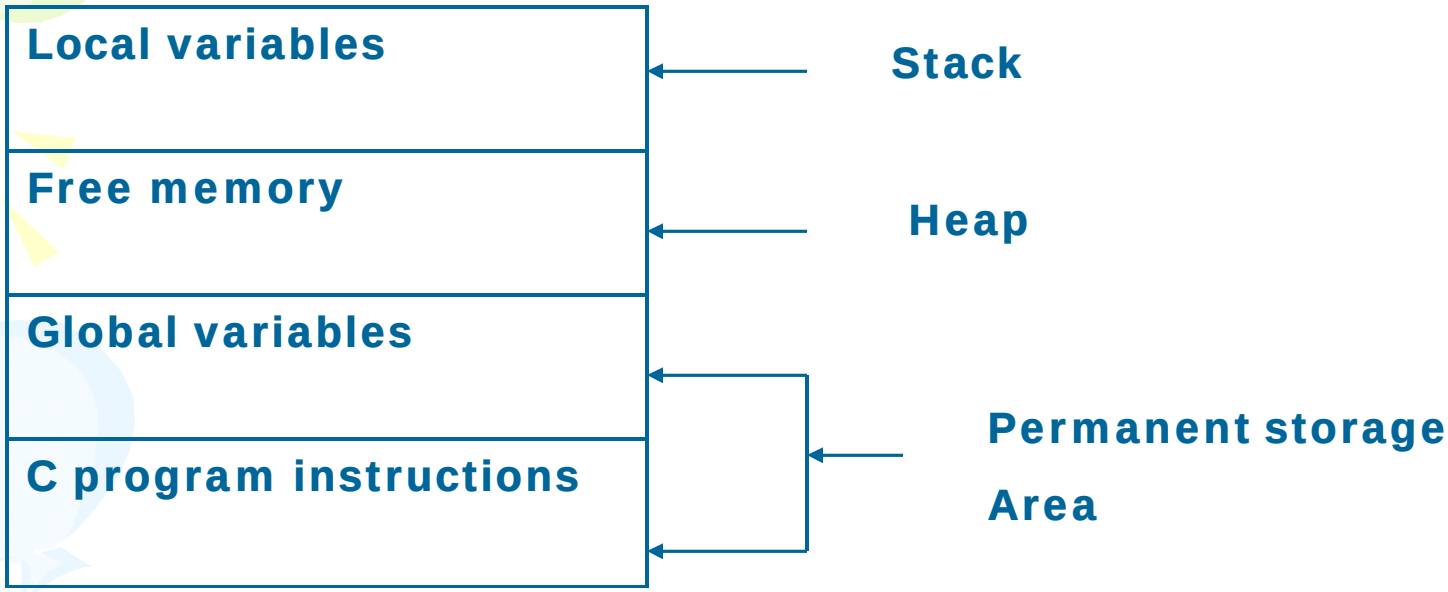
- **Need for DMA?**
  - ✓ Most often we face situations in programming where the data is dynamic in nature.
  - ✓ Ex : A program for processing the list of customers of corporation. The list grows when the names are added and shrinks when names are deleted. When list grows we need to allocate more memory space to the list to accommodate additional data items. such situations can be handled more easily and effectively by using what is known as **dynamic data structures.**

❑ What is DMA ?

- ✓ The process of allocating memory at runtime is known as dynamic memory allocation.
- ✓ There are four memory allocation function :

1. malloc : It allocates requested size of bytes and returns a pointer to the first byte of the allocated space.
2. calloc : Allocates space for an array of elements ,initializes them and then returns a pointer to the memory.
3. free : Frees previously allocated space.
4. realloc : Modifies the size of previously allocated space.

❑ Memory Allocation Process :



- ✓ The memory space that is located between this two region is available for dynamic memory allocation during execution of the program. This free memory region is called heap. The size of the heap keep changing during the execution of the program. Therefore, it is possible to encounter memory overflow problem during dynamic allocation process. In such situation memory allocation function returns a NULL pointer.

❑ malloc :

- ✓ Using this function we can allocate a block of memory. The malloc function reserve a block of memory of specified size and a return a pointer of type **void**. This means that we can assign it to any type of the pointer.

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

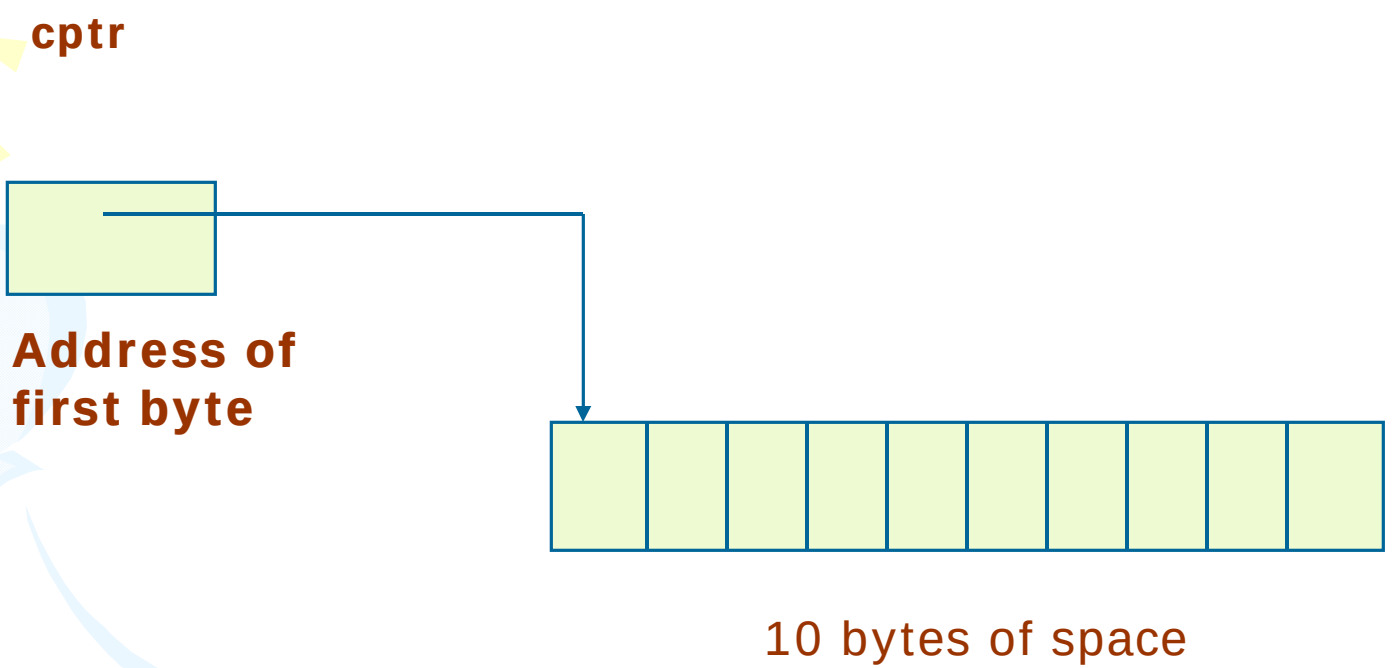
ptr : It is a pointer of the type cast-type.

malloc : It returns a pointer to an area of memeory with size byte-size.

Ex: x = (int *) malloc (100 * sizeof(int));

- ✓ 100 * sizeof(int) memory space is allocated and x points to the first byte of the memory allocated spaces.

- ✓ **cptr = (char *) malloc (10);**
- ✓ **It allocates 10 bytes of space for the pointer cptr of type char.**



- ✓ **Remember , the allocation fail if the space in the heap is not sufficient to satisfy request. If it fails, it returns a NULL. We should therefor check whether the allocation is successful before using the memory pointer.**

- **calloc :**
  - ✓ Using this function we can allocate multiple block of storage and each of the same size and then sets all bytes to zero.
  - ✓ **syntax :** ptr = (cast-type *) calloc (n,elem-size)
  - ✓ The above statements allocates contiguous space for n blocks , each of size elem-size bytes. All bytes are initialized to zero and a pointer returns to the first bytes of the allocated region. If there is not space then it returns NULL pointer
  - ✓ It is generally used for storing derived data – type such as arrays and structures.

- **free :**
  - ✓ Using free function we can releases the block of memory which is created by malloc and calloc function.
  - ✓ **syntax :** free(ptr);
  - ✓ Where ptr is a pointer to a memory block which is created by malloc and calloc.
- **realloc :**
  - ✓ Using this function we can change the already allocated memory – size and this process is called the reallocation process.

Ex: ptr = malloc(size) / * allocates the space * /

ptr = realloc(ptr,newsize) / * reallocates the space * /

Chapter 14: The Preprocessor

What is preprocessor?

- ✓ The preprocessor is a program that processes the source code Before it passes through the compiler.

- ✓ Preprocessor directives:

- ✓ Following are the preprocessor directives:

#define	Define a macro substitution
#undef	Undefined a macro
#include	specifies the file to be included
#ifdef	Test for macro definition
#endif	Specifies the end of if
#ifndef	Tests whether a macro is not defined
#if	Tests a compile time condition
#else	Specifies alternative when #if test fails.

Preprocessor directive follow special syntax rule.

- They all begin with the symbol #.
- They don't require the semicolon at the end.
- Preprocessor directives are placed before the main line.

These directives can be divided into three categories:

- Macro Substitution directives
- File inclusion directives
- Compile control directives

❑ Macro Substitution:

✓ Macro substitution is a process where an identifier in a Program is a replaced by a predefined string composed of one or more tokens.

✓ There are different form of macro substitution.

- Simple macro substitution.
- Argument macro substitution.
- Nested Macro substitution.

- **Simple macro substitution**

✓ **The preprocessor accomplish this task under the direction of the #define statement**

✓ **This statement usually known as a macro definition or simply macro.**

Syntax: **#define identifier string**

Ex:

```
#define count 100
#define False 0
#define PI 3.14
#define AREA 5*12.46
#define SIZE sizeof(int)*4
#define D 45-22
#define A 78+32

#define TEST  if(x>y)
#define AND
#define PRINT printf("Very Good.\n");

TEST AND PRINT
```

The preprocessor translate above line to.

```
If(x>y)
   printf("Very Good.\n");
```

```
#define EQUALS ==
#define AND &&
#define BLANK LINE printf("\n");
#define INCREMENT ++
```

- **Macro with Arguments :**

✓ When you passed arguments to the macro is called the macro with arguments.

Syntax: `#define identifier(f1,f2……fn) string`

Ex:

```
#define CUBE(x) (x*x*x)
```

If the following statements occur in the program.

```
Volume = CUBE(side);
```

Then the preprocessor expand this statement to :

```
Volume = (side * side * side);
```

If consider the following statement:

Volume = CUBE(a+b)

This would be expand to :

Volume = (a+b * a+b * a+b)

- **Nesting of the macros :**

✓ We can also use one macro inside another macro is called the Nesting of the macro.

Ex:

```
#define M 5
#define N M+1
#define SQU(x)  ((x)*(x))
#define CUBE(x) (SQU(x) *(x))
#define SIXTH(x) (CUBE(x) * CUBE(x))
```

- ✓ The preprocessor expands each #define macro, until no more macro appear in the text.
- ✓ For example, the last definition is first expanded into
  ((SQU(x) * (x)) * (SQU(x) * (x)))
- ✓ Since SQU(x) is still a macro, it is further expanded into line.
  ((((x) * (x))*(x)) * (((x) * (x)) * (x)))
- ✓ Which is finally evaluated as x^6

- **Undefining a Macro:**
- ✓ A define macro can be undefined, using the statement.
  #undef identifier
- ✓ This is useful when we want to restrict the definition only to a particular part of the program.

❑ FILE INCLUSION:

- ✓ An external file containing functions or macro definition can be included as a part of a program so that we need not rewrite those functions or macro definitions. This is achieved by the preprocessor directives.

  #include “filename”

- ✓ Where filename is the name of the file containing the required definition of functions. At this point, the preprocessor inserts the entire contents of filename into the source code of the program. When the filename is included with the double quotation marks, the search for the file is made first in the current directory and then in standard directories.

  #include <filename>

- ✓ Without double quotation marks. In this case, the file is searched only in the standard directories.
- ✓ Nesting of included file is allowed. That is an included file include another file.

Ex:

```
#include<stdio.h>
#include “SYNTAX.C”
#include “STAT.C”
#include “TEST.C”
```

❑ COMPILER CONTROL DIRECTIVES :

- ✓ While developing large programs, you may face one or more of the following situation.

- **Situatuion1 :**

- ✓ You have included a file containing some macro definitions. It is not known whether a particular macro(say, TEST) has been defined in that header file.

- ✓ This situation refers to the conditional definition of a macro. We want to ensure that the macro test is always defined, irrespective it has been defined in the header file or not.
- ✓ This can be achieved as follow.

```
#include “DEFINE.H”
#ifndef TEST
#define TEST 1
#endif
```

- In the above case searched for the definition of TEST in the header file and if not defined, then all the lines between the #ifndef and the corresponding #endif directives are ‘active’ in the program.
- That is the processor

  #define TEST is processed.

- In case, the TEST has been defined in the header file, the #ifndef condition become false , therefore the directive #define TEST is ignored.

  #ifdef TEST

  #undef TEST

  #endif

- In the above case, even if TEST is defined in the header file, its definition is removed.

- **Situation 2:**

✓ Suppose a customer has two different types of computers and you are required to write a program that will run on both the system. You want to use the same program, although certain lines of code must be different for each system.

✓ The main concern is to make the program portable.

✓ This can be achieved as following:

```
main()
{
    ……..
 #ifdef IBM_PC
{
    ……..          / / Code for IBM_PC
}
#else
{
 ……..             / / Code for HP machine
}
#endif
   ……..
}
```

✓ If we want the program to run of IBM PC. We include the directive

#define IBM_PC

- **Situation 3**

✓ This is similar to the above situation

```
#ifdef  ABC
   group A lines
#else
   group B lines
#endif
```

✓ Group A lines are included if the customer ABC is defined. Otherwise Group B lines are included.

- **Situation 4:**

✓ This is useful for debugging a particular macro.

```
Ex: #ifdef Test
    {
      printf("Array Element");
      for(i=0;i<m;i++)
        printf("x[%d] = %d\n", I, x[i]);
    }
    #endif
    ……..
    #ifdef TEST
      printf(…….);
    #endif
```

✓ Directives #ifdef and #endif are included only if the macro TEST is defined. Once Every thing is OK undefine the TEST.This makes the #ifdef TEST condition become false and therefore all the debugging statements are left out.

- ✓ **The C preprocessor also support a more general form of test condition - #if directive. This takes the following form:**

```
#if constant expression
{
statement 1
statement 2
……..
……..
}
#endif
```

- ✓ **If the result of the constant expression is true then all the statements are executed otherwise they are skipped.**

- ✓ **As a constant expression may be any logical expression.**

- **ANSI ADDITION :**
  - ✓ ANSI committee has added some more preprocessor directives listed below.

    #elif - Provides alternative test facility

    #pragma – Specifies certain instructions

    #error – Stops compilation when an error occurs
  - ✓ The ANSI standard also includes two new preprocessor operations.

    # - Stringizing operator

    ## - Token-pasting operator

Chapter 15 Debugging

❑ Program Design :

- ✓ Before coding a program, The program should be well conceived and all aspects of the program design should be considered in detail.
- ✓ Program design is basically concern with the development of a strategy to be used in writing a program.
- ✓ The Program design involves the following four stages.
  1. Problem Analysis
  2. Outlining the Program Structure
  3. Algorithm Development
  4. Selection of Control Structure

❑ Program Coding :

- ✓ The Algorithm developed in the previous section must be translated into set of instructions that the computer can understand.
- ✓ The major emphasis in coding should be simplicity and clarity.
- ✓ A Program written by one may have to be read by other later. Therefore it should be readable and simple to understand.
- ✓ Complex logic and coding should be avoided.
- ✓ The elements of coding style includes,
  1. Internal Documentation
  2. Construction of Statements
  3. Generality of the Program
  4. Input / Output Format

❑ Common Programming Errors :

✓ Some common programming errors are following:

1) Missing semicolon
2) Misuse of semicolon
3) Missing braces
4) Missing quotes
5) Misusing quotes
6) Improper comment character
7) Undeclared variables
8) Forgetting the precedence of operators
9) Ignoring the order of evaluation of increment / decrement operators
10) Forgetting to declare function parameters
11) Mismatching type in function calls
12) Non declaration of function
13) Missing & operator in scanf()
14) Crossing the bounds of an array
15) forgetting a space for null character in a string
16) Using uninitialized pointers
17) Missing indirection and address operators
18) Missing parentheses in pointer expression
19) Omitting parentheses around arguments in macro

❑ Program Testing and debugging :

- ✓ Testing and debugging refer to the tasks of detecting and removing errors in a program, so the program produce the desired results.
- ✓ It is therefore necessary to make efforts to detect ,isolate and correct any errors.

▪ Types of error:

1) Syntax errors
2) Run time errors
3) Logical errors
4) Latent errors

▪ Program Testing :-

- ✓ Testing is the process of reviewing and executing a program with the intent of detecting errors, which may belong to any of the four kinds discussed above. Testing should include necessary steps to detect all possible errors in the program. Testing process may include the following two stages:

1) human testing : It is effective process and is done before the computer based testing begins.
2) computer based testing : It involves two stages compiler testing and run time testing.

- **Program Debugging :-**

✓ **Debugging is the process of isolating and correcting the errors. One simple method of debugging is to place print statements through the program to display the values of variables.**

✓ **Another approach is to use the process of deduction. The location of an error is arrived at using the process of elimination and refinement.**

✓ **The third error locating method is to backtrack the incorrect results through the logic of the program until the mistake is located.**

❑ Program Efficiency:

✓ the efficiency of program is measured with two computer resources, execution time and memory. Efficiency can be improved with good design and coding practices.

- **Execution Time:**

✓ The execution time is directly tied to the efficiency of the algorithm selected. Certain coding techniques can consider to improve the execution efficiency.

1) Select the fastest algorithm
2) Simplify arithmetic and logical expression
3) Use fast arithmetic operations
4) Carefully evaluates loops
5) Avoid the use of multi dimensional array
6) Use pointer for handling arrays and strings

- **Memory requirement:**

✓ There are some necessary steps to compress memory requirements:

1) keep the program simple.
2) Use an algorithm that is simple.
3) Declare arrays and strings with correct sizes.
4) Limit the use of multi dimensional arrays.

❑ Syntax error, Run time error, Logical error and Latent error

There are four types of errors:

1) Syntax errors:-

Any violation of rules of the language results in syntax errors. The compiler can detect and isolate such errors. When syntax errors are present, the compilation fails and is terminated after listing the errors and the line numbers in the source program, where the errors have occurred.

2) Run time errors:-

Errors such as a mismatch of data types or referencing an out-of-range array element go undetected by the compiler. A program with these mistakes will run, but produce erroneous results and therefor, the name run time error is given such errors. Isolating the run time error is usually a difficult task.

3) Logical errors :-

As the name implies, these errors are related to the logic of the program execution. Such action as taking the wrong path, failure to consider a particular condition and incorrect order of evaluation of statement belong to this category. Logical errors do not show up as compiler generated messages. Rather they cause incorrect results. These errors are primarily due to a poor understanding of the problem, incorrect translation of the algorithm into the program and lack of clarity of hierarchy of operators.

4) Latent errors:-

It is a hidden error that shows up only when a particular set of data is used. For example, consider the following statement:

Ratio = (x+y)/ (p-q);

An error occurs only when p and q are equal. An error of this kind can be detected only by using all possible combination of test data.