Civinnovate

Discover, Learn, and Innovate in Civil Engineering

Notes on C

Lecture 1

Introduction to Computer

All digital computers are basically electronic devices that can transmit, store and manipulate information (data). The data may be numeric, character, graphic, and sound. So, a computer can be defined as *it is an electronic device which gets raw data as input, processes it and gives result as output, all under instructions given to it.*

To process a particular set of data, the computer must be given an appropriate set of instructions, called a program. Such instructions are written using codes which computer can understand. Such set of codes is known as language of computer.

Types of programming language

Many different languages can be used to program a computer. As we know that computer is composed of millions of switches which acts like electric switch. Such a switch has two states, ON and OFF, and can be represented with 1 and 0. So, a computer can understand only 0, and 1, which is known as BInary digiTS, in short BITS. *So, language which uses these binary codes to instruct a computer is known as Machine language.* Machine language is very rarely used to instruct a computer because it is very difficult and machine dependent (different machine may need different machine codes).

Instructions written in high level language is more compatible with human languages. Hence, it is easier to instruct computer using it, however it is necessary to be translated into machine codes using translator such as compiler and interpreter. Such programs written in High level language, are portable (can be used in any computer without or with some alteration).

A compiler or interpreter is itself a computer program that accepts a high level program as input and generates a corresponding machine language program as output. The original high level program is called the source program and the resulting machine language program is called the object program. Every high level language must have its own compiler or interpreter for a particular computer.

Introduction to C

It is a high level programming language. Instructions consist of algebraic expressions, English keywords such as *if, else, for, do, while* etc. In C, a program can be divided down into small modules. Hence, it is also called as structured programming language.

- Flexible to use as system programming as well as application programming.
- Huge collection of operators and library functions.
- User can create own user defined functions.
- C compilers are commonly available for all types of computers and program developed in C, can be executed in any computer without or with some alteration, hence it's portable.
- A program developed in C is compact and highly efficient.
- Because of modularity, it is easy to debug (to find error).

- It can also be used as low level language.

Lecture 2

History of C

C was developed in the 1970's by Dennis Ritchie at Bell Lab. It was developed from earlier languages, called BCPL and B which were also developed at Bell Lab. It was confined within Bell Lab till 1978. Dennis Ritchie and Brian Kernighan further developed the language. By mid 80's, it became more popular. Later on, it was standardized by ANSI (American National Standard Institute).

Structure of a C program

Every C program consists of one or more functions, one of which must be called *main*. The program always begins by executing the main function however it contains other functions.
Each function must contain:

- A function heading, that consists function name and arguments enclosed in parenthesis.
- A pair of compound statement (curly braces).
- It may consist of number of input/output statements.
- Library file access
- Comments

Program 1:

```
/* A program to print Hello*/          Comment
#include<stdio.h>                      Library file access
main( )                                Call of main function
{                                      Compound statement start
printf("\n Hello");                    Output statement
}                                      Compound statement end
```

program 2:

```
/* A Program to find sum of two integer numbers 12, & 17 */

#include<stdio.h>
void main( )
{
    int x=12, y= 17;
    z= x + y;
    printf(" sum is %d", z);
}
```

C Fundamentals

The basic elements of C includes C character set, identifiers, keywords, data types, constants, variables, arrays, declarations, expressions and statements.

The C character set

C uses uppercase A to Z, the lowercase letters a to z, the digits 0 to 9 and certain special characters such as:

! ^ # % ^ & * ( ) ~ _ -

= + \ | [ ] { } ; : '

" , < . > / ? (blank)

Most versions of C also allow using @ $.

It can be combination of certain characters such as \n, \t to represent non-printable characters new line, horizontal tab respectively. Such character combination to print non printable character is known as *escape sequence*.

Identifiers

Identifiers are the names given to various element of program such as variables, functions and arrays. Identifiers consist of letters and digits. Rules are to be followed for identifiers:

- It may consist of character, digits but first character must be letter.
- It permits uppercase and lowercase but they not interchangeable.
- It may begin with underscore (_) too.
- Most of C allows 31 chars.
- Space and some special character are not allowed.

eg. x1, sum, _temp, Table etc.

Some invalid identifiers are
1x, "x", -temp, error flag etc.

Keywords

There are certain reserved words in C, which are called as keywords and such words has predefined meaning. These words can only be used for their intended purpose.

The standard keywords are:

auto	extern	sizeof	break	float
static	case	for	struct	char
goto	switch	const	if	typedef
int	union	Default	long	continue
signed	unsigned	Do	register	void
double	return	Volatile	else	short
while	enum			

Some compilers may also include:

ada	far	near	asm	fortran
pascal	entry	huge		

Note: keywords must be in lowercase.

Constant

Constant is a basic element of C which doesn't change its value during execution of program. Four basic types of constant in C are:

constant type	*example*	*Illegal*
integer	200, -5	12,200; 3.0; 10 20; 090; 1-2
floating-point	20.5; -2.5; 1.6e+8	1; 1,00.0; 2e+10.2
character	'a'; '3'; ' '; '\n'	3
string	"anuj"	'st xavier's'

Variables

A variable is an identifier that is used to represent some specified type of information within a designated portion of the program. A variable represents a single data item, that is, a numerical quantity, or a character constant. Such data item can be accessed later in any portion of program by referring name of variable.

Array

An array is *an identifier that refers to a collection of data items* which all have the same name with different subscript but they must be same data type (i.e. integer, floating point or character). Individual data item in an array is known as *array element*.

e.g.

```
int a=4, b=5, c=2, d= -5, e=0;
```

In terms of array, it can be expressed as follows:

```
int x[5] = {4, 5, 2, -5, 0};
```

where,

```
x[0] = 4
x[1] = 5
x[2] = 2
x[3] = -5
x[4] = 0
```

Data types

C supports different types of data, each of which may be represented differently within the computer's memory. But memory requirement for each data type may vary from one compiler to another.

Data type	Description	Memory in bytes
int	integer quantity	2
char	single character	1
float	floating point number	4
double	double precision floating point number	8

Declaration

All variables must be declared before they appear in a program in order to reserve memory space for each data item. A declaration may consist of one or more variables of same data type. A declaration begins with data type following with one or more variables and finally ends with a semicolon.

e.g.

```
int x=6, y=7, z;
float a=3.0, b=1.5e+5, c;
char section='a', name[20] = "Xavier";
```

\

```
/* A Program to find sum of any two input integer numbers */

#include<stdio.h>
void main( )
{
     int  x, y;
     printf("\n Enter a number");
     scanf(" %d",&x);
     printf("\n Enter another number");
     scanf("%d",&y);
     z= x + y;
     printf(" sum is %d", z);
}
```

Program 4:

```
/* A Program to find area of a circle for input radius */

#include<stdio.h>
void main( )
{
     float a, r;
     printf("\n Enter radius");
     scanf(" %d",&r);
     a = 3.1415 * r * r;
     printf(" \n area of circle is %f", a);
}
```

Expression

An expression represents a single data item, such as a number or a character. The expression may consist of a single entity, such as a constant, a variable, an array element or a reference to a function. It may consist of some combination of such entities interconnected by one or more operators.

```
a > b
c = a + b
```

Statement

A statement causes the computer to carry out some action. Three different types of statements are:

Expression statement :

An expression statement consist of an expression followed with a semicolon.

e.g.

```
c = a + b;
```

Compound statement :

A compound statement consists of several individual statements enclosed within a pair of braces ({ and }).

e.g.

```
    {
        int  x=3;
        printf ("%d", x);
    }
```

Control statement :

A control statement is such a statement which controls execution of other statements.

e.g.

```
    if(x>0)
        printf(" x is positive");
```

Symbolic Constant

A symbolic constant is name that substitutes for a sequence of characters. The characters may be numeric, character or string constant. It replaces in place of numeric, character or string constant in the program. While compiling the program, each occurrence of a symbolic constant is replaced with its corresponding character sequence.

A symbolic constant is defined by writing

```
# define   name    text
```

e.g.

```
# define          PI        3.1415
#define           NAME  "Kathmandu"
```

Program 5:

/ A Program to find area and perimeter of a circle for input radius */*

```
# include <stdio.h>
 # define PI 3.1415
    void main( )
    {
        float a, r, p;
        printf("\n Enter radius of circle");
        scanf("%f",&r);
        a = PI * r * r;
        p = 2*PI*r;
        printf(" \n area of circle is %f", a);
        printf(" \n perimeter of circle is %f", p);
    }
```

Problem 1:	Write a program to find area of a triangle for input base and height.
Problem 2:	Write a program to find Total Amount for input Rate and Quantity.
Problem 3:	Write a program to convert length in cm to inch for input length.

Problem 4: Write a program to convert temperature in Celsius to temperature in Fahrenheit for input temperature.

OPERATOR

Operator is a symbol that is used to combine data items such as constant, variable, function reference and array element to form an expression. The data item that acts upon is called Operand.

e.g.

c = a + b

in this expression, = and + are the operators which combines variables a, b and c.

There are 7 basic types of operators in C.

UNARY OPERATOR

Operator that acts upon a single operand is known as Unary Operator. Some unary operators are -, ++, --, sizeof, type

e.g.

```
        -x;     i++; sizeof(a); (float) 5/3;
```

Program 1.

/ Program to differentiate use of unary operator as prefix and suffix */*

```
#include<stdio.h>
void main()
{
        int x=5;
        printf("%d",x);
        printf("%d",x++);
        printf("%d",x);

        printf("%d",++x);
        printf("%d",x);

}
```

Program 2.

/ Program to illustrate use of sizeof and type operators */*

```
#include<stdio.h>
void main()
{
        int x=5,y=4;
        float z;
        printf("size of x is %d bytes",sizeof(x));
        printf("%d",x/y);
        printf("size of z is %d bytes",sizeof(z));
        printf("%f",(float)x/y);
}
```

ARITHMETIC OPERATOR

The operator which is used for general mathematical operations, is called Arithmetic operator. The five arithmetic operators are +, -, *, /, %.
Here, % is known as modulus operator which returns reminder after integer division.

Program 3.

```
/* to differentiate /(division) and % (modulus operator)*/
void main()
{
        int x=5,y=2, z1,z2;
        z1=x/y;
        z2=x%y;
        printf("\n Quotient : %d",z1);
        printf("\n Remainder : %d",z2);
}
```

RELATIONAL OPERATOR

Relational operators are used to compare two data items. There are four relational operators, they are >, <, >=, <=.
e.g.

```
        (x>y)
```

Program 4.

```
/* Program to check whether user can or cannot vote*/
main()
{
        int age=25;
        if(age>=18)
                printf("\n You can vote.");
        else
                printf("\n You cannot vote.");
}
```

EQUALITY OPERATOR

Equality operator is also used to compare two data items whether they are equal or not. There are two equality operators, ==, !=.

Program 5.

```
/* Program to show whether input number is positive, zero or negative*/
```

```
void main()
{
        int n;
        if(n>0)
                printf("\n %d is positive.",n);
        else if(n==0)
                printf("\n %d is zero.",n);
        else
                printf("\n The number %d is negative");
}
```

LOGICAL OPERATOR

The logical operator acts upon operands those are themselves logical expression. Such logical expressions are combined together to form more complex expression. The two logical operators are *logical and* (&&) and *Logical or* (||).

Program 6.

/* Program to find highest number among three input integer numbers */

```
void main()
{
        int x,y,z;

        printf("\n enter three integer numbers");
        scanf("%d%d%d",&x,&y,&z);

        if(x>y && x>z)
                printf("%d is the highest number",x);
        else if(y>x && y>z)
                printf("%d is the highest number",y);
        else
                printf("%d is the highest number",z);

}
```

Program 7.

/* Program to check whether user can or cannot apply for DV */

```
#include<stdio.h>

void main()
{
        int age;

        printf("\n enter your age");
        scanf("%d",&age);

        printf("\n enter your qualification (no. of years)");
        scanf("%d",&qual);

        printf("\n enter your experience (no. of years)");
        scanf("%d",&exp);

        if(age >= 18 && (qual >= 12 || exp >= 2)
                printf(" You can apply");
        else
                printf("You cannot apply ");

}
```

CONDITIONAL OPERATOR

Simple conditional operator can be carried out with conditional operator (? :). It can be written in place of if...else statement. It can be used in this form.
expression1 ? expression2 : expression3;
e.g.

```
        (age>=18) ? printf("can vote") : printf("cannot vote");
```

/* Program to check whether user can or cannot vote using conditional operator*/

ASSIGNMENT OPERATOR

This type of operator assigns some value to left of operator after executing expression to its right. Some assignment operators are =, +=, -=, *=, /=, %=.

e.g.

x+=2	equivalent to	x=x+2
x-=3	equivalent to	x=x-3
x*=2	equivalent to	x=x*2
x/=2	equivalent to	x=x/2
x%=2	equivalent to	x=x%2

Operator precedence

Precedence is the hierarchical order of operators. Operation with higher precedence group is carried out before lower precedence group.

Another important consideration is the order in which consecutive operations within the same precedence group is carried out. This is known as associativity.

Type	Operators	Associativity
Unary	-, ++, --, sizeof, type	R → L
Arithmetic	*, /, %	L → R
	+, -	L → R
Relational	<, <=, >, >=	L → R
Equality	==, !=	L → R
Logical	&&	L → R
	||	L → R
Conditional	? :	R → L
Assignment	=, +=, -=, *=, /=, %=	R → L

e.g. a – b/c * d

First, it evaluates b/c then multiplies with d and finally result is subtracted from a.

c+= (a>0 && a<=10) ? ++a : a/b;
if a, b & c have values 1, 2 & 3 respectively, then result will be c=5;
if a, b & c have values 50, 10 & 20 respectively, then result will be c=25;

Library Functions

Library functions are pre-defined module in header files which can easily be accessed anywhere in the program by including its respective header file. Library function is the one of the important feature of C language. Because of huge collection of library function, it makes very easy to programmers for programming.

Library functions are defined for

- commonly used operations such as sqrt to find square root, pow to find power of any base etc.
- machine dependent standard input/output operations.

A library function is accessed simply by writing the function name, followed with list of arguments that represent information being passed to the function. The arguments must be enclosed in parentheses and separated by commas. The argument can be constant, or variable. But the parentheses must be present, even if there is no argument.

Some commonly used library functions:

abs(i)	- returns absolute value of i
clrscr()	- to clear screen
getch()	- to input a character without echo
getche()	- to input a character with echo
tolower(c)	– to convert a letter to lowercase
toupper(c)	– to convert a letter to uppercase
sin(d)	- return sine of d
cos(d)	- return cosine of d
tan(d)	- return tangent of d

sqrt(d)	- return square root of d
pow(d1,d2)	- return d1 raised to power d2
log(d)	- return natural logarithm to d
exp(d)	- return e raise to power d

Program 8.

/*To convert an input char to capital letter*/

```
 #include<stdio.h>
 #include<ctype.h>
 #include<conio.h>
 void main()
 {
        char c;
        c=getche();
        putch(toupper(c));
}
```

Program 9.

```
/*To find value of sine of angle 30 degree*/
 #include<stdio.h>
 #include<conio.h>
 #include<math.h>
 #define PI 3.141593

 void main()
 {
         float x=30;
         clrscr();
         printf("%f",sin(x*(PI/180)));
         getch();
 }
```

Program 10.

```
/*To find value of antilog of log of specified no.*/
 #include<stdio.h>
 #include<conio.h>
 #include<math.h>
 void main()
 {
         float x=1;
         clrscr();
         printf("%f",log(x));

         printf("%f",exp(log(x)));
         getch();
 }
```

Some more about Escape Sequence:

Character	*Escape Sequence*
bell	\a
backspace	\b
horizontal tab	\t
new line	\n
form feed	\f
quotation mark (")	\"
apostrophe (')	\'
question mark	\?
backslash	\\
null	\0

Input data using Scanf Function

This function can be used to enter any combination of numerical values, single characters and strings. The function returns the number of data items that have been entered successfully.

scanf(control string, arg1,arg2,...argn)

where control string refers to a string containing certain required formatting information. arg1, arg2, ... argn represents the individual input data items.

Within the control string comprises individual groups of characters, with one character group of each input data item. Each character group must begin with a percent sign (%) followed with a conversion character which indicates the type of the corresponding data items.

Conversion character	***Meaning***
c	**data item is a single character**
d	**data item is an integer without decimal**
e	**data item is a floating point value in exponent form**
f	**data item is a floating point value**
g	**data item is a floating point value without trailing zeros.**
h	**data item is a short integer**
i	**data item is a signed integer number**
o	**data item is an octal integer**
s	**data item is a string followed by a whitespace character**
u	**data item is unsigned integer**
[...]	**data item is a string which may include white space**

```
/* To print a line of input text without white
space */

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

void main()
{

char x[20];

printf("\n Enter a string");
scanf("%s",x);

printf("%s",x);

getch();
}
```

```
/* To print a line of input text with white space
*/

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

void main()
{

char x[20];

printf("\n Enter a string");
scanf("%[^\n]",x);

printf("%s",x);

getch();
}
```

```
/*displaying a floating point number with diff format*/

#include<stdio.h>
#include<conio.h>
void main()
{
float x=123.456;
printf("%f %.2f ",x,x);
printf("%e %.2e",x,x);
printf("%g %.2g",x,x);

getch();
}
```

Output:

```
123.456000                  123.46
1.234560e+02                1.23e+02
123.456                     1.23e+02
```

Program 11.

```
/* To convert an integer number into hexadecimal number */
#include<stdio.h>
#include<conio.h>
void main()
```

```
{
        int x=65;
        clrscr();
        printf("%x",x);
        printf("%o",x);

}
```

Program 12.

```
/* To print unsigned integer and long integer number */
#include<stdio.h>
#include<conio.h>
void main()
{
        unsigned int x=65535;
        long int y=2000000000;

        printf("\n%u",x);
        printf("\n%ld",y);

    getch();
}
```

Program 13.

```
/* To print long integer number */
#include<stdio.h>
#include<conio.h>
void main()
{
        long int x=2000000000;
        printf("\n%ld",x);
    getch();
}
```

Control Statement

There may be a statement in a program which controls the execution of other statements, such type of statement is known as Control Statement. It controls the sequence of execution of program.

Some logical test may be carried out at particular point of program. Then one action will be carried out depending upon the outcome of logical test which is known as conditional execution. It may execute a group of statements among several available groups of statements, which is known as selection.

Some control statements are if...else, for, while, do...while, switch...case, break, continue, goto.

Many programs require that a group of instructions can be executed repeatedly, until some logical condition is satisfied. This is known as looping.

if...else statement
It carries out a logical test and then takes one of two possible actions, depending upon the outcome of test. But the else portion is optional.

```
if<expression>
{
        Statement1; Statement2; ...         Statement n;
}
else
{
        Statement1; Statement2; ...         Statement n;
}
```

If the expression is TRUE, it executes the group statements following if statement, otherwise executes the group of statements following else statement.

```
/* To check whether input character is capital or small letter */
#include<stdio.h>
#include<conio.h>
void main()
{
char x;
clrscr();
printf("Press any alphabet key");
x=getch();
if(x>='A'&&x<='Z')
        printf("\nIt's capital letter");
else
        printf("\nIt's small letter");
getch();
}
```

if...else if ... else statement
It carries out logical tests and then executes one of number of possible actions, depending upon the outcome of test.

```
if<expression1>
```

```
{
        S1; S2; ...     Sn;
}
else if <expression2>
{
        S1; S2; ...     Sn;
}
....
else if <expression n>
{
        S1; S2; ...     Sn;
}

else
{
        S1; S2; ...     Sn;
}
```

```
/* To check whether input character is capital, small, digit or other character */
#include<stdio.h>
#include<conio.h>
void main()
{
char x;
clrscr();
printf("Press any alphabet key");
x=getch();
if(x>='A'&&x<='Z')
        printf("\nIt's capital letter");
else if(x>='a'&&x<='z')
        printf("\nIt's small letter");
else if(x>='0'&&x<='9')
        printf("\nIt's digit");
else
        printf("\nIt's other character");
getch();
}
```

goto statement

The goto statement is used to alter normal sequence of program execution by transferring its control to some other part of program.

syntax:

```
          goto label;
```

where, label is an identifier to which the control is to be transferred. Label should be given at any part of program where the control is to be transferred, followed with a colon(:).

```
/* To print integer numbers from 1 to 10 */
#include<stdio.h>
main()
{
          int i=1;
start:
          if(i<=10)
          {
                    printf("\n %d",i);
                    i++;
                    goto start;
          }
}
```

Assignment

1. WAP to find highest number among 3 input integer numbers.
2. WAP to display whether an input integer number is even or odd.
3. WAP to find total, percentage, result (Pass/Fail) and division of a student for input marks of ENG, PHY, CHEM.
4. To print series 1,3,5,7,9 using goto.
5. To print 1,1,2,3,5,8,13,21 using goto.

for statement

The *for* statement is the most commonly used as looping statement. It includes 3 expressions in which first expression initializes index (counter), second expression determines whether the loop is to be continued or not and third expression modifies index (counter) value. So, third expression is generally unary or assignment expression is used.

The general form of for statement is as follows:

```
for(expression 1; expression 2; expression 3)
```

When for loop is executed, first it initializes counter as specified in expression 1, then evaluates the expression 2 whether the loop is to be continued or not. If the expression 2 is satisfied (TRUE or 1) then only the group of statements within the loop is executed. And finally, only the expression 3 is executed by modifying the counter. So, the loop is continued until expression 2 is FALSE or 0.

```
/* To display a message 10 times */
#include<stdio.h>
main( )
{
        int i;
        for(i=0; i<10; i++)
        {
                printf("\n Good Morning");
        }
}
```

```
/* To print 10 consecutive integer numbers */
#include<stdio.h>
main( )
{
        int i;
        for(i=1; i<=10; i++)
        {
                printf("\n %d", i);
        }
}
```

```
/* To print odd integer numbers between 1 to 50 */
#include<stdio.h>
main( )
{
        int i;
        for(i=1; i<=50; i+=2)
        {
                printf("\n %d", i);
        }
}
```

Assignment 2

6. **WAP to print numbers 100, 81, 64 ... 1.**
7. **WAP to print numbers 1, 2, 4, 8 .... 512**
8. **WAP to print 0.1,0.01,0.001. ....0.0000001**
9. **To print as 0.3, 0.33,0.333..... 0.3333333**
10. **To check whether input no. is prime or composite.**

Infinite For loop
Infinite loop occurs when the expression 2 in the For loop is TRUE for infinite times.
e.g.

```
        for(i=0;i<10;)
```

But,

```
        for(i=0;i<10;i--)
```

is not infinite loop.

Null Loop
It executes without any embedded statements.

```
        for(i=0;i<10;i++);
```

This loop executes 10 times without executing any other statements. But final value of counter will be 11 after completion of loop.

While Statement
The While loop is also used to carry out looping operations. Generally it is used for prior unknown steps of loops to be executed whereas the For loop is used for prior known steps of loops to be executed.
The general form of While statement is :

while(expression)

The while loop is executed repeatedly till the expression is TRUE (not zero). So, it is used for unknown loops till some condition is not satisfied within the While loop.

Counter is initialized before the loop and modified within the loop, if necessary.

```
/* To print 10 consecutive integer numbers using while loop */
#include<stdio.h>
main( )
{
        int i=1;
        while(i<=10)
        {
                printf("\n %d", i);
                i++;
        }
}
```

```
/* To find sum of positive integer numbers until negative number is entered */

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

void main()
{
        int x=0,sum=0;

        while(x>=0)
        {
                sum+=x;
                scanf("%d",&x);
        }
        printf("\n%d",sum);
        getch();
}
```

do... while statement

When a loop is written using *while* or *for* statements, the condition is checked at the beginning of each pass. Whereas, it may require to write such a loop in which condition is to be checked at end of each pass whether the loop is to be continued or not. So, such a loop can be achieved by means of *do... while* statement.

The general form of do...while statement is:

```
do{
        statements;
        }while(expression);
```

The statements within the loop will be executed as long as the expression is TRUE. *The statements will be executed at least once however the expression is never satisfied.*

It is better to test the condition before the continuation of loop. So, *for* & *while* statements are frequently used with comparing to *do...while* statement.

It can be initialized & modified the counter as in *while* statement.

```
/* To print 1 to 10 consecutive integer numbers */
#include<stdio.h>
#include<conio.h>

void main()
{
        int i=1;
        do{
                printf("\n %d",i);
                i++;
        }while(i<=10);
        getch();
}


/* To convert a line of input lowercase text to uppercase */
#include<stdio.h>
#include<conio.h>
#include<ctype.h>

void main()
{
        char x[80];
        int i= -1,k;
        clrscr();

        do      {
                        i++;
                } while((x[i]=getchar())!='\n');

        k=i;
        i=0;
        while(i<k)
        {
                putchar(toupper(x[i]));
                i++;
        }
        getch();
}
```

Nested loops

One loop can be embedded within another loop without overlapping each other, is known as *Nested loops*. It may consist of different control statements as inner loop and outer loop, but index (counter) must be different for different loop.

```
/* To print multiplication tables of 1 to 10 */
#include<stdio.h>
#include<conio.h>

void main()
{
        int i,k;
        for(k=1;k<=10;k++)
        {
                i=1;
                while(i<=10)
                {
                        printf("\n %d x %d = %d",k,i,i*k);
                        i++;
                }
        getch();
        }
}
```

Assignment 3:

1. **Write a program to find individual average of 5 lists of input numbers.**
2. **Write a program to print prime numbers among 1 to 100.**
3. **WAP to print**

a)
```
1
23
345
4567
56789
678901
7890123
```

b)
```
*
**
***
****
*****
```

Switch Statement

The switch statement is used to execute particular group of statement among available groups of statements. The selection depends upon the current value of expression following switch statement.

The general form of switch statement is:

```
switch (expression)
{
        case label1:
        {
                statements;
                break;
        }
        case label2:
```

```
        {
                statements;
                break;
        }
        ......................
        ......................
        case labeln:
        {
                statements;
                break;
        }
        default:
        {
                statements;
        }

        where, expression may be int or char type.
```

Switch statement is similar to if...else if...else statement, but switch statement executes faster with comparing to if...else if...else because it directly executes the matching group of statement from the available groups of statements. But, the case label (case prefix) must be unique for each group.

```
/* To find sum, difference, product or quotient of any two input integer numbers using integer type
expression*/

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

void main()
{
        int n,m,r=0;
        int c;
        clrscr();

        printf("enter 2 no.");
        scanf("%d%d",&n,&m);
start:
        printf("\n1. sum");
        printf("\n2. diff");
        printf("\n3. product");
        printf("\n4. quotient");
        printf("\n Select numbers (1-4)");
        scanf("%d",&c);

        switch(c)
        {
                case 1:
                {
                        printf("\nsum is %d",n+m);
                        break;
                }

                case 2:
                {
                        printf("\n Difference is %d",n-m);
                        break ;
                }
                case 3:
                {
                        printf("\n Product is %d",n*m);
                        break;
                }
                case 4:
                {
                        printf("\n Quotient is %f",(float)n/m);
                        break;
                }
                default:
                {
                        printf("\n wrong selection");
                        goto start;
                }
        }
getch();
}
```

```
/* To find sum, difference, product or quotient of any two input integer numbers using char type
expression*/

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

void main()
{
        clrscr();
        int n,m,r=0;
        char c;
        printf("enter 2 no.");
        scanf("%d%d",&n,&m);
start:
        printf("\n1. sum");
        printf("\n2. diff");
        printf("\n3. product");
        printf("\n4. quotient");
        printf("\n Select numbers (1-4) or first char(s,d,p or q)");

        c=getch();

        switch(toupper(c))
        {
                case 'S':
                case '1':
                {
                        printf("\nsum is %d",n+m);
                        break;
                }
                case 'D':
                case '2':
                {
                        printf("\nsum is %d",n-m);
                        break ;
                }

                case 'P':
                case '3':
                {
                        printf("\nsum is %d",n*m);
                        break;
                }
                case 'Q':
                case '4':
                {
                        printf("\nsum is %f",(float)n/m);
                        break;
                }
                default:
                {
                        printf("\n wrong selection");
                        goto start;
                }
        }

getch();
}
```

Assignment

1. To find Area of a triangle, rectangle, square or circle asking required parameter
2. To print the value of angle of sine, cosine or tangent for input angle in degree

***break* statement**
The *break* statement is used to exit from a loop or from switch statement by transferring control out of entire loop or switch. It can be used within *for*, *while*, *do...while* or *switch* control statement.

It can be simply called as,

```
        break;
```

***continue* statement**
The *continue* statement is used to bypass the remainder statements of current step of loop but it continues the remaining steps of loop. So, it only skips the remaining statements of current step of loop.

It can also be used within for, while or do...while control statement as break statement.

```
/* To find sum of non negative integer numbers until negative number is entered*/
#include<stdio.h>
#include<conio.h>
void main()
{
        int i,sum=0,x;
        printf("\nEnter integer numbers");
        for(i=0;i<10;i++)
        {
                scanf("%d",&x);

                if(x<0)
                        break;
                sum+=x;
        }
        printf("\n Total is: %d",sum);
}
```

```
/* To find sum of non negative integer numbers among 10 input integer numbers*/
#include<stdio.h>

void main()
{
        int i,sum=0,x;
        printf("\nEnter integer numbers");
        for(i=0;i<10;i++)
        {
                scanf("%d",&x);
                if(x<0)
                        continue;
                sum+=x;
        }
        printf("\n Total is: %d",sum);
}
```

The *comma* operator

The comma operator is used to permit two expressions where generally one expression is used. Such as:

```
for(expn1a,expn1b;expn2;expn3a,expn3b)
```

where, expn1a and expn1b are the two expressions which initializes two counters(index); expn2 checks whether the loop is to be continued or not; and finally, expn3a & expn3b modifies the counters.

```
/* To check whether the input string is palindrome or not */

#include<stdio.h>
#include<conio.h>
#define EOL '\n'
#define TRUE 1
#define FALSE 0
void main()
{
	char x[80];
	int tag,i, k,flag;
	flag=TRUE;
	printf("\nPlease enter a word, phrase or sentence");

	for(i=0;(x[i]=getchar())!=EOL;i++)
		;
	tag=i-2;

	for((i=0,k=tag);i<=tag/2;(i++,k--))
	{
		if(x[i]!=x[k])
		{
			flag=FALSE;
			break;
		}
	}
	for(i=0;i<=tag;i++)
	{
		putchar(x[i]);
	}

	if(flag)
		printf(" is a palindrome");
	else
		printf(" is not a palindrom");
}
```

# Modify the previous program so that it checks words or phrases for palindrome or not as long as user wishes.

# WAP to print specified number of characters starting from specified character position of an inputstring

Functions

As we know that one of the important feature of c is its modularity. A program can be broken into small and self contained components modules(pieces), such a small module is known as function. In other words, ***a function is a self-contained program segment that carries out some specific, well-defined task.***

Some modules which are already defined in C library files, are known as library function. Such function can be accessed in any program by including its respective library/header file and they have their own predefined meaning and syntax to use.

C also allows defining a programmer his/her own function, which is known as user-defined function. Such functions can also be stored into a file as library file so that they can be accessed in any program whenever it is required.

Advantages of function:

Redundancy: While programming, it may require executing some instructions repeatedly. Such instructions can be defined as a function, which can be accessed easily by calling its name in any portion of program which reduces repetition of same instructions.

Clarity: It makes a program logically clear and easy to debug. Each function is well defined for a particular problem and it will have particular name which can be accessed by its name. So in main module, from which the particular function is being called, will have only function name. If any error found in the function, we need to debug only in the particular body of function definition.

Customized library: After defining a well-defined function, user may store into a library file. Hence, a function can be accessed in many programs which avoid repetition.

Every C program contains one or more functions. One is must, which is called ***main*** function and it always starts to execute from this function however there are more than one function in any order.

One function definition can not be embedded within another function i.e. functions can not be nested.

As a function is being called at any portion of program, the control will be transferred to the calling function with identifiers called arguments (parameters) and return back to the point from which the function was accessed.

/*To find sum of any two integer numbers using user defined function */

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


/* fx definition */
int sum(int x,int y)
{
      int z;
      z=x+y;
      return(z);
}
```

```
void main()
{
	int a,b,c;

	int sum(int x,int y);	/*Declaration of fx*/

	printf("\nEnter any two numbers");
	scanf("%d%d",&a,&b);

	c=sum(a,b);		/* fx call */
	printf("\n Total is %d",c);
	getch();
}
```

/*To convert an input character into uppercase using user defined function */

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

char toup(char x)
{
	char y;
	y=(x>='a'&&x<='z')?x-32:x;
	return(y);
}


void main()
{
	char x,y;
	char toup(char x);

	printf("\nEnter any character");
	x=getchar();

	y=toup(x);
	printf("\n Total is %c",y);
	getch();
}
```

Defining a function

A function definition contains major components: first line/heading, and the body of the function. The first line of function definition contains function type, name of function and arguments separated with commas, enclosed in a pair of parentheses. Here, arguments/parameters are optional but a pair of parentheses must be included however non of arguments are to be passed to the function.

The general form of first line of function definition:

data_type name(formal arg1, formal arg2,. formal argN)

Here,

- data_type represents data type of return value
- name represents the function name

and, number of formal arguments which represents the different data are to be passed into the function with their individual data types.

The arguments following the function name in function definition are known as ***formal arguments*** and they get data from calling program to the function.

The identifiers used within a function, have scope within the current function only. Hence there may be same identifier at different functions. Such type of identifier is called ***local*** identifiers.

The remainder of function definition is a compound statement that defines the action to be taken by the function which is known as ***body of function***. At the end of body of function may consist of ***return*** statement which shows the output of the function. Hence, the data type of function depends upon the data type of value that returns.

The general form of return statement as:

return expression;

The value of expression is returned to the calling portion of the program. The expression is optional, however, a return statement can be written without it. If the expression is omitted, the return statement simply causes control to revert back to the calling portion of the program, without any information transfer. Only one expression can be included in the return statement. Hence one function can return only one value to the calling portion of the program via return statement.

Accessing a function

A well defined function can be accessed by specifying its name following a list of arguments separated by commas, enclosed in a pair of parentheses. If there is no argument is to be passed, an empty pair of parentheses must follow the function name.

The corresponding argument in the function reference (function call) from where the function is being called, is known as ***actual argument***.

Passing arguments to a function

A value from a function can be passed via actual argument to a function. The value of corresponding formal argument can be altered within the function but it doesn't change the value of actual argument of calling function. This process of passing value of argument to a function is known as ***passing by value***.

```
#include<stdio.h>
modify(int a)
{
	a=0;
	printf("\n Value of a within fx is: %d",a);
}

void main()
{
	int a=5;
	printf("\n Value of a before fx is: %d",a);
	modify(a);
	printf("\n Value of a after fx is: %d",a);
}
```

Function Prototypes

A function should be declared in the ***main*** function as other data items, if the function is defined below the ***main*** function. But function declaration is optional if the function is defined before the ***main*** function.

The general form of function declaration which is also known as ***function prototype,*** is as follows:

```
data_type name(type1 arg1, type2 arg2, ... type n arg n);
```

Recursive

Sometime, a function may be called within itself repeatedly, until some specified condition is satisfied. Such a method of calling a function within the body of own function is known as RECURSION.

While writing an iterative (repetitive) function, we should take care of two things.

1) the fx should be called within the body of fx definition.
2) there should be a stopping condition in order to stop the execution of fx.

```
/* To find factorial of an input number using recursive
function*/
#include<stdio.h>
#include<conio.h>

double fact(double a)
{
    if(a<=1)
        return 1;
    else
        return(a*fact(a-1));
}

void main()
{
    clrscr();
    double a=6,x;
```

```
	x=fact(a);

	printf("\n %f",x);

getch();
}

/* To reverse an input string*/
#include<stdio.h>
#include<conio.h>

void reverse()
{
	char c;
	if((c=getchar())!='\n')
		reverse();

	putchar(c);
}

void main()
{
clrscr();
	printf("Please enter a line of text\n");
	reverse();
getch();
}
```

Assignments(Use user defined functions):

1. ***To find product of two numbers***
2. ***To determine the larger number of two integer numbers***
3. ***To calculate the factorial of N***
4. ***To find real roots of a quadratic equation.***
5. ***To determine value of x^n.***
6. ***To find square root of x.***
7. ***To check an input number is prime or composite.***
8. ***To determine largest number among 4 numbers.***

Program Structure

In earlier chapters, we've used local variables, which have scope within the single function in which the variable is defined. Such variable is not recognized in other functions. However, the same name is used in other functions; it is required to re-define the variable. In other words, it reserves different memory for both the variables however same name is used.

But, in some situations, it may require to define a variable in such a way so that the scope of variable remains more than one function or throughout the program from the point of its definition.

So, permanence of a variable and its scope within the program is characterized with Storage Class. It shows the portion of program over which the variable is recognized.

Hence, we can say that a variable has two characteristics, one data type and another storage class.

The four types of storage class are –

automatic, external, static and register which can be defined using keywords auto, extern, static and register respectively. Some typical variables can be declared as follows:

```
auto int a,b,c;
extern char name[10];
static int x;
```

Automatic variable

Automatic variables are always declared within a function and its scope retains within the function only in which they are declared. So, same name can be used as variables in different functions and different memories are allocated for them. Hence such variables are also known as ***local variables***. While declaring automatic variable within a function, it is not required to use ***auto*** keyword i.e. auto is optional.

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

float quad(int a, int b,int c, int i)
{
	float x=(-b+i*(sqrt(pow(b,2)-4*a*c)))/(2*a);
	return x;
}

void main()
{
	clrscr();
	auto int a=4,b=5,c=1;
```

```
        float x,y;

        x=quad(a,b,c,1);
        y=quad(a,b,c,-1);

        printf("\n %f",x);
        printf("\n %f",y);

getch();
}
```

External variables

The scope of external variable extends from the point of definition throughout the remainder of program. Since the variable is recognized throughout the program from the point of declaration, it retains its values. Hence, external variable can be assigned a value within a function and can be used within another function. It provides a convenient way for transferring information back and forth between functions without using arguments. One more, it can return more than one data items from a function without using RETURN statement.

```
/* To find real roots of quadratic equation */
#include<stdio.h>
#include<conio.h>
#include<math.h>

int a=4,b=5,c=1;

float quad(int i)
{
        float x=(-b+i*(sqrt(pow(b,2)-4*a*c)))/(2*a);
        return x;
}

void main()
{
        clrscr();

        float x,y;

        x=quad(1);
        y=quad(-1);

        printf("\n %f",x);
        printf("\n %f",y);
getch();
}
```

```
/* To find real root of quadratic equation */

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


extern int a=4,b=5,c=1;

float x,y;

void quad()
{
        x=(-b+(sqrt(pow(b,2)-4*a*c)))/(2*a);
        y=(-b-(sqrt(pow(b,2)-4*a*c)))/(2*a);

}

void main()
{
        clrscr();
        quad();

        printf("\n %f",x);
        printf("\n %f",y);

getch();
}
```

Static Variable

Static variable is defined within individual functions and therefore has the same scope as automatic variable. So, it is similar to automatic variable i.e. local to the current function in which it is defined. But, Static variable retains its previous values throughout the life of program however the function (in which static variables are defined) is re-called after exit.

Static variable is defined in the same way as automatic variable with keyword ***static*** before data type of variables which shows ***static storage class***.

```
#include<stdio.h>
#include<conio.h>
long int fact(int i)
{
        static long int x=1;
        return(x*=i);
}

void main()
{
        long int y;
        int a=6;
        for(int i=1;i<=a;i++)
                y=fact(i);

        printf("\n %ld",y);
}
```

It's unusual to define automatic or static variable having the same names as external variables. If happened, local variables will take precedence over the external variables. But it doesn't affect the values of external variables.

Here is one skeletal structure of a C program showing various storage class.

```
float a,b,c;

main()
{
        static float a;
        void dummy(void);
        ....
}

void dummy(void)
{
        static int a;
        int b;
        ........
}
```

Write programs using UDF(user defined function) different storage class.

1. to convert a character to uppercase using UDF.
2. to determine the greater number between two input integer numbers.
3. To check an input number is prime or composite
4. to find factorial of N.

Array

Some programs may need to handle same type of different number of data having same characteristics. In such situation, it will be easier to handle such data in Array, where same name is shared for all data with different subscripts. In an array, all the data items must be of same type and same storage class. Eg. either int, floating point or characters

Each array element (individual array element) is denoted with a name followed by one or more subscripts (where each subscript must be non negative integers within a pair of square bracket).

The number of subscript depends on the dimensionality of the array. Eg. a[i] refers to an element of one dimensional array. Whereas b[i][j] refers to an array element of two dimensional array. In the same manner c[i][j][k] for three dimensional array.

Defining an array

Array is also defined in the same manner as ordinary variables, but it should also include the size of specification that shows the maximum size of elements in that array. The size must be defined with a constant ie. cannot be used any variable as subscript while defining an array. But it can be a symbolic constant while defining size of an array.

It is defined such as follows:

Storage_class data_type array[expression];

```
e.g.    int x [5];
        float y[3][4];
        char name [MAX];     where MAX is symbolic constant.
```

Prog 1:
To store integer nos in an array and print them (1 dim)

```
#include<stdio.h>

void main()
{
        int i, x[10]={4,5,2,8,4,7,9,6,5};

        printf("\n The stored numbers in the array are:");

        for(i=0;i<10;i++)
        {
                printf("\n %d", x [ i ] );
        }
}
```

Prog 2:
Modify prog 1 so that it allows to enter integer nos. those are to be stored in an array.

Prog 3:
Modify prog 2 so that it could find out sum, average & deviation of data with average value.
Note: deviation = No. - average

Prog 4:
To sort a list of integer numbers in ascending order.

```
#include<stdio.h>

void main()
{
        int i,k, temp, x[10];

        printf("\n Enter numbers to be sorted :");

        for(i=0; i<10; i++)
        {
                scanf("%d", &x [ i ] );
        }

        for(i=0;i<9;i++)
        {
                for(k=i+1;k<10;k++)
                {
                        if(x [ i ] > x [ k ]  )
                        {
                                temp = x [ i ];
                                x [ i ]= x [ k ];
                                x [ k ]= temp;
                        }
                }
        }



        printf("\n The sorted numbers are:");

        for(i=0;i<10;i++)
        {
                printf("\n %d", x [ i ] );
        }
}
```

Two dimensional array
Prog 5:
To store numbers in a two dimensional array and print them

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


void main()
{
        int i,k,x[2][3]={1,2,3,4,5,6};
        for(i=0;i<4;i++)
        {
                for(k=0;k<4;k++)
                {
                        printf("%d\t",x[i][k]);
                }
                printf("\n");
        }
}
```

Prog 6:
Modify prog 5 so that it allows to enter numbers.

Prog 7:
Write a program to find out sum of two matrices of size 2x3

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


void main()
{
        int i,k,x[2][3],y[2][3],z[2][3];

        for(i=0;i<4;i++)
        {
                for(k=0;k<4;k++)
                {
                        scanf("%d",&x[i]);
                }
        }

        for(i=0;i<4;i++)
        {
                for(k=0;k<4;k++)
                {
                        scanf("%d",&y[i]);
                }
        }

void modify (int x[ ])
{
        int i;
        for(i=0;i<3;i++)
        {
                x [ i ]=9;
        }


}
```

```
void main()
{
        int i, x[10];

        printf("\n Enter numbers to be stored in the array :");
        for(i=0;i<3;i++)
        {
                scanf("%d", &x [ i ] );
        }

        modify(x);
        printf("\n The stored numbers in the array are:");

        for(i=0;i<3;i++)
        {
                printf("\n %d", x [ i ] );
        }
}
```

Prog 10:
Modify prog 4 to sort a list of numbers passing to a function.

1. To search a number in a list of numbers.
2. Modify Problem 1 using UDF.
3. To search a character in an input string.
4. Modify Problem 1 using UDF.
5. To convert a line of text from lowercase to uppercase.
6. Modify Problem 1 using UDF.
7. To search a district whether it is present in Bagmati Zone or not.
8. Modify Problem 1 using UDF.
9. To sort a list of of districts in Bagmati Zone in ascending order.

10. To input a list of words & print them.
11. To sort a list of names in ascending order.
12. To search name of a student, if found, display the name using UDF

Structure

Sometime, it may require to process multiple data stucture whose individual data elements can be differ in type. In a single structure may contain integer element, floating point element and character element. It may consist of arrays, pointers and other structures as element of the structure.

So, a combination of data structure with various type of elements within the same storage class, is known as structure in C.

Defining a structure

It is a little bit difficult to define a structure with comparing to an array. Since in a structure, it may consist of various data elements and each data element is to be defined separately within the structure such as:

```
struct tag {
        member 1;
        member 2;
        member 3;
        …………..
        ………….
        member m;
};
```

where struct is a keyword which defines the following combination is a structure.
tag represents the name of structure of combined members included within compound statement.
and, member 1,member 2, member 3,… ..... member m are the individual declaration of elements of current structure.

- The individual member can be ordinary variable, array, pointer or other structure.
- The name must be distinct from another structure.
- But the member name of the structure and outside the structure may be same but refers to different identity.
- Storage class can not be defined for individual members and also cannot be initialized individual members within the structure.

Once the **composition of structure** has been defined, individual structure type variables can be declared as follows:

storage_class struct tag variable 1, variable 2, variable 3,......... , variable n;

where, storage class is optional.
struct is required keyword to declare following variables as structures.
tag is the name of the structure
and variable 1, variable 2, variable 3 are the structure variables.
e.g.

```
struct account {
        int acctno;
        char acct_type;
        char name[80];
        float balance;
};
```

```
struct account oldcustomer, newcustomer;
```

It is also possible to combine the declaration of structure with structure variables as follows:

```
storage_class struct tag {
        member 1;
        member 2;
        member 3;
        …………..
        ………….
        member m;
}variable 1, variable 2, variable 3, ……., variable n;
```

Note : tag is optional in this case.

```
struct account {
        int acctno;
        char acct_type;
        char name[80];
        float balance;
} oldcustomer, newcustomer;
```

A structure may also consist of another structure as a member of structure but the embedded structure must be declared before the outer structure.
eg.

```
struct date {
        int month;
        int day;
        int year;
};

struct account {
        int acctno;
        char acct_type;
        char name[80];
        float balance;
        struct date lastpayment;
} oldcustomer, newcustomer;
```

The member of a structure can not be initialized within the structure. It can be initialized in the same order as they are defined within the structure as follows:

```
storage_class struct tag variable={value 1, value 2, value 3. ....., value m};
```

e.g.

```
struct date {
        int month;
        int day;
        int year;
};
```

```
struct account {
            int acctno;
            char acct_type;
            char name[80];
            float balance;
            struct date lastpayment;
};

static struct account customer = {101,'S', "Anup", 10000.50, 5, 15, 04};
```

array of structure

It can also be defined an array of structures upon which each element of the array represents a structure.

e.g.

```
struct date {
            int month;
            int day;
            int year;
};

struct account {
            int acctno;
            char acct_type;
            char name[80];
            float balance;
            struct date lastpayment;
}customer[100];
```

In this declaration, each element of array represents a structure of a customer. So, we have 100 structures for 100 customers. That means, we can store 100 records of customer in this data structure.

Processing a structure

The members of a structure are usually processed individually, as separate entities. So, each member of a structure can be accessed individually as follows:

variable.member
where variable refers to name of a structure type variable
and, member refers to name of a member of the structure.

e.g.

```
            customer.acctno
```

where, customer refers to name of structure
and accno refer to member of the structure
similarly,

```
        customer.name
        customer.balance
        etc.....
```

let's see some expressions

```
        ++customer.balance
        customer.balance++
        &customer.balance
```

Similarly, it can also be accessed sub-member of a structure as follows:

variable.member.submember
where variable refers to name of a structure type variable
member refers to name of a member within outer structure
and, submember refers to name of the member within the embedded structure.
e.g.

```
        customer.lastpayment.month
```

```
/* to read input of a customer and write out it's information again */

#include<stdio.h>
struct date {
        int month;
        int day;
        int year;
};

struct account {
        int acctno;
        char acct_type;
        char name[80];
        int balance;
        struct date lastpayment;
}customer[100];

main()
{
        int i,n;
        printf("\n How many no. of cusstomer?");
        scanf("%d",&n);
        for(i=0;i<n;i++)
                readinput();

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

```
void writeoutput(int i)
{
        printf("\n Customer No.: %d",i+1);
        printf("Name : %s",customer[i].name );
        printf("Account Number : %d ",customer[i].acctno );
        printf("Account Type : %c ",customer[i].acct_type );
        printf("Balance : %d ",customer[i].balance );
        printf("Payment     Date     :     %d/%d/%d     ",customer[i].lastpayment.month,
        customer[i].lastpayment.day, customer[i].lastpayment.year);
}

void readinput(int i)
{
        printf("\n Customer No.: %d",i+1);
        printf("Name :  ");
        scanf("%[^n] ",customer[i].name );
        printf("Account Number : ");
        scanf("%d ",&customer[i].acctno );
        printf("Account Type :  ");
        scanf("%c ",&customer[i].acct_type );

        printf("Balance :  ");
        scanf("%d ",&customer[i].balance );

        printf("Payment Date :  ");
        scanf("%d/%d/%d",&customer[i].lastpayment.month,          &customer.lastpayment.day,
&customer.lastpayment.year);
}
```

User Defined data types(typedef)

In c, the data type of any identity can be customized by the user. i.e. new data type can also be made so that such type can be used to define any identities. typedef is the keyword, which enables users to define new data type equivalent to existing data types. Such user defined data types can be used any new variables, arrays, structures. New data type can be defined as follows:

typedef type new_type;

e.g.

typedef int age;	and, is equivalent to
age male female;	int male, female;

Similarly,
typedef float cust[100];
cust newcust,oldcust;

or,
typedef float cust;
cust newcust[100],oldcust[100];

By the same way, it can also be used to define a structure too. It removes the repeatition of struct tag.
In general,

```
typdef struct{
        member 1;
        member 2;
        ...............
        member m;
}new_type;
```

e.g.

```
typedef struct{
        int acctno;
        char acct_type;
        name[80];
        float balance;
}record;
record oldcustomer,newcustomer;
```

Here, record is defined as new structure data type and newly define data type is used to define structure variables oldcustomer and newcustomer.

Different ways of declaration of structures

```
typedef struct {                                              typedef struct {
        int month;            typedef struct {                        int month;
        int day;                      int month;                      int day;
        int year;                     int day;                        int year;
}date;                                int year;               }date;
                              }date;
typedef struct {              typedef struct {                struct {
        int acctno;                   int acctno;                     int acctno;
        char acct_type;               char acct_type;                 char acct_type;
        char name[80];                char name[80];                  char name[80];
        float balance;                float balance;                  float balance;
        date lastpayment;             date lastpayment;               date lastpayment;
}record;                      }record[100];                   }customer[100];
record  customer[100];        record customer;
```

Structures and Pointers

The beginning address of a structure can also be accessed in the same manner as other address, using &(address) operator. So, a variable represents a structure, it can also be represented its address as &variable and pointer to the structure can be denoted as *variable.
such as
type *ptvar;

A pointer to a structure can be defined as follows:

e.g.

```
typedef struct {
        int acctno;
        char acct_type;
        char name[80];
        float balance;
        date lastpayment;
}account;
account customer, *pc=&customer;
```

```
typedef struct {
        int acctno;
        char acct_type;
        char name[80];
        float balance;
        date lastpayment;
} customer, *pc=&customer;
```

Here, customer is a structure variable of type account and pc is a pointer variable whose object is account type structure. The beginning address of the structure can be accessed as
pc=&customer;

Generally, each member of structure can be accessed by using selection operator as

```
        ptvar -> member
```

which is equivalent to
variable.member

The -> operator can be combined to period (.) operator and their associativity is left to right. Similarly, it can be used for array too.
ptvar ->member[expn]

```
typedef struct {
        int month;
        int day;
        int year;
}date;

struct {
        int acctno;
        char acct_type;
        char name[80];
        float balance;
        date lastpayment;
} customer, *pc=&customer;
```

So, if we want to access customer's account number, then we can write any one of these

```
        customer.acctno                pc->acctno            (*pc).acct_no
```

Similary, for month of last payment,

```
customer.lastpayment.month
pc->lastpayment.month
(*pc).lastpayment.month
```

If the structure is defined as:

```
struct {
        int *acctno;
        char *acct_type;
        char *name;
```

```
            float *balance;
            date lastpayment;
} customer, *pc=&customer;
```

So, if we want to access customer's account number, then we can write any one of these

```
            *customer.acctno          *pc->acctno      *(*pc).acct_no
```

struct2.cpp

```
/* To access data items from members of structure */
#include<stdio.h>
#include<conio.h>

void main()
{
int n=111;
char t='c';
float b=99.99;
char name[20]="srijan";
int d=25;

typedef struct {
        int *month;
        int *day;
        int *year;
}date;

struct {
        int  *acctno;
        char *acct_type;
        char *name;
        float *balance;
        date lastpayment;
}customer,*pc=&customer;

pc->acctno=&n;
customer.acct_type=&t;
customer.name=name;
customer.balance=&b;
*customer.lastpayment.day=25;
clrscr();

printf("\n%d      %c      %s      %.2f      %d", *customer.acctno, *customer.acct_type, customer.name,
*customer.balance, *customer.lastpayment.day);

printf("\n%d %c %s %.2f %d",*(*pc).acctno,*pc->acct_type,
        pc->name,*pc->balance,*pc->lastpayment.day);

        getch();
}
```

struct3.cpp

```
/* To determine the size of a structure and it's address */

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

void main()
{

clrscr();

typedef struct {
        int month;
        int day;
        int year;
}date;

struct {
        int  acctno;
        char acct_type;
        char name[80];
        float balance;
        date lastpayment;
}customer, *pc=&customer;


printf("\nNumber of bytes %d",sizeof *pc);
printf("\nstarting address : %d",pc);
printf("\nstarting address : %d",++pc);

getch();
}
```

Passing structure to a function
A structure can generally be passed to a function by two ways. Either, individual members can be passed to a function or entire structure can be passed to a function. By the same way a single member can be returned back to the function reference or entire structure.

In general,
```
void main()
{
typedef struct {
    int month;
    int day;
    int year;
}date;

struct {
    int acctno;
    char acct_type;
    char name[80];
    float balance;
    date lastpayment;
}customer;
float modify(char name[], int acctno, float balance);

....................
customer.balance = modify(name, acctno, balance);
....................
}

float modify(char name[], int acctno, float balance)
{
        float newbalance;
        ...................
        newbalance =......... ;
        return(newbalance);
}
```

In this example, individual member is passed to a function and the new value is returned back to a member of a structure.

struct4.cpp

```
/* To pass an entire structure to a function*/

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

typedef struct {
        int  acctno;
        char acct_type;
        char *name;
        float balance;
}record;

void main()
{
void modify(record *pc);
record customer={101,'c',"Anup",5000.00};

        printf("\n%d %c %s %.2f",customer.acctno,customer.acct_type,customer.name,customer.balance);

        modify(&customer);

        printf("\n\n%d %c %s %.2f", customer.acctno, customer.acct_type, customer.name, customer.balance);

        getch();
}

void modify(record *pc)
{
        pc->acctno=999;
        pc->acct_type='d';
        pc->name="Sabin";
        pc->balance=99999.99;
}
```

In this example, it passes entire structure to the function with it's address, then modifies values of member of the structure and finally returns back to the function reference with modified values of the structure.

```
struct5.cpp

/*to illustrate how an array of structure is passed to a function, and how a
pointer to a particular structure is returned */

#include<stdio.h>
#include<conio.h>
#define N 3

typedef struct {
      int acctno;
      char acct_type;
      char *name;
      float balance;
}record;

void main()
{

static record customer[N]={
                            {101,'c',"Anup",5000.00},
                            {102,'d',"Anil",9000.00},
                            {103,'o',"Sabina",7000.00}
};
      int ano;
      record *pc;
      record *search(record table[N], int ano);

      do{
           printf("\n Enter the record no. to be searched, type 0 to end");
           scanf("%d",&ano);
           pc=search(customer,ano);

           if(pc!=0)
           {
                printf("\n Name : %s",pc->name);
                printf("\n Account No. : %d",pc->acctno);
                printf("\n Account type : %c",pc->acct_type);
                printf("\n Balance : %.2f",pc->balance);
           }
           else
                printf("\n Error - Record not found");
      } while(ano!=0)
}

record *search(record table[N], int ano)
{
      int i;
      for(i=0;i<N;i++)

           if(table[i].acctno==ano)

                return(&table[i]);
```

```
        return(0);
}
```

Unions

It is similar to a structure which may contain individual data item may be different data types of same storage class. Each member of the structure is assigned its own unique storage area in memory whereas all members of a union share the same storage area in memory. It reserves the space equivalent to that of member which requires highest memory. So, it is used to conserve memory. It is useful to such applications, where all members need not to be assigned value at the same time. If assigned, it will produce meaningless result.

In general,

```
union tag {
                member 1;
                member 2;
                ...............
                member m;
                };
```

storage-class union tag variable1, variable2,.......... variable n;

or, combined form,

```
storage-class union tag {
                member 1;
                member 2;
                ...............
                member m;
                }variable1, variable2, ..........variable n;
```

e.g.

```
union id{
                char color[12];
                int size;
        }shirt, blouse;
```

Here we've two union variables, shirt and blouse of type id and occupies 12 bytes in memory, however value is assigned to any union variable.

e.g.

```
union id{
                char color[12];
                int size;
        };

struct clothes{
        char manufacturer[20];
        float cost;
        union id description;
        }shirt, blouse;
```

Here, shirt and blouse are structure clothes type variables. Each variable contains manufacturer, cost and either of color or size.

Each individual member can be accessed in the same way as structure using . and -> operators.

struct6.cpp

```
/* a program using union */
# include <stdio.h>
#include<conio.h>
void main()
{
clrscr();
union id{
                            char color[12];
                            int size;
    };

struct clothes{
    char manufacturer[20];
    float cost;
    union id description;
    }shirt, blouse;

printf("%d",sizeof(union id));

scanf("%s",shirt.description.color);
printf("\n %s %d ", shirt.description.color, shirt.description.size);

shirt.description.size=12;
printf("\n %s %d ", shirt.description.color, shirt.description.size);
getch();
}
```

struct7.cpp

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

typedef union {
    float fexp;                // floating point exponent
    int nexp;                  // integer exponent
}nvals;

typedef struct{
    float x;
    char flag;
    nvals exp;
}values;

void main()
{
    values a;
    float power(values a);
    int i;
    float n,y;

    printf("y=x^n\n enter value of x:");
    scanf("%f",&a.x);
    printf("enter value for n: ");
    scanf("%f",&n);

    i=(int) n;
    a.flag=(i==n) ?'i' : 'f';
    if(a.flag == 'i')
                        a.exp.nexp = i;
    else
                        a.exp.fexp=n;

    if(a.flag=='f' && a.x<=0.0)
    {
                        printf("ERROR cannot raise negative no. to a");
                        printf("floating point power");
    }
    else
    {
                        y=power(a);
                        printf("\n y=%.4f",y);
    }
getch();
}

float power(values a)
{
    int i;
    float y=a.x;
    if(a.flag=='i')
    {
                        if(a.exp.nexp==0)
                                        y=1.0;
                        else
                        {
                                        for(i=1;i<abs(a.exp.nexp);i++)
                                        {
                                           y*=a.x;
                                           if(a.exp.nexp<0)
                                                y=1.0/y;
                                        }
                        }
    }
    else
                        y=exp(a.exp.fexp*log(a.x));  //y=exp(n(log(x)))
    return(y);
}
```

What is a structure? How does a structure differ from an array?
What is member? Write down the relation between member and structure?
What is the purpose of the typedef feature? How is this feature used in conjuction with structures?
What is a union? How does a union differ from a structure?

Self Referential Structures
It is sometimes desirable to include within a structure one member that is a pointer to the parent structure type.

Generally,

```
struct tag {
        member 1;
        member 2;
        ...........
        struct tag *name;
        };
```

where name refers to the name of a pointer variable. Thus the structure of type tag will have a member that points to another structure of type tag. Such structure is known as Self-referential structure.

eg.
```
struct list{
        char item[40];
        struct list *name;
        };
```

array

Some programs may need to handle same type of different number of data having same characteristics. In such situation, it will be easier to handle such data in Array, where same name is shared for all data with different subscripts. In an array, all the data items must be of same type and same storage class. Eg. either int, floating point or characters

Each array element (individual array element) is denoted with a name followed by one or more subscripts (where each subscript must be non negative integers within a pair of square bracket).

The number of subscript depends on the dimensionality of the array. Eg. a[i] refers to an element of one dimensional array. Whereas b[i][j] refers to an array element of two dimensional array. In the same manner c[i][j][k] for three dimensional arry.

Defining an array
Array is also defined in the same manner as ordinary variables, but it should also include the size of specification that shows the maximum size of elements in that array.
It is defined such as follows:

Storage_class data_type array[expression];

Prog 1:
To store integer nos in an array and print them (1 dim)

```
#include<stdio.h>

void main()
{
        int i, x[10]={4,5,2,8,4,7,9,6,5};

        printf("\n The stored numbers in the array are:");

        for(i=0;i<10;i++)
        {
                printf("\n %d", x [ i ] );
        }
}
```

Prog 2:
Modify prog 1 so that it allows to enter integer nos. those are to be stored in an array.

Prog 3:
Modify prog 2 so that it could find out sum, average & deviation of data with average value.

Prog 4:
To sort a list of integer numbers in ascending order.

```
#include<stdio.h>

void main()
{
        int i, x[10];

        printf("\n Enter numbers to be sorted :");

        for(i=0;i<10;i++)
        {
                scanf("%d", &x [ i ] );
        }

        for(i=0;i<9;i++)
        {

                for(k=i+1;k<10;k++)
                {
                        if(x [ i ] > x [ k ]  )
                        {
                                temp = x [ i ];
                                x [ i ]= x [ k ];
                                x [ k ]= temp;
                        }
                }
        }



        printf("\n The sorted numbers are:");

        for(i=0;i<10;i++)
        {
                printf("\n %d", x [ i ] );
        }
}
```

Processing an array
Lllllllll

Two dimensional array
Prog 5:

To store numbers in a two dimensional array and print them

Prog 6:
Modify prog 5 so that it allows to enter numbers.

Prog 7:
Write a program to find out sum of two matrices of size 2x2
Modify it for any size.

Prog 8:
To find out product of two matrices of size 3x2 & 2x1.

Passing array to a function

Prog 9:
To pass a list of numbers into an user defined function to change values of the array.

```
#include<stdio.h>

void modify (int x[ ])
{
        int i;
        for(i=0;i<3;i++)
        {
                x [ i ]=9;
        }


}

void main()
{
        int i, x[10];

        printf("\n Enter numbers to be stored in the array :");
        for(i=0;i<3;i++)
        {
                scanf("%d", &x [ i ] );
        }

        modify(x);
        printf("\n The stored numbers in the array are:");

        for(i=0;i<3;i++)
        {
                printf("\n %d", x [ i ] );
```

```
          }
}
```

Prog 10:
Modify prog 4 to sort a list of numbers passing to a function.

Pointers

Let us assume x is variable of data type int and it's value is 5 then, &x represents its address/location of variable x. If, &x (Here, & is called as address operator) is assigned to another variable px then px is known as pointer of x.

So, a pointer is a variable which represents the location of a data item, such as a variable or an array element.

ie.

```
int x=5;
px=&x;
```

Here, px is known as pointer variable.

The data item represented by x (data item stored in x's memory) can be accessed by the expression *pv where * is a unary operator, called the indirection operator that operates upon a pointer variable. So, x represents its direct value and *px represents its value indirectly.

Advantages of pointers

- can be used to pass information to & fro between a function & its reference point.
- multiple data items can be returned from a function
- requires less memory while using multiple function. So, it makes program execution faster.
- One less subscript can be used to represent multi dimensional array. ie. it permits one less dimension to multi dimensional array.

```
/* A program using pointer*/
```

/* point1.prg*/

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

/* point2.prg: direct & indirect expression */

```
#include<stdio.h>

void main()
{
	int x=5,y;
	int *px;


	y=3*(x+2);

	printf("\n %d",y);

	px=&x;

	y=3*(*px+2);


	printf("\n %d",y);
}
```

/* point3.prg : indirectly changing value*/

```
# include<stdio.h>

void main()
{
	int v=3;
	int *pv;

	pv=&v;
	printf("\n v = %d, *pv = %d", *pv,v);
	*pv=0;
	printf("\n v = %d, *pv = %d", *pv,v);
}
```

Passing pointer to a function

Pointers can also be passed to a function as arguments as other data items. It allows to access any data item to a calling function, then alter within the function and finally returned back to the calling portion of reference function. But the pointer is passed to the function by its address and this method of passing address to a function is known as ***passing by reference***. When pointers are used as formal argument to a function, it should be preceded with indirection operator (asterisk symbol) in each data item.

But the earlier method to pass the value of a data item to a calling function then alter within the function and finally return back the result to the calling portion of the function reference. This method of passing value to a function is known as ***passing by value***.

In this method, it can only be returned a single data item to a calling function using RETURN statement. While passing values to a function, it copies it's value to different location in memory. Whereas by passing reference, it uses same memory allocation and the change of value of any data item, changes to function reference too from that calling portion.

```
/* point4.prg : Difference of passing by value &
passing by reference */

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

void fxbyvalue(int x,int y)
{
	x=0,y=0;
	printf("\n wt1: %d\t%d",x,y);
}

void fxbyreference(int *x,int *y)
{
	*x=0,*y=0;
	printf("\n wt2: %d\t%d",*x,*y);
}

void main()
{
	int x=3,y=5;

	clrscr();

	printf("\n bf1: %d\t%d",x,y);
	fxbyvalue(x,y);
	printf("\n af1: %d\t%d",x,y);

	printf("\n\n bf2: %d\t%d",x,y);
	fxbyreference(&x,&y);
	printf("\n af2: %d\t%d",x,y);

getch();
}
```

/* point5.prg : To count number of different types of characters in a line ***/**

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

void main()
{
clrscr();

char x[80];
int vow=0,con=0,dig=0,whi=0,oth=0;
void analyse(char x[ ],int *vow,int *con,int *dig,int *whi,int *oth);

printf("\n enter a line of text");
scanf("%[^\n]",x);

analyse(x,&vow,&con,&dig,&whi,&oth);
printf("\n vow : %d \n con = %d \n dig=%d\nwhi=%d \n oth=%d",
                        vow,con,dig,whi,oth);
}


void analyse(char x[ ],int *vow,int *con,int *dig,int *whi,int *oth)
{
        char c;
        int i=0;
        while((c=toupper(x[i]))!='\0')
        {
                if(c=='A'||c=='E'||c=='I'||c=='O'||c=='U')
                        ++*vow;
                else if (c>='A'&&c<='Z')
                        ++*con;
                else if (c>='0'&&c<='9')
                        ++*dig;
                else if (c==' '||c=='\t')
                        ++*whi;
                else
                ++*oth;
            i++;
        }
}
```

Pointer and one dim array

If x is a one dimensional array, then first element of array is represented with x[0] and it’s address as &x[0]. By the same way i+1th element is represented as x[i] and it’s address as &x[i]. But it can be represented as x+i and it's value as *(x+i) in pointer.

/* point6.prg*/

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

void main()
{
int x[5]={5,4,6,3,1};
int i;

for(i=0;i<5;i++)
{
	printf("\n %d x[i]=%d *(x+i)=%d &x[i]=%x
(x+i)=%x",i,x[i],*(x+i),&x[i],(x+i));

}
getch();
}
```

Suppose that x is to be defined as a one-dimension, 10 element array of integers. It is possible to define x as a pointer variable rather than as an array. Thus, we can write

```
int *x;
instead of
int x[10];
instead of
#define SIZE 10
int x[SIZE];
```

Dynamic Memory Allocation

However, x is not automatically assigned a memory block when it is defined as a pointer variable, though a block of memory large enough to store 10 integer quantities will be reserved in advance when x is defined as an array. To assign sufficient memory for x, we can make use of the library function malloc, as follows:

```
x= (int *) malloc(n*sizeof(int));
```

This function reserves a block of memory whose size in bytes is equivalent to the sizeof an integer quantity. It returns a pointer to x.

The allocation of memory in this manner, as it is required, is known as dynamic memory allocation.

```
/*reorder a one dim integer array in ascending order using
pointer notation*/

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

void main()
{
clrscr();
int i,n,*x;
void sort(int n,int *x);

printf("Enter how many nos. ");
scanf("%d",&n);
x=(int *)malloc(n*sizeof(int));

for(i=0;i<n;i++)
        scanf("%d",x+i);

sort(n,x);

for(i=0;i<n;i++)
        printf("\n%d",*(x+i));

getch();
}

void sort(int n,int *x)
{
        int i,j,temp;
        for(i=0;i<n-1;i++)
        {
                for(j=i+1;j<n;j++)
                {
                        if(*(x+i)>*(x+j))
                        {
                                temp=*(x+i);
                                *(x+i)=*(x+j);
                                *(x+j)=temp;
                        }
                }
        }
}
```

Pointers and Multidimensional arrays

A two dimensional array is actually a collection of one dimensional arrays. So, we can define two dimensional arrays as a pointer to a group of continuous one dim array. It can be written as follows:

Data_type (*ptvar)[experession 2];
In stead of
Data_type array [expression 1][expression 2];

And, same for more

Data_type (*ptvar)[expn 2][expn 3]……….[expn n];
In stead of
Data_type array [expn 1][expn 2][expn 3]……….[expn n];

e.g.
int (*x)[20]   for int x[10][20];

int (*y)[20][30]   for int y[10][20][30];

Suppose that x is a 2 dim integer array have 10 rows and 20 columns, the item in row 2 and column 5 can be access as follows:

X[2][5]
Or *(*(x+2)+5)

Here, (x+2) is a pointer to the row 2, so the object of this pointer is *(x+2). If 5 is added to this pointer, (*(x+2)+5) points the address of 5th element of 2nd row and *(*(x+2)+5) points to object/value of 5th element of 2nd row.

Pstrsort.cpp

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

void reorder(int n, char (*x)[20])
{
	int i,j;
	char *temp;
	for(i=0;i<n-1;i++)
	{
		for(j=i+1;j<n;j++)
		{
			if(strcmpi(*(x+i),*(x+j))>0)
			{
				strcpy(temp,*(x+i));
				strcpy(*(x+i),*(x+j));
				strcpy(*(x+j),temp);
			}
		}
	}
}

void main()
{
clrscr();
int n=0,i;
char (*x)[20];
do{
      printf("string %d: ",n+1);
      scanf("%s",x+n);
}while(strcmpi(*(x+n++),"End")!=0);

reorder(--n,x);

printf("\n\n Reordered list\n");
for(i=0;i<n;i++)
{
      printf("\n string %d : %s",i+1,*(x+i));
}

getch();
}
```

Arrays of Pointers

A multidimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays. In such situations the newly defined array will have one less dimension than the original multidimensional array. Each pointer will indicate the beginning of a separate (n-1) dimensional array.

In general terms, a two dimensional array can be defined as one dim array of pointers by writing
Data_type *array[expn1];
In stead of

Data_type array [expression 1][expression 2];

And, same for more

Data_type *ptvar[expn 1][expn 2]……….[expn n-1];
In stead of
Data_type array [expn 1][expn 2][expn 3]……….[expn n];

e.g.
int *x[10]   for int x[10][20];

int *y[10][20] for int y[10][20][30];

Suppose that x is a 2 dim integer array having 10 rows and 20 columns, the item in row 2 and column 5 can be access as follows:

X[2][5]
Or *(*(x+2)+5)

Program :To find sum of two matrices using array of pointers.

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#define row 20

void readinput(int *x[row],int m,int n)
{
      int r,c;
```

```
	for(r=0;r<m;r++)
	{
		printf("\n Enter data for row no. %d",r+1);
		for(c=0;c<n;c++)
			scanf("%d",(*(x+r)+c));
	}
}

void computesum(int *x[row],int *y[row],int *z[row],int
m,int n)
{
	int r,c;
	for(r=0;r<m;r++)
	{
		for(c=0;c<n;c++)
			*(*(z+r)+c)=*(*(x+r)+c)+*(*(y+r)+c);
	}
}

void writeout(int *x[row],int m,int n)
{
	int r,c;
	for(r=0;r<m;r++)
	{
		for(c=0;c<n;c++)
			printf("%d\t",*(*(x+r)+c));

		printf("\n");
	}

}
```

```
void main()
{
clrscr();
int nrows,ncols,r;
int *x[row],*y[row],*z[row];
printf("How many rows?");
scanf("%d",&nrows);
printf("How many cols?");
scanf("%d",&ncols);
for(r=0;r<=nrows;r++)
{
x[r]=(int *) malloc(ncols*sizeof(int));
y[r]=(int *) malloc(ncols*sizeof(int));
z[r]=(int *) malloc(ncols*sizeof(int));
}


printf("\n First table\n");
readinput(x,nrows,ncols);

printf("\n second table\n");
readinput(y,nrows,ncols);
computesum(x,y,z,nrows,ncols);

printf("\n Sum of elements\n");
writeout(z,nrows,ncols);
getch();
}
```

More about pointer declarations

A pointer can be declared in different ways and there is difficulty of dual use of parentheses. Generally, parentheses are used to indicate function, and they are used for nesting purpose for precedence within more complicated declarations.

Thus, the declaration

(int *)p(int a);

Indicates a function that accepts an integer argument, and returns a pointer to an integer.

Whereas,

int (*p)(int a);

Indicates a pointer to a function that accepts an integer argument and returns an integer. In this case, the first pair of parentheses is used for nesting and the second pair is used to indicate function.

A pointer can be declared more complex too. Such as,

Int (*p)(int (*a)[]);

It can be interpreted as follows. In this declaration, (*p) (….) indicates a pointer to a function. So, int (*p) (….) indicates pointer to a function that returns an integer quantity.

Within the last pair of parentheses, (*a)[ ] indicates a pointer to an array.

As a result, int (*a)[ ] represents a pointer to an array of integers.

Keeping the pieces together, (*p) (int (*a)[ ]) represents a pointer to a function whose argument is a pointer to an array of integers. And finally, the entire declaration

```
int (*p) (int (*a)[ ]);
```

```
int *p;
int *p[10];
int (*p)[10];
int *p(void);
int p(char *a);
(int *) p(char *a);
int p(char (*a)[]);
```

Write program to solve following problems with function with pointer

1. to find real root of a quadratic equation
2. to calculate the factorial of N
3. to determine value of x^n
4. to find square root of x
5. to find product of two matrices
6. to sort a list of numbers in ascending order
7. What is a pointer? What is the relationship between the address of a variable v and corresponding variable pv? Write down the advantages of using pointer.

Introduction of Computer

Computer is an electronic device. It takes raw data as input, processes it and gives information as output, all under instruction given to it.

e.g. a =5, b=8

c=a+b

whereas, a and b are data those we input and value of c is information/result.

Weakness:

It's a dull machine. That means, it cannot do anything of its own because of lack of intelligence.

Software:

A computer contains two basic parts: (i) Hardware and (ii) Software. Without software a computer will remain just a metal. With software, a computer can store, retrieve, solve different types of problems.

A software can be defined as : It's a set of instructions, arranged in a such a way to do some useful work.

There are mainly two types of Software:

1) System Software :

This type of software deals with system/hardware. It enables the system to operate, translates the codes in different computer language to machine understandable codes, or utilities for various systems. There are further classification of system software, such as

a) Operating System

An operating system (OS) is the most important system software and is a must to operate a computer system. An operating system manages a computer's resources very effectively, takes care of scheduling multiple jobs for execution and manages the flow of data and

instructions between the input/output units and the main memory. Operating system became a part of computer software with the second generation computers. Since then operating systems have undergone several revisions and modifications in order to achieve a better utilisation of computer resources. Advancement in the field of computer hardware, have also helped in the development of more efficient operating systems.

Some important features of operating systems are:

(1) It interacts with the user.
(2) It controls the system including peripherals.
(3) It manages user files and other applications.

e.g. MSDOS, PCDOS, Windows95, windows98, windowsme, windows xp, linux, unix etc.

Operating System Commands : Apart from system calls, users may interact with operating system directly by means of operating system commands. For example, if you want to list files or sub-directories in MS-DOS, you invoke dir command. In either case, the operating system acts as an interface between users and the hardware of a computer system lie fundamental goal of computer systems is to solve user problems. Towards this goal, computer hardware is designed. Since the bare hardware alone is not very easy to use programs (software) are developed. These programs require certain common operations, such as controlling peripheral devices. The command function of controlling and allocating resources are then brought together into one piece of software; the operating system.