Introduction

Types of Programming Languages:

There are many different languages that can program a computer. The most basic of these is the **machine language** – a collection of very detailed, cryptic instructions that control the computer's internal circuitry. A machine language program written for one computer cannot be run on another computer without significant changes. Usually, a computer program is written in some **high-level language**, whose instruction set is more compatible with human languages and human thought processes.

As a rule, a single instruction in a high-level language will be equivalent to several instructions in machine language. The rules for programming in a particular high-level language are much the same for all the computers, so that a program written for one computer can generally be run on different computers with little or no changes. Thus a high-level language offers 3 significant advantages over machine language – simplicity, uniformity and portability (machine independence).

Compilation

A program written in high-level languages must be translated into machine language before it can be executed. This is known as **compilation** or **interpretation**. **Compilers** translate the entire program into machine language before executing any of the instructions. **Interpreters** proceed through a program by translating and then executing single instructions or small group of instructions.

A compiler or interpreter is itself a computer program. It accepts a program written in high-level language as input, and generates a corresponding machine-language program as output. The original high-level language is called the **Source program**, and the resulting machine language is called as **Object program**.

Introduction to C++

C++ is a general purpose programming language with a bias towards systems programming that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming**.**

C++ has been standardized by ANSI (The American National Standards Institute), BSI (The British Standards Institute), DIN (The German national standards organization), several other national standards bodies, and ISO (The International Standards Organization)

C is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system. It has since spread to many other platforms, and is now one of the most widely used programming languages. C has also greatly influenced many other popular languages, especially C++, which was originally designed as an enhancement to C. It is the most commonly used programming language for writing system software, though it is also widely used for writing applications.

C++ is a general-purpose, high-level programming language with low-level facilities. It is a statically typed free-form multi-paradigm language supporting procedural programming, data abstraction, object-oriented programming, generic programming and RTTI. Since the 1990s, C++ has been one of the most popular commercial programming languages. Initially there was a language stimula in 1970s at Cambridge University. Bjarne Stourpus modified it and implemented it with C in 1978 at Bell Laboratories and named it as **C++**.

The name "C++"

This name is credited to Rick Mascitti (mid-1983) and was first used in December 1983. Earlier, during the research period, the developing language had been referred to as "new C", then "C with Classes". In computer science C++ is still referred to as a superstructure of C. The final name stems from C's "++" operator (which increments the value of a variable) and a common naming convention of using "+" to indicate an enhanced computer program. According to Stroustrup: "the name signifies the evolutionary nature of the changes from C". C+ was the name of an earlier, unrelated programming languag

Features of C++

- **high Level Language**: C++ combines the features of high-level languages as well as machine languages. So, it is suited for systems programming (creating operating systems) as well as application programming.
- **Efficiency and Speed**: This is due to its variety of data types and powerful operators. There are only 32 keywords in C++ and its strength lies in its built-in functions.
- **Portability**: C++ programs written for one computer can be run on another computer with little or no modification.
- **Extendable**: A C++ program is basically a collection of functions that are supported by the C++ library. We can add our own functions to the C++ library.
- **Case-sensitive**: C++ is a case-sensitive language, i.e., the functions in C++ are used in lower case only. Variables and constants declared must be used with the same name and case as given in the declaration.

Basic Structure of a C++ Program

Documentation Section
Link Section
Definition Section
Global Declaration Section
Function Section
- Declaration Part
- Executable Part

Sub-program Section
User-defined Functions

- **Documentation Section**: It consists of a set of comment lines giving details about the name of the program, its author, etc. Comment lines are not executable statements and therefore ignored by the compiler. Single line comments begin with //. Multiple lines comments begin with /* and end with */.

- **Link Section**: It provides instructions to the compiler to link functions from the system library.
- **Definition Section**: It defines all symbolic constants.
- **Global Declaration Section**: It defines global variables that can be used in more than one function.
- **Function Section**: Every C++ program has the main ( ) function section. All statements in the Function section must end with a semi-colon. This is the only compulsory section in a C++ program.
  - **Declaration Section**: It declares all variables used in the executable part.
  - **Executable Section**: It has at least one statement.
- **Sub-program Section**: It contains all user-defined functions that are called in the main () function. Statements in the sub-program section also end with a semi-colon.

Four Steps in Executing a C++ Program are

1. Creating
2. Compiling
3. Linking
4. Executing

Character Set of C++: Characters in C++ can be grouped into:

- **Letters**: Uppercase A-Z or lowercase a-z
- **Digits**: 0-9
- **Special Characters**:

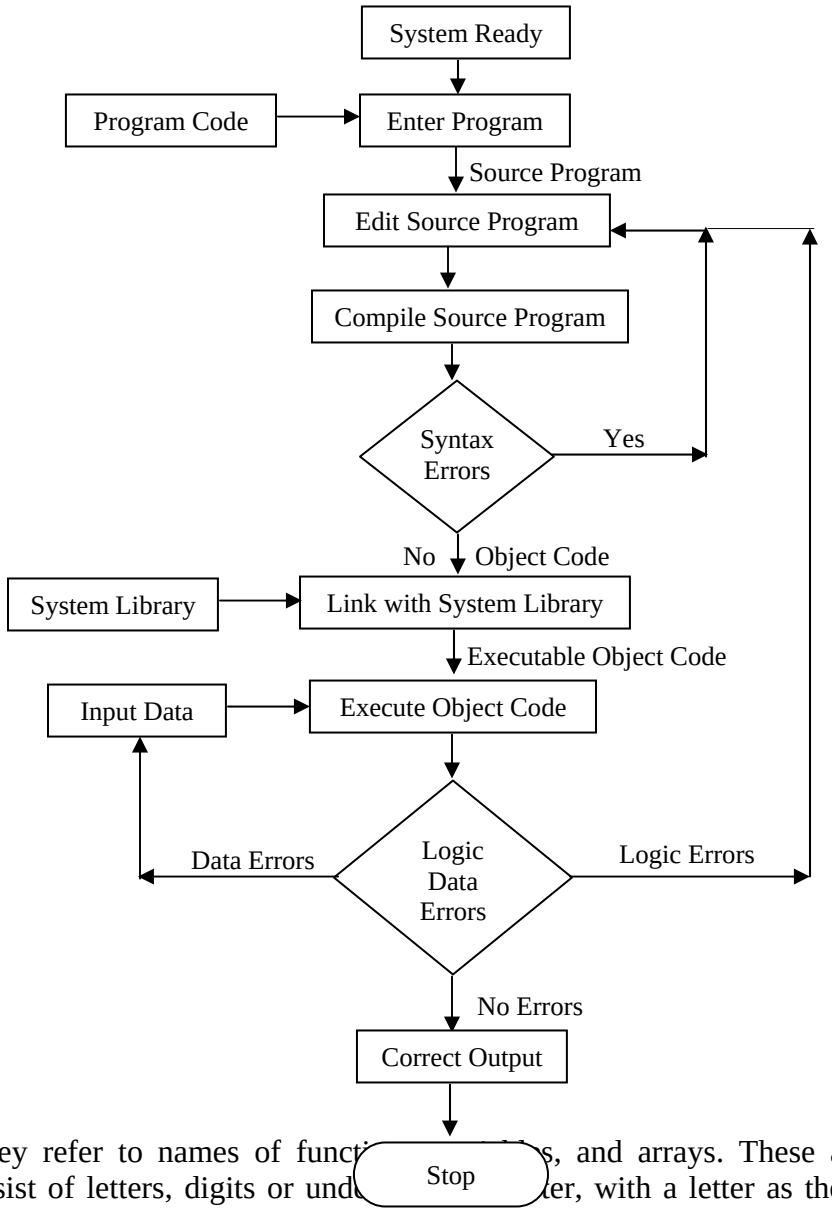
, Comma	. period	; semicolon	: colon	? question mark
‘ apostrophe	“ quotation mark	! exclamation mark	| vertical bar	/ slash
\ backslash	~ tilde	_ underscore	$ dollar sign	% percent sign
# number sign	ampersand	^ caret	* asterisk	- minus sign
+ plus sign	< opening angle bracket	< closing angle bracket	(left parenthesis	) right parenthesis
[left bracket	] right bracket	{ left brace	} right brace	

- **White Space**: Blank space / Horizontal Tab / New Line / Form Feed

Keywords:

Auto	double	Int	struct
Break	else	long	switch
Case	enum	register	typedef
Char	extern	return	union
Const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
Do	If	static	while

Executing a C++ Program



Identifiers: They refer to names of functi[illegible]s, and arrays. These are user-defined names and consist of letters, digits or und[illegible]er, with a letter as the first character. They cannot be the same as any of the keywords.

Basic Data Types

When programming, we store the variables in our computer's memory, but the computer has to know what we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer

can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

	Description	Size*	Range*
Char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
Int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
Bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
Float	Floating point number.	4bytes	3.4e +/- 38 (7 digits)
Double	Double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
Wchar_t	Wide character.	2bytes	1 wide character

To provide control over the range of numbers and storage space, C++ has 3 classes of integer storage – **short int, int** and **long int** in both **signed** and **unsigned** forms. For e.g., short int represents fairly small integer values and requires half the storage as a regular int number uses. Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive. The use of qualifier signed on integers is optional as the default declaration assumes a signed number.

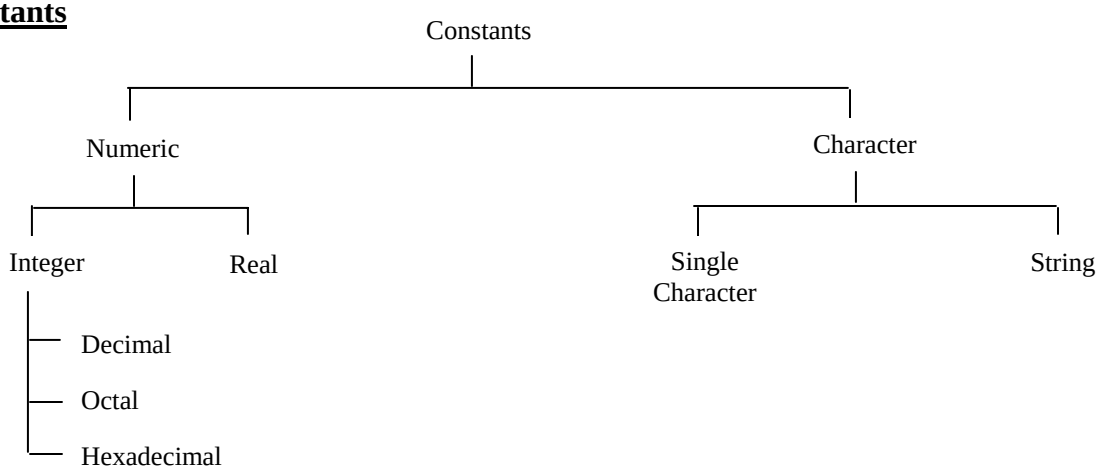
Escape Sequences / Backslash Character Constants: They are used to display characters that cannot be entered through the keyboard.

\a	Audible alert (Bell)	\’	Single quote (Apostrophe)
\b	Backspace	\”	Double quotes
\f	Form feed (Page break)	\?	Question mark
\n	New line	\\	Backslash
\t	Horizontal tab	\0	Null

Shortcut keys for the Editor

F2 – Save File	**F3** – Open File	**Alt + F3** – Close File	**F5** – Toggle to zoom window
Alt + F5 – Display user screen	**F6** – Switch between windows	**F9** – Make	**Alt + F9** – Compile
Ctrl + F9 – Compile & Execute	**Shift + Del** – Cut	**Ctrl + Ins** – Copy	**Shift + Ins** – Paste
Ctrl + Del – Delete selected text	**Shift + Arrow keys** – Select text	**Alt + Bksp** – Undo	**Alt + X** - Exit

Constants



Decimal: Digits from 0 to 9 and optional leading sign (+ / -)
Octal: Digits from 0 to 7, with a leading 0 (zero)
Hexadecimal: Digits from 0 to 9 and letters from a-f, preceded by 0x or 0X.
Real: Floating point numbers
Single Character: Enclosed in single quotes
String: Multiple characters – enclosed in double quotes.

Pre-processor Directives

The pre-processor is a unique feature of c that is unavailable in other high-level languages. It is a program that processes the source code before it passes through the compiler. It operates under the control of pre-processor command lines or directives. Pre-processor directives are placed in the source program before the main ( ) line. Before ht source code passes through the compiler, it is examined by the pre-processor for any pre-processor directives. If there are any, appropriate actions are taken and then the program passes on to the compiler. Pre-processor directives begin with # and do not require a semi-colon at the end

#define – Defines a symbolic constant
#include – Specifies header files to be included

Symbolic Constants

It is a name that substitutes for a value. The value could be a numeric, character or string constant.
Syntax:
#define <*name*> <*value*>
Symbolic names are usually given in uppercase to differentiate them from variables.

Programs

1. **Program to display a message**

```
#include<iostream.h>
#include<conio.h>
main ( )
{
    clrscr ( );
    cout<<"Hello World";
      cout<<"\n Have a good day";
    getch ( );
}
```

Notes:

i. iostream.h: Standard Input / Output Header file, which includes the functions related to standard input and output of data.
ii. conio.h: Console Input / Output Header file, which includes the functions related to the screen (Console means screen).
iii. clrscr ( ): This function is included in conio.h and is used to clear the screen.
iv. cout: Print Formatted – This function is included in istream.h and is used to display formatted data on the screen
v. getch ( ): Get character – This function is included in conio.h and accepts a single character which is not displayed not the screen. Usually this is given as the last statement in the program so that the user can see the output before returning to the editor (Otherwise the output can be displayed using Alt + F5).
vi. All the statements in the main ( ) function are enclosed in { }.
vii. the main is the function which has return type if does not return any value it is declared as void.

2. **Program to accept and add two numbers**

```
#include<iostream.h>
#include<conio.h>
void main ( )
{
    int n1, n2, s=0;
    clrscr( );
    cout<<"Enter two numbers:";
    cin>>n1>>n2;
    s = n1+n2;
    cout<<"\n Sum = "<<s;
    getch ( );
}
```

Notes: cout<<-same like printf use to print line.also known as exertion.
cin>>-same like scanf use as access specifier also known as insertion.

3. **Program to accept radius of a circle and display the circumference and area**

```
#include< iostream.h >
#include<conio.h>
#define PI 3.14 ⟶ Symbolic Constant
main ( )
{
    int r;
    float a,c;
    clrscr ( );
    cout<<"Enter the radius of a circle:";
    cin>>r;
    c=2*PI*r;
    a=PI*r*r;
    cout<<"\n Circumference = "<<c;
    cout<<"\n Area ="<<a;
    getch ( );
}
```

ASSIGNMENT:

1) write a program to swap two numbers using third variable and without third variable

```
void main()
{
  int a ,b,c;
  printf("enter the values of a and b");
 scanf("%d%d",&a,&b);
 c=a;   /* a=a+b;
 a=b;       b=a-b;
 b=c;       a=a-b;     without third variable */
 printf("the swapped values are%d%d",a,b);
}
```

Operators

Operator is a symbol that tells the compiler to perform some mathematical or logical manipulation.

Arithmetic Operators

+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Modulo Division (i.e. returns the remainder after dividing two numbers)

For e.g. a =10, b=3

a%b = 1

b%a = 3 *If numerator is less than the denominator, returns the remainder as it is without dividing*

a/b = 3 *Decimal part will be truncated*

Unary Operators: Changes the sign of a number.

+	Unary Plus
-	Unary Minus

Assignment Operators: Used to assign a value to a variable.

=

Relational Operators

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Logical Operators

&&	And
||	Or

Shorthand Assignment Operators

+=	Increase the value of the variable on the left by the value on the right
-=	Decrease the value of the variable on the left by the value on the right
/=	Assign the value of the division to the variable on the left
*=	Assign the value of the multiplication to the variable on the left
%=	Assign the value of the modulo division to the variable on the left

Increment / Decrement Operator: Increases / decreases the value of the variable by 1.

sizeof Operator

It is a compile time operator. When used with an operand, it returns the number of bytes the operand occupies. The sizeof operator is normally used to determine the lengths of arrays and structures when their size is not known.

4. Program to illustrate increment / decrement operators

```
#include<iostream.h>
#include<conio.h>
main ( )
{
    int a=1,b=10;
    clrscr ( );
    cout<<"\n a= , b="<<a<<b;
    cout<<"\n a= , b="<<a++<<b--;
    cout<<"\n a= , b="<<a<<b;
    cout<<"\n a= , b="<<++a<<--b;
    cout<<"\n a= , b="<<a<<b;
}
```

Note:
a++ is post-increment, where current value of a is used and then incremented by 1.
++a is pre-increment where the value is first incremented by 1 and then used.
The same rule applies for the decrement operator as well.

5. **Shorthand Assignment Operator**

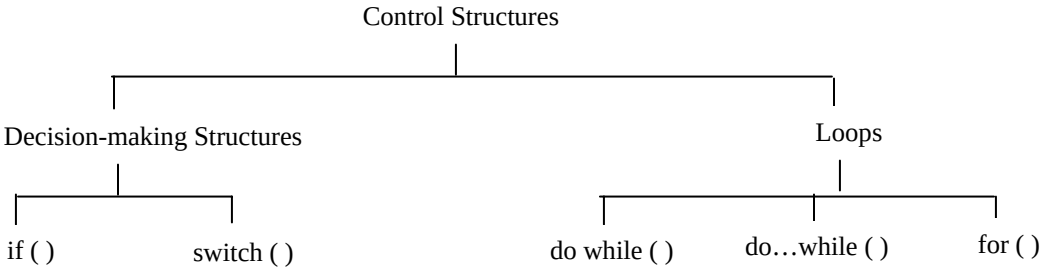
```
#include<iostream.h>
#include<conio.h>
void main()
{
    int a=5, b=10, c=20, d=40, x=10,y=3;
    clrscr ( );
    a+=b;  ------------------------>  a=a+b =5+10=15
    cout<<"\na="<<a;
    b-=y;  ------------------------>  b=b-y=10-3=7
    cout<<"\nb="<<b;
    c*=x ;------------------------>  c=c*x=20*10=200
    cout<<"\nc="<<c;
    d/=x;  ------------------------>  d=d/x=40/10=4
    cout<<"\nd="<<d;
    x%=y;------------------------>  x=x%y=10%3=1
    cout<<"\nx="<<x;
    getch();
}
```

Arithmetic Operations on Characters

Whenever a character constant or character variable is used in an expression, it is automatically converted to an integer value by the system. The integer value depends on the local character set.
E.g.

```
x='a';
cout<<x;        ---->  97
x='z'-1;------>        121
```

Control Structures



if ( )

Syntax:

- if (*condition*)
  {
  statement_block;
  }

- if (*condition*)
  {
  statement_block_if_true;
  }
  else
  {
  statement_block_if_false;
  }

- if (*condition1*)
  {
  statement_block_1;
  }
  else if (*condition 2*)
  {
  statement_block_2;
  }
  ⋮
  else
  {
  default_statement_block;
  }

Note: The braces are required only if multiple statements are to be executed, not otherwise.

Conditional Operator

Syntax**:**

(*condition*) **?** *statement_if_true* **:** *statement_if_false*;

Note: Only one statement can be executed, depending on whether the condition is true/ false.

6. If Statement

```
#include<iostream.h>
void main()
{
    int n1,n2;
    cout<<"Enter first number\t";
    cin>>n1;
    cout<<"\nEnter second number\t";
    cin>>n2;
    if (n1 == n2)
            cout<<"\nThe nos are equal";
    else
            cout<<"\nThe nos are unequal";
}
```

7. Determining whether the number is odd / even

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n;
    cout<<"Enter a number\t";
    cin>>n;
    if ((n%2)== 0)
            cout<<"\nEven number";
    else
            cout<<"\nOdd number";
}
```

8. If – else if structure

```
#include<iostream.h>
#include<conio.h>
void main()
{
   int n1,n2;
   cout<<“Enter first number\t”;
   cin>>n1;
   cout<<“\nEnter second number\t”;
   cin>>n2;
   if (n1 > n2)
        cout<<“Greater number = ”<<n1;
   else if (n1 < n2)
        cout<<“Greater number = ”<<n2;
   else
         cout<<“The nos are equal”;
}
```

9. If – else if structure with logical operator

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n1,n2,n3;
    clrscr ( );
    cout<<“Enter 3 numbers\t”;
    cin>>n1>>n2>>n3;
    if (n1>n2 && n1>n3)
         cout<<“Greatest number = ”<<n1;
    else if (n2 < n3)
        cout<<“Greatest number = ”<<n2;
    else
        cout<<“Greatest number = ”<<n3;
    getch();
}
```

Assignment – Accept the marks of 5 subjects and allocate grades as : Percentage >=90 – Outstanding, 80-90 – Excellent, 70-80 – Good, 60-70 – Average, 50-60 – Satisfactory, <50 – Fail

```
    {
        float m1,m2,m3,m4,m5,t,p;
        clrscr ( );
        //Accept the marks
        t = m1 + 2 + m3 + m4 + m5;
        p = (t * 100) / 500;
        //Display total  percentage – t (%.0f), p(%.2f)
        if (p >= 90)
                cout<<“\nGrade – Outstanding”;
        else if(p<90  p>80)
                cout<<“\n Grade – Excellent”;
        ⋮
        getch();
    }
```

10. Conditional Operator

```
    #include<iostream.h>
    #include<conio.h>
    void main()
    {
        int n1,n2;
        cout<<“Enter first number\t”;
        cin>>n1;
        cout<<“\nEnter second number\t”;
        cin>>n2;
        (n1 == n2)?cout<<“\nThe nos are equal”:cout<<“\nThe nos are unequal”;
    }
```

switch ( )

Syntax:

```
switch ( expression)
{
    case value 1:
        statement_block_1;
        break;
    case value 2:
        statement_block_2;
        break;
    .
    .
    .
    default:
        default_statement_block;
}
```

Note: Each case ends with break statement; otherwise all the cases will be executed. The default case does not require a break statement.

11. Switch Operator

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n1,n2,n3;
    char op;
    clrscr ( );
    cout<<"Enter two numbers\t";
    cin>>n1>>n2;
    cout<<"\nEnter the operator\t";
    cin>>op;

    switch(op)
        {
            case '+': n3=n1+n2;
                 cout<< n3;
                 break;
            case '-': n3=n1-n2;
                cout<< n3;
                break;
            case '*': n3=n1*n2;
                cout<< n3;
                break;
            case '/': n3=n1/n2;
                 cout<< n3;
                 break;
            case '%': n3=n1%n2;
                cout<< n3;
                break;
            default:
                cout<<"\n Not a valid operator";
        }
    getch();
}
```

Note: If you accept a single character as the second value, then precede the %c format specifier with a space; otherwise the character for Enter key ('\n') will be stored as the value. To ignore \n and then allow the user to enter a character, precede %c with a space.

12. Accepting a character and checking whether it is a vowel or a consonant

```
#include<iostream.h>
#include<conio.h>
void main()
{
    char ch;
    clrscr ( );
    cout<<"Enter a character\t";
    cin>>ch;

switch(ch)
    {
       case 'a':
       case 'A':
       case 'e':
       case 'E':
       case 'i':
       case 'I':
       case 'o':
       case 'O':
       case 'u':
       case 'U':
            cout<<"\n It is a vowel";
            break;
       default:
            cout<<"\n It is a consonant;
    }
    getch();
}
```

Note: If the statements to be executed are common for the cases, it need not be repeated several times. Instead all the cases can be clubbed together, and then the statements to be executed can be specified.

13. Assignment – Accept a number from the user. If the number is from 1-12, display the corresponding month name, else display the message – 'Error'.

```
#include<iostream.h>
#include<conio.h>
void  main()
{

    int n;
    cout<<"Enter a numberfrom 1-12\t";
    cin>>n;
    switch(n)
    {
    case 1:
                cout<<"\nJanuary";
                break;
    case 2:
                cout<<"\nFebraury";
                break;
    .
    .
    .

     default:
                 cout<<"\nError";
                }
}
```

14. **menu driven – program to add,sub,mul,div.**

```
#include<iostream.h>
#include<conio.h>
void  main()
{
     int n1,n2,n3, s;
     clrscr();
    cout<<"1>add \n 2>sub \n 3>mul \n 4>div";
    cout<<"enter the choice";
    cin>>s;
    cout<<"enter the value of a and b";
    cin>>n1>>n2;
   switch(s)
   {
     case 1: n3=n1+n2;
              cout<<"the add is "<<n3;
             break;
     case 2: n3=n1-n2;
              cout<<"the sub is "<<n3;
             break;
    case 3: n3=n1*n2;
              cout<<"the mul  is "<<n3;
              break;
     case 4: n3=n1/n2;
              cout<<"the div is "<<n3;
              break;
     default: cout<<"invalid case";
   }
  getch();
  }
```

15. **Assignment - program to find area of circle,triangle,rectangle using menu driven**

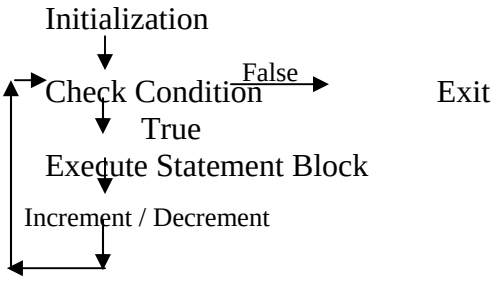
```
#include<iostream.h>
#include<conio.h>
void  main()
{
 int s;
clrscr();
cout<<"1>circle \n 2> triangle \n 3> rectangle";
cout<<"enter the choice";
cin>>s;
switch(s)
{
 case 1: float a,r;
          cout<<"enter the radius";
          cin>>r;
          a= 3.14*r*r;
          cout<<"the area is "<<a;
          break;
.
.
default: cout<<"invalid case";
}
getch();
}
```

while ( )

Syntax:

while (*condition*)
{
    statement_block;
}

Order of Execution:



16. **Printing first 10 numbers using while loop**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n = 1;
    clrscr ( );
    while (n<=10)
    {
            cout<<"\n"<<n;
     n++;
     }
    getch();
}
```

17. **Printing first 10 odd or even numbers**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n = 1,ctr=1;
    //int n = 2,ctr =1;
    clrscr ( );
    while (ctr<=10)
    {
            cout<<"\n"<<n;
            ctr ++;
            n+=2;
    }
    getch();
}
```

18. **Printing sum of first 10 numbers using while loop**

```
#include<iostream.h>
void main()
{
        int n = 1,sum=0;
        while (n<=10)
        {
                sum+=n;
                n++;
        }
          cout<<"\nSum of first 10 numbers = "<<sum;
}
```

19. **Factorial of a number using while loop**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n;
    long int  f  = 1;
    clrscr ( );
    cout<<"Enter a number\t";
    cin>>n;
        while (n>0)
    {
            f*=n;
            n--;
    }
    cout<<"\nFactorial = "<< f;
    getch();
}
```

20. **Printing a table of temperature in Celsius and Fahrenheit**

```
#include<iostream.h>
#include<conio.h>
#define F_LOW 0
#define F_MAX 250
#define STEP 25
void main()
{
    float f,c;
    clrscr ( );
    f = F_LOW;
    cout<<"Fahrenheit \t\t Celsius \n\n";
    while (f <= F_MAX)
    {
            c =(f-32.0)/1.8;
            cout<<f<<c;
            f += STEP;
    }
    getch();
}
```

21. **Assignment – Accept 5 numbers and display their sum (using while loop)**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n,ctr=1, sum=0;
    clrscr ( );
    while (ctr <= 5)
    {
            cout<<"\nEnter a number\t";
            cin>>n;
            sum+=n;
            ctr++;
    }
    cout<<"\n\nSum of the numbers = "<< sum;
    getch();
}
```

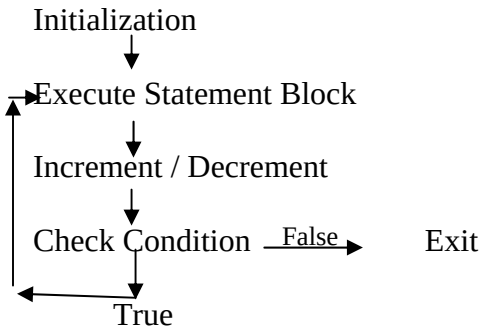
do…while ( )

Syntax:

```
do
{
    statement_block;
}
while (condition);
```

Note: As condition is evaluated at the end of the loop, statement block is executed at least once.

Order of Execution:



22. **Difference between while and do...while**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int i=50;
    clrscr ( );
    while (i<=10)
    //do
    {
            cout<<i;
            i++;
    }
    //while(i<=10);
    }
```

23. **Displaying sum of (1/5)+(3/6)+…+(21/15)**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    float i =1, j=5, s =0;
    clrscr ( );
    do
    {
        s += (i/j);
        i+=2;
        i++;
    }while (i <= 21);
    cout<< s;
    getch();
}
```

24. **Assignment – Sum of squares of first 10 numbers using do…while loop**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n=1, s=0;
    clrscr ( );
    do
    {
            s+=n*n;
            n++;
    }while (n<=10);
    cout<<"\n "<<s;
    getch();
}
```

for ( )

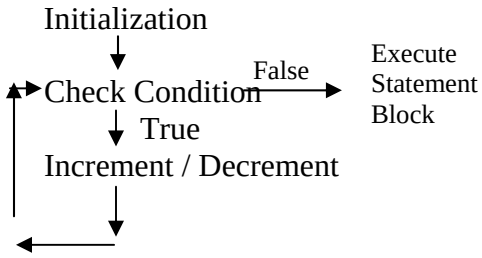
Syntax:

for (*initialization*; *condition*; *incre / decre*)
{
    statement_block;
}

for (*initialization*; *condition*; *incre / decre*);
{
    statement_block;
}

Order of execution:

Same as while ( ) loop



Note: In case of for ( ) loop with a semi-colon at the end, the statement block is executed only once when the condition evaluates to false.

25. **Displaying numbers in reverse order using for loop**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n;
    clrscr ( );
    for(n=10; n>0;n--)
    {
            cout<<"\n"<<n;
    }
    getch();
}
```

26. **Accept a number and display its table**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int  x, y,z;
    clrscr ( );
    cout<<"Enter a number\t";
        cin>>x ;
    for (y=1; y<=10;y++)
    {
        z=x*y ;
        cout<< z<< "\n";
    }
    getch();
}
```

27. Displaying the fibonacii series using for loop

```
#include<iostream.h>
#include<conio.h>
void main()
{
        int i, n, a=1, b=1, s;
    clrscr ( );
    cout<<“Enter a number\t”);
    cin>>n;
    cout<<“\n” <<a;
    cout<<“\n\n”<< b;
    for (i=0; i<=n-2,i++)
        {
            s = a+b;
            b = a;
            a = s;
            cout<<“\t”<<s;
        }
        getch();
}
```

28. Nested loops

```
#include<iostream.h>
#include<conio.h>
#define ROWMAX 12
#define COLMAX 10
void main()
{
    int r =1,c,n;
    while(r <= ROWMAX)
    {
            for(c=1;    c<=COLMAX;
c++)
            {
                    n=r*c;
                    cout<<n;
            }
            cout<<“\n”;
            r++;
        }
        getch();
}
```

29. Comma operator with for loop

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n, ctr;
    clrscr ( );
    for (n=2, ctr=1; ctr<=10; n+=2,ctr++)
            cout<< ctr<<"\t"<< n<<"\n";
    getch();
}
```

Note: In the for ( ) loop, multiple initializations, conditions, or expressions (increment / decrement) can be separated with a comma. In case of multiple conditions, the loop terminates when any one condition evaluates to false.

30. What will be the output of the following program?

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n,s=0;
    for(n=1;n<=10;n++)
            s+=n;
    cout<<s;
}
```

31. **Assignment – Sum of (1) + (1+2) + (1+2+3) +…+ (1+2+3+4+5+6+7+8+9+10)**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int x,n=0, s=0;
    clrscr ( );
    for (x=1; x<=10; x++)
    {
            s+=x;
            n+=s;
    }
    cout<<n;
    getch();
}
```

32. **Assignment – Sum of (1) + (1*2) + (1*2*3) +…+ (1*2*3*4*5*6*7*8*9*10)**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    long int x, n=1, s=0;
    clrscr ( );
    for (x=1; x<=10; x++)
        {
           n*=x;
           s+=n;
    }
    cout<<s;
    getch();
}
```

goto statement: C++ supports the goto statement to branch unconditionally from one point to another in a program. It requires a label to identify the place where the branch is to be made. A label is any valid variable name and must be followed with a colon (:). The label is placed immediately before the statement where the control is to be transferred.

Forward Jump

goto *label;*
.
.
.
label:
statement;

Backward Jump

label;
statement;
.
.
.
goto *label:*

33. Keep accepting a number till the user enters an even number

```
#include<iostream.h>
#include<conio.h>
main()
{
    int i, x;
    num:
    cout<<"a number\t";
    cin>>i;
    x = i%2;
    if (x==0)
        cout<<"\nEven number";
    else
        goto num;
}
```

33. program - to print natural numbers upto 50

```
#include <iostream.h>
void main()
{
 int a =0;
 p:
   cout<<"\n"<<++a;
 if (a<=50) goto p;
//getch();
}
```

break statement: You can exit from a loop using the break statement. When the break statement is encountered in a loop, the loop terminates immediately and the program continues with the statement after the loop. In case of nested loops, break will cause only that loop to terminate, in which the break statement is placed.

continue statement: Unlike the break statement, which causes the loop to terminate, the continue statement causes the loop to continue with the next iteration after skipping any statements in between.

34. Check whether the entered number is a prime number

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int a,n,d=2,p=0;
    clrscr ( );
    cout<<“Enter a number\t”;
    cin>>n;
    while (d<n)
    {
       a = n;
       if (a==0)
    {
            cout<<“\n Not a prime no”;
            p = 1;
            break;
     }
    else
        d++;
 }
if (p==0)
        cout<<“\n Prime number”;
getch();
}
```

35. Assignment – Displaying alphabets in decimal and character format using for, if and continue

```
{
    char c;
    for (c=65; c<=122; c++)
    {
            if (c>90  c<97)
                    continue;
            cout<<c<<"\n";
    }
    cout<<“\n”);
}
```

Testing a Character Type

These tests are done with the help of functions which are contained in the file **ctype.h**

Function	**Test**
isalnum ( )	Alpha-numeric character
isalpha ( )	Alphabetic character
isdigit ( )	Numeric character
islower ( )	Lower case character
isprint ( )	Printable character
ispunct ( )	Punctuation mark (Also includes special characters)
isspace ( )	White space character
isupper ( )	Upper case character

36. Example to test a character type

```
#include<iostream.h>
#include<ctype.h>
 void main()
{
    char c;
    c=getchar();
    if(isalpha(c))
        cout<<"\nAlphabet";
    else if (isdigit(c))
        cout<<"\nDigit";
    else if (ispunct(c))
        cout<<"\nPunctuation Mark";
    else
        cout<<"\nError";
}
```

37. Pyramid 1

```
#include<iostream.h>
#include<conio.h>
main ( )
{
    int i,j;
    clrscr ( );
    for(i=1;i<=5;i++) // for(i=5;i>=1;i--)
    {
        for (j=1;j<=i;j++)
            cout<<" * "<<"\t"; // cout<< j;  OR cout<< i;
        cout<<"\n";
    }
    getch ( );
}
```

```
*          1          1
* *        1 2        2 2
* * *      1 2 3      3 3 3
* * * *    1 2 3 4    4 4 4 4
* * * * *  1 2 3 4 5  5 5 5 5 5
```

38. Pyramid 2

```
#include<iostream.h>
main ( )
{
    int i, j, k;
    for (i=1;i<=5;i++)
    {
        for(k=5;k>=i;k--)
            cout<<" ";
        for(j=1;j<=i;j++)
            cout<<" * ";
        cout<<"\n";
    }
}
```

*Note: One space after the * will give centre-aligned output, two spaces, right-aligned*

39. Pyramid 3

```
#include<iostream.h>
#include<conio.h>
main ( )
{
    int i, j, n;
    clrscr ( );
    cout<<"Enter no. of stars:";
    cin>>n;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=i;j++)
            cout<<" * " ;
        cout<<"\n";
    }
    for (i=n-1;i>=1;i--)
    {
        for(j=1;j<=i;j++)
            cout<<" * " ;
        cout<<"\n";
    }
    getch ( );
}
```

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

Assignment

1)write a program to display the following output?

1
0 1
1 0 1
0 1 0 1

2) write a program to display the following output?

ABCDEF
ABCD
AB
A

Arrays

An array is a group of related data items that share a common name. The individual values are known as **elements** and can be accessed by writing a number called **index number** or **subscript** in brackets after the array name. The index number of the first element is 0 and the last element is total_no._of_elements-1.

Declaration of Arrays

The general form of array declaration is

data_type array_name [*size*];

When the compiler sees a character array, it terminates it with an additional null character. When declaring character arrays, always allow one extra element for the null terminator.

Initializing Arrays

Examples:

1. int num[4] = {10,20,30,40};
2. int num[4]={10,20}; *Remaining 2 elements will be automatically initialized to 0.*
3. int num[ ] = {10,20,30,40};
4. int num[4];
   num[0]=10;
   num[1]=20;
   num[2]=30;
   num[3]=40;
5. char word[6] = “Hello”;
6. char word[ ] = “Hello”;
7. char word[6] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’};
8. char word[6];
   word[0] = ‘H’;
   word[1] = ‘e’;
   word[2] = ‘l’;
   word[3] = ‘l’;
   word[4] = ‘o’;

40. Storing numbers and characters in array and displaying them

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int n[10] = {0,1,2,3,4,5,6,7,8,9};
    char s[10] = “abcdefghi”;
    clrscr ( );
    int ctr;
    for (ctr=0; ctr<10; ctr++)
    {
            cout<<“\n n[] = ”<< ctr<< n[ctr];
            cout<<“\t\t s[] = ”<< ctr<< s[ctr];
    }
    getch();
}
```

41. Accept 5 numbers from the user, store them in an array and display them in ascending order

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int num[5], n, i=0, j=0;
    clrscr ( );
    cout<<“Enter 5 numbers\n”;
    while (i<5)
            cin>>num[i++];
    for (i=1; i<5; i++)
    {
            for (j=1; j<5; j++)
            {
                    if (num[j-1] > num[j])
                    {
                            n = nm[j-1];
                            num[j-1] = num[j];
                            num[j] = n;
                    }
            }
    }
    cout<<“\n\nSorted list \n”;
    for (i=0; i<5; i++)
            cout<<“\n”<< num[i];
    getch();
}
```

42. Accept 5 numbers, store them in an array and display them as individual array elements and also display their sum

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int num[5], s=0, ctr;
    clrscr ( );
    for (ctr=0; ctr<5; ctr++)
    {
            cout<<“Enter a number\t”;
            cin>>num[ctr];
            s+=num[ctr];
    }
    for (ctr=0; ctr<5; ctr++)
            cout<<“\n num[] = ”<<ctr<< num[ctr];
    cout<<“Sum of the 5 numbers = ”<<s;
    getch();
}
```

Assignment

42. write a program to add two matrix and print their output?

```
    void main()
{
    int  a[5],b[5],c[5],i,j;
    clrscr();
  cout<<"enter the numbers";
  for(i=0;i<=5;i++)
  {
    cin>>a[i]>>b[i];
  }
  cout<<"the addition is";
  for(i=0;i<=5;i++)
{
   c[i]=a[i]+b[i];
   cout<<c[i];
}
}
```

Two-Dimensional Arrays

The general form of declaration of a two-dimensional array is

datatype arrayname [*rowSize*][*columnSize*];

As with single dimensional arrays, each dimension of the array is indexed from 0 to its maximum size – 1. The first index selects the row and the second index selects the column within the row.

Initializing Two-Dimensional Arrays

E.g.

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

[0][0] 0	[0][1] 0	[0][2] 0
[1][0] 1	[1][1] 1	[1][2] 1

43. read a multidimension array

```
    #include<stdio.h>
    #include<conio.h>
    void main()
     {
        int a[3][3],i,j;
        cout<<"enter the elements";
        for(i=0;i<3;i++)
        {
         for(j=0;j<3;j++)
         {
         cin>>a[i][j];
          }
          }
        cout<<"the elements are\n";
         for(i=0;i<3;i++)
         {
          for(j=0;j<3;j++)
          {
            cout<<a[i][j];
            }
           cout<<"\n";
            }
          }
```

Assignment

44. write a program to print add of two matrix using multidimension array?

```
void main()
    {
      int a[3][3],b[3][3],c[3][3],i,j;
      cout<<"enter the elements of a";
      for(i=0;i<3;i++)
      {
       for(j=0;j<3;j++)
       {
        cin>>a[i][j];
       }
      }
     cout<<"the elements are\n";
      for(i=0;i<3;i++)
      {
       for(j=0;j<3;j++)
       {
         cout<<a[i][j];
        }
       cout<<"\n";
        }
      cout<<"enter the elements of b";
      for(i=0;i<3;i++)
      {
       for(j=0;j<3;j++)
       {
        cin>>b[i][j];
       }
      }
     cout<<"the elements are\n";
      for(i=0;i<3;i++)
      {
       for(j=0;j<3;j++)
       {
         cout<<b[i][j];
        }
        cout<<"\n";
         }
       }
      cout<<" the addition of elements is\n";
      for(i=0;i<3;i++)
      {
       for(j=0;j<3;j++)
       {
        c[i][j]=a[i][j]+b[i][j];
        cout<<c[i][j];
       }
       cout<<"\n";
      }
      }
```

String & Math Functions

String Functions: The string handling functions are included in string.h

strcat ( )	Appends a string
strchr ( )	Finds first occurrence of a given character
strcmp ( )	Compares two strings character-wise and returns the difference in the first non-matching character
strcmpi( )	Compares two strings, non-case-sensitive
strcpy ( )	Copies one string into another Syntax: strcpy (*destination*, *source*)
strlen ( )	Finds length of a string
strlwr ( )	Converts a string to lower case
strncat ( )	Appends n characters of a string
strncmp( )	Compares n characters of two strings
strncpy( )	Copies n characters of one string into another
strnset ( )	Sets n characters of a string to a given character
strrchr ( )	Finds last occurrence of a given character in a string
strrev ( )	Reverses a string
strset ( )	Sets all characters of a string to a given character
strspn ( )	Finds first substring from given character set in a string
strupr ( )	Converts string to upper case

45.Example 1

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
main ( )
{
    char name[15]= “Archana”;
    char surname[10] = “Talekar”;
    char fullname[30]= “”;
    int len,cmp;
    clrscr ( );
    strcat (fullname, name);
    strcat (fullname, “ ”);
    strcat (fullname, surname);
    cout<<“\n”<< fullname;
    len=strlen (fullname);
    cout<<len;
    cmp=strcmp (name, surname);
    cout<<“\n”<< cmp;
    cout<<“\n”<< strlwr (fullname);
    cout<<“\n”<< strupr (fullname);
    cout<<“\n”<< strrev (fullname);
    getch ( );
}
```

Note: C permits nesting of strcat ( ).
E.g. strcat (strcat (strcat(fullname, name), “ ”), surname

46.Example 2

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{
    int cmp,len1,len2;
    char s1[25], s2[25];
    clrscr ( );
    cout<<“\nEnter two strings\n”;
    cin>>s1>>s2);
    len1 = strlen(s1);
    len2 = strlen(s2);
    cmp = strcmp(s1,s2);
    cout<<“\nLength of s1 = ”<<len1;
    cout<<“\nLength of s2 = ”<<len2;
    cout<<“\nResult of the comparison = ”<<cmp;
}
```

Note:
String functions are included in string.h
strlen ( ) returns the no. of characters in the string
strcmp ( ) compares the strings character-wise and returns the difference in ASCII values of first non-matching character.

A-Z –> 65-90
a-z –> 97-122

47.Accept a word from the user and check whether it is a palindrome

```
#include<iostream.h>
#include<string.h>
void main()
{
        char word[25];
        int a,b=0,len;
        cout<<“Enter a word\n”;
        gets(word);
        len = strlen(word) – 1;
        for (a=0; a<(len/2); a++)
        {
                if (word[a] != ‘\0’)
                        if (word[a] != word[len-a])
                        {
                                cout<<“\nNot a palindrome”;
                                b++;
                                break;
                        }
        }
        if (b==0)
                cout<<“\nIt is a palindrome”;
}
```

```
#include<string.h>
 void main()
{
        char s1[15],s2[15];
        clrscr();
        cout<<"Enter a word:");
        cin>>s1;
        strcpy(s2,s1);
        strrev(s2);
        if(strcmp(s1,s2)==0)
                cout<<"\nPalindrome";
        else
                cout<<"\nNot a palindrome";
        getch();
}
```

48.Sorting of strings

Note: In case of two-dimensional character arrays, the column subscript is not required while accepting or displaying values.

```
#include<iostream.h>
#include<string.h>
main ( )
{
    char names[5][10],temp[10];
    int i=0,j;
    while (i<5)
            cin>>names[i++];
    for(i=1;i<5;i++)
    {
            for(j=1;j<5;j++)
            {
                    if(strcmp(names[j-1],names[j])>0)
                    {
                            strcpy(temp,names[j-1]);
                            strcpy(names[j-1],names[j]);
                            strcpy(names[j],temp);
                    }
            }
    }
    for(i=0;i<5;i++)
            cout<<names[i];
}
```

49.Math Library Functions

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
void main()
{
    int n,x,y,p,a;
    float s;
    clrscr ( );
    cout<<"Enter a number\t";
    cin>>n;
    n = abs(n); ──────────────►  Absolute value, i.e., positive value
    cout<<"\nAbsolute value = "<<n;
    cout<<"\nEnter two numbers\t";
    cin>>x>>y;
    cout<<"\nEnter a positive number\t");
    cin>>a;
    s = sqrt(a); ──────────────►  Square root
    cout<<"\nSquare root of  = "<<a<<s;
    getch ( );
}
```

User-defined Functions

C++ functions can be classified into two categories – **library functions** and **user-defined functions**. main ( ) is an example of user-defined function as the user decides the task of the main ( ) function in each program. cout(), clrscr ( ), etc are examples of library functions.

The main distinction between the two categories is that library functions need not be written by the user and can be directly called, where as user-defined functions have to be created by the user before calling them.

UDFs enhance readability and reduce the size of the program. UDF can be called by simply using the function name in a statement. A function can have any number of arguments but can return only one value.

The program always starts executing with the main ( ) function irrespective of whether it is defined at the beginning or end of the program.

Types of Functions:

- **Functions with no arguments and no return value**: When a function has no arguments, it does not receive nay data from the calling function. Similarly when a function does not return a value, the calling function does not receive any data from the called function. There is no data transfer between the calling function and the called function. E.g. clrscr ( );

50.User defined function with no arguments and no return values

```
#include<iostream.h>
#include<conio.h>
void main()
{
    void printline(), value();
    clrscr ( );
    printline();
    value();
    printline();
    getch();
}
void printline()
{
    int i;
    cout<<"\n";
    for (i=1; i<=35; i++)
        cout<<"-";
    cout<<"\n";
}
void value()
{
    int year=1, period;
    float inrate, sum, principal;
    cout<<"\nEnter principal amount, rate and period :";
    cin>> principal>> inrate>> period;
    sum=principal;
    while (year <= period)
    {
        sum *= (1+ (inrate /100));
        year ++;
    }
    cout<<"\nPrincipal amount : "<<principal;
    cout<<"\nRate of Interest : "<< inrate;
    cout<<"\nPeriod : "<<period;
    cout<<"\nSum = "<<sum;
}
```

Function Declaration

Function Call

Function Definition

Note: When a function is defined after main ( ), it has to be declared in the main ( ) function with the data type of return value. If the function does not return any value it is declared and defined as void.

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

51.User defined function with argument but no return value

```
#include<iostream.h>
#include<conio.h>
void printline(char ch)
{
    int i;
    cout<<"\n";
    for (i=1; i<=52; i++)
        cout<< ch;
    cout<<"\n";
}
void value(float p, float r, int n)
{
    int year =1;
    flaot sum = p;
    while (year <=n)
    {
        sum *= (1 + (r/100));
        year++;
    }
    cout<<"\nPrincipal amount :"<<p;
    cout<<"\nRate of Interest : "<< r;
    cout<<"\nPeriod : "<<n;
    cout<<"\nSum = "<<sum;
}
void main()
{
    float principal, inrate;
    int period;
    clrscr ( );
    cout<<"Enter principal amount, interest rate, and period\n";
    cin>> principal>> inrate>> period;
    printline('-');
    value(principal, inrate, period);
    printline('=');
    getch();
}
```

Formal Arguments

Actual Arguements

Note: When the UDFs are defined before the main ( ) function, they need not be declared in the main ( ) function.

The actual and formal arguments should match in number, data type and order. The values of AA are assigned to FA on a one-to-one basis.

If the AA are more than FA, the extra AA are ignored. However if the FA are more than AA, the unmatched FA are initialized to junk values. Any mismatch in data type also results in passing of junk values.

The FA should be valid variable names, where as AA could be variable names or constants. The variables used in AA must be assigned values before calling the function. When the function is called only a copy of the values of variables of AA is passed to the called function. Any manipulation within the function has no effect on the variables used in AA.

- **Functions with arguments and return value**

52.User defined function with arguments and return values

```
#include<iostream.h>
#include<conio.h>
void main()
{
    void printline(char, int);          Data type of arguments
    float value(float, float, int), principal, inrate, amount;
    int period;        Data type of return value
    clrscr ( );
    cout<<"Enter the principal amount, interest rate and period\t";
    cin>> principal>> inrate>> period;
    printline('-',52);
    amount = value(principal, inrate, period);      Variable to store the return value
    cout<<"\nPrincipal amount : "<<principal;
    cout<<"\nRate of Interest : "<< inrate;
    cout<<"\nPeriod : "<<period;
    cout<<"\nSum ="<<amount;
    printline('=',52);
    getch();
}
void printline(char ch ,int len)
{
    int i;
        cout<<"\n";
    for (i=1; i<=len; i++)
            cout<< ch;
    cout<<"\n";
}
float value(float p, float r, int n)
{
    int year =1;
    float sum = p;
    while (year <= n)
    {
            sum *= (1 + (r/100));
            year ++;
    }
    return (sum);
}
```

- **Functions with no arguments but returns a value**: Example given later

Nested Functions: C permits nesting of functions where main ( ) calls function1, which calls function2, which calls function3 and so on. There is no limit as to how deeply functions can be nested.

53.Accept three values a, b and c and display the ratio a/(b-c) using nested functions

```
#include<iostream.h>
#include<conio.h>
int difference (int p, int q)
{
    if(p!=q)
        return (1);
    else
        return (0);
}
float ratio (int x, int y, int z)
{
    if (difference (y,z))
        return (x/(y-z));
    else
        return (0);
}
void main ( )
{
    int a,b,c;
    clrscr ( );
    cout<<"Enter values for a, b and c:";
    cin>>a>>b>>c;
    cout<<"\nRatio = "<<ratio(a,b,c);
    getch ( );
}
```

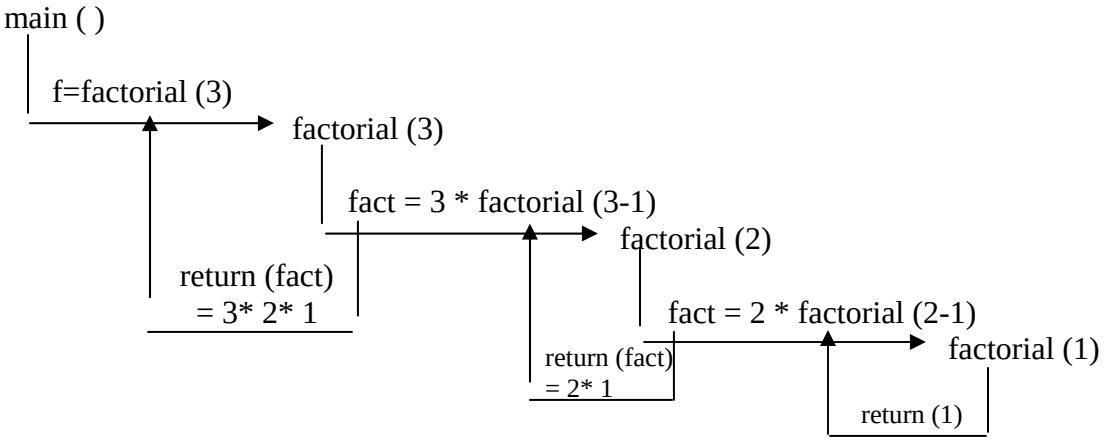
Note: main ( ) calls ratio ( ), which calls difference ( ). difference ( ) will return the value to ratio ( ) and ratio will return the value to main ( ). if (difference (y,z)) means if difference ( ) returns a non-zero value.

Recursion: When a called function in turn calls another function, a process of nesting occurs. Recursion is a special case of this process where a called function calls itself. When we write a recursive function, there must be at least one if ( ) statement to force the function to return without the recursive call being executed.

54.Accept a number and display its factorial using recursive function

```
#include<iostream.h>
#include<conio.h>
long int factorial (int n)
{
    long int fact;
    if(n==1)
        return(1);
    else if(n<1)
        return(0);
    else
        fact = n * factorial (n-1);
    return(fact);
}
void main()
{
    int num;
    long int f;
    clrscr ( );
    cout<<"Enter a number\t";
    cin>>num;
    f =fact(num);
    cout<<"\nFactorial of  = "<< num<< f;
    getch();
}
```

Note:



55.Assignment-Accept 5 nos from the user, store them in an array and display the largest number using UDF

```
#include<iostream.h>
#include<conio.h>
int largest(int a[])        When an array is formal argument, the subscript is not specified.
{
    int i;
    int max = a[0];
    for (i=1; i<5; i++);
            if (max < a[i])
                    max = a[i];
    return(max);
}
void main()
{
    int num[5], ctr;
    clrscr ( );
    cout<<"Enter 5 numbers\n";
    for (ctr=0; ctr<5; ctr++)
            cin>>num[ctr];
        cout<<"\nMaximum value = "<< largest(num);
        getch();
}
```

While passing an array as an actual argument, only the array name is used without any brackets.

Storage Class for Variables

i. **Automatic Variables**: They are declared in a function or block in which they are to be utilized. They are created when the function or block starts executing and automatically destroyed when the function or block ends. Automatic variables are also referred to as **local** or **internal** variables. A variable declared without any storage class specification is by default an automatic variable. Automatic variables may also be explicitly declared using the keyword *auto*.

56.Automatic Variables in Blocks

```
#include<iostream.h>
#include<conio.h>
void main()
{
    auto int n =5;
    clrscr ( );
    {
        auto int n =2;
        {
            auto int n = 1;
            cout<<"\n"<<n;
        }
        cout<<"\n"<<n;
    }
    cout<<"\n"<<n;
    getch();
}
```

Note:The storage class declaration auto is optional.

57.Automatic Variables (Using user defined functions)

```
#include<iostream.h>
#include<conio.h>
void function1()
{
    int m =10;
    cout<<"\n'<<m;
}
void function2()
{
    int m=100;
    function1();
    cout<<"\n"<<m;
}
void main()
{
    int m=50;
    clrscr ( );
    function2();
    cout<<"\n"<<m;
    getch();
}
```

ii. Global Variable: Variables that are available throughout the program are known as global variables. They are also known as **External** variables. Unlike local variables, global variables can be accessed by any function in the program. In case if a local and global variable have the same name, the local variable will have precedence over the global variable in the function where it is declared.

Once a global variable has been declared, any function can access it and change its value. The subsequent functions can access only the changed value. A global variable is available from the point of declaration to the end of the program.

To access a global variable declared later in the program, it can be declared with the storage class extern in the current function.

58.Global Variable (Also an example of UDF without arguments and with return value)

```
#include<iostream.h>
#include<conio.h>
int f1( )
{
    extern int x;
    x+=10;→ Changing value of global v.
    retrun(x);
}
int x;→ Declaring global variable
int f2( )
{
    int x=1;→          Local variable
    return(x);
}

→ Referring to global v. declared later in the program


int f3( )
{
    x+=10;→          Changing value of global v.
    return(x);
}
main( )
{
    clrscr ( );
    x=10;→ Initializing global variable
    cout<<"\n x = "<<x;
    cout<<"\n x = "<<f1();
    cout<<"\n x = "<<f2();
    cout<<"\n x = "<<f3();
    getch();
}
```

iii. Static Variables: Unlike automatic variables, which retain their value till the end of the block or function, static variables retain their value till the end of the program.

59.Static Variables

```
void stat()
{
    static int x =0;
    x++;
    cout<<"\n x = "<<x;
}
void main()
{
    int i;
    for (i=1;i<=5;i++)
            stat();
}
```

Note: A static variable is initialized only once and the subsequent function calls can access the value of the previous execution.
If the declaration static is removed, x will be initialized 5 times.

new concepts in c++-oops

Pointers and Arrays

One of the most common problems in programming in C (and sometimes C++) is the understanding of pointers and arrays. In C (C++) both are highly related with some small but essential differences. You declare a pointer by putting an asterisk between the data type and the name of the variable or function:

```
char *strp;      /* strp is `pointer to char' */
```

You access the content of a pointer by dereferencing it using again the asterisk:

```
*strp = 'a';          /* A single character */
```

As in other languages, you must provide some space for the value to which the pointer points. A pointer to characters can be used to point to a sequence of characters: the *string*. Strings in C are terminated by a special character NUL (0 or as char '\0'). Thus, you can have strings of any length. Strings are enclosed in double quotes:

```
strp = "hello";
```

In this case, the compiler automatically adds the terminating NUL character. Now, *strp* points to a sequence of 6 characters. The first character is `h', the second `e' and so forth. We can access these characters by an index in *strp*:

```
strp[0]    /* h */
strp[1]    /* e */
strp[2]    /* l */
strp[3]    /* l */
strp[4]    /* o */
strp[5]    /* \0 */
```

The first character also equals ``*_strp_" which can be written as ``*(_strp_ + 0)". This leads to something called *pointer arithmetic* and which is one of the powerful features of C. Thus, we have the following equations:

```
*strp == *(strp + 0) == strp[0]
*(strp + 1) == strp[1]
*(strp + 2) == strp[2]
...
```

Note that these equations are true for any data type. The addition is not oriented to bytes, it is oriented to the size of the corresponding pointer type!

The *strp* pointer can be set to other locations. Its destination may *vary*. In contrast to that, *arrays* are *fix* pointers. They point to a predefined area of memory which is specified in brackets:

```
char str[6];
```

You can view *str* to be a constant pointer pointing to an area of 6 characters. We are not allowed to use it like this:

```
str = "hallo";   /* ERROR */
```

because this would mean, to change the pointer to point to 'h'. We must copy the string into the provided memory area. We therefore use a function called strcpy() which is part of the standard C library.

```
strcpy(str, "hallo"); /* Ok */
```

Note however, that we can use *str* in any case where a pointer to a character is expected, because it is a (fixed) pointer.

Object-Oriented programming

Object-oriented programming can best be describe as a programming paradigm; a methodology of programming adopted by a programmer. Another example of a programming paradigm would be procedural (or sometimes, imperative) programming.

Procedural programming involves viewing a computer program as a sequential list of computational steps (or procedures) to be carried out. In procedural programming, the list of procedures can be further broken down into subroutines and functions (not to be confused with mathematical functions which are used within functional programming).

This is very much an “old skool” approach to programming and is adopted by many high-level languages such as BASIC. It’s also worth pointing out that most Web-based languages, such as PHP, ASP and Javascript, can be written using this approach - in fact when I first started out as a web developer, I was using ASP script, PHP and Javascript in this very manner, blissfully unaware of the OOP potential there-in.

In OOP, a program is seen as comprising a collection of individual modules, or objects, that act on each other. Each of these objects could be seen as an independent program in itself, with a distinct role or responsibility.

OOP provides greater flexibility and easier maintainance across large systems and can sometimes make understanding and analysing complex procedures a lot easier.

It's worth noting, at this point, that most OOP languages (such as C++ or Java) are stateful where as often procedural programming languages are stateless.

Enemy of the State

In computer science, a "state" is a particular set of instructions which will be executed in response to the machine's input. An information system or protocol that relies upon state is said to be stateful. One that does not is said to be stateless.

OOP PHP is stateless because each script must be called on a page refresh where as Javascript is stateful because it can make use of event listeners - specific input - and the wonders of XMLHttpRequest - thus negating the need for refreshes.

Objects and Classes

The term "Object," that gives OOP it's name, refers to a conceptual object that represents an item in our program or system. This could be anything from an online form or a computer file, to a real world object such as a car.

This representation consists of attributes - the characteristics of our object; and methods - a set of functions and calculations that are either performed to modify the object itself, or are involved in some external effect.

The term "Class" represents the definition (or classification - class) of our object. For example, if we were to write a class called "car", we could create any number of instances of that class - say "Porsche", "Ferrari" and "Jaguar". Each of these instances is an Object. This illustrates that a class is effectively a set of objects that all share common attributes.

Keeping it "in the family"

One of the greatest advantages to using objects is encapsulation. This basically means that data within an object is only available/modifiable via the object's methods - this is generally known as the interface of the object.

This resultant limitation of scope allows an object-oriented programmer the freedom to declare attributes, variables and methods without having to worry about clashes with those in other objects.

Encapsulation also means that, as long as we don't alter the interface, we can change how an object works (to increase performance, add functionality etc) without affecting the rest of our system.

Now let's take a look at how objects and classes can relate to each other.

Note: From this point on, the concepts are going to get a little more complicated. Please bear in mind that I'm writing an introduction here - I fully intend to go over these concepts again (with examples) whilst applying them to Javascript and PHP in the tutorials to follow.

Characteristics of a Class

When writing a class, there are a number of characteristics that are worth taking into account:

Constructor
The constructor of a class is a special operation that is run upon instantiation - when an object is created. They are often distinguished by having the same name as the class itself. It's main purpose is to set-up the attributes of a class and to establish the class invariant - to basically make sure that the attributes of the class conform to the class interface. It cannot have a return type.

A properly written constructor should never contain any functionality that could fail, thus leaving the object in an invalid state.

Destructor
The destructor of a class is the opposite of the constructor in that it is run when an object is destroyed. It's function is to clean up and to free the resources which were used by the object during run-time and unlink it from other objects or resources. This should result in the invalidation of any references in the process.

Relationships
There are a number of relationships that can be used when interaction is needed between objects. These are as follows:

Inheritance
Inheritance allows a (sub)class to inherit all the attributes and methods of a parent class - in effect, extending the parent. It can best be described as an "is a" relationship. For instance, if we had a class of "fruit", we could extend it by defining classes of "apple", "orange" and "banana". Each of these subclasses could be described in the following way:

apple "is a" fruit
orange "is a" fruit
banana "is a" fruit
Because each of our subclasses extends the "fruit" class, it has all the attributes and methods of a "fruit" plus any specific characteristics of it's own.

Here's a more web-development-oriented example using form elements and presented in pseudocode:

```
class formElement
{
  Attributes:
    id
    name
    class
}

class input extends formElement
{
  Attributes:
    value
    type
}

class textarea extends formElement
{
  Attributes:
    cols
    rows
}
```

In this example the two classes "input" and "textarea" have inherited from "formElement". This means they inherit formElement's attributes like so:

input is a formElement

```
input
{
```

```
  Attributes:
    id
    name
    class
    value
    type
}
```

```
textarea is a formElement

textarea
{
  Attributes:
    id
    name
    class
    cols
    rows
}
```

However, as the parent (super) class, formElement stays exactly the same:

```
formElement
{
  Attributes:
    id
    name
    class
}
```

As you can imagine, this relationship can be incredibally useful. Some languages even allow multiple inheritance where a class can have more than one parent (super) class. Sadly this isn't the case in either Javascript or PHP, however it is possible to obtain the same effect using other types of relationship.

Composition - Association and Aggregation

Composition is a slightly different sort of relationship - this is where it could be said that a class was "composed" of other classes. For instance, a wall is "composed" of bricks and a molecule is "composed" of atoms. Neither of these examples could be described as inheritance - the statement, "a wall is a brick" simply isn't true. Composition can be described as "has a" and "uses a" relationships; a wall "has a" brick or a wall "uses a" brick.

Let's take a look at a "has a" relationship in pseudocode. Let's first define some simple classes:

```
class brick
{
}

class wall
{
  Attributes:
    brick1
    brick2
    brick3
    brick4
```

```
  Methods:
    // Constructor
    wall()
    {
      this.brick1 = new brick();
      this.brick2 = new brick();
      this.brick3 = new brick();
      this.brick4 = new brick();
    }
}
```

You can see that “wall” contains a number of brick attributes. We want each of these attributes to be a “brick” object. To do this we simply instantiate them within the constructor of the “wall” class. Each of these brick classes will function as a normal class but also as an attribute of “wall.” This is known as association.

Now let’s take a look at a “uses a” relationship:

```
class person
{
}

class car
{
  Attributes:
    driver

  Methods:
    // Constructor
    car(driver)
    {
      this.driver = driver;
    }
}
me = new person();

myMotor = new car(me)
```

;In this example, we can see that the “person” class has been instantiated into the object “me”. This object is then passed into an instantiation of the “car” class. This means that the object “myMotor” (our instantiation of the “car” class) uses the object “me”. This relationship is known as aggregation.

The main difference between these two relationships is simple; in association the composing classes are destroyed when the parent is destroyed, however, in aggregation they are not. This ultimately means that, when aggregation is adopted as a relationship, the composing classes and objects can be re-used in other classes and objects within the system.

Polymorphism

An object-oriented programming language must support polymorphism; meaning different classes can have different behaviours for the same attribute or method. This can best be illustrated with an example:

```
class formElement
{
  Attributes:
    id
    name
    class

  Methods:
    getHtml()
    {
      // returns generic form element HTML
    }
}
```

```
class textarea extends formElement
{
  Attributes:
    cols
    rows

  Methods:
    getHtml()
    {
      // returns textarea form element HTML
    }
}
```

As you can see in the example, both classes have the method “getHtml” but the “textarea” class is a subclass of the “formElement” class. This results in the “getHtml” method being overloaded. Overloading is a good example of polymorphism in action - an object-oriented language must support polymorphism in order to know which “getHtml” method applies to which object.

Polymorphism, at it’s most basic, describes the fact that a given function may have different specifications, depending on the object to which it is applied.

Why use Object-Oriented programming?

Object-orientation can help keep projects simple by breaking them down in to managable chunks. Those chunks can then be re-used in other projects, thus saving time in the long-run. In fact, adopting an object-oriented approach can be the foundation of a truly successful team environment; through promoting modularity, an object-oriented environment breeds improved code reusability and maintainability.

Classes and Objects

C++ allows the declaration and definition of classes. Instances of classes are called *objects*. There we have developed a class *Point*. In C++ this would look like this:

```
class Point {
  int _x, _y;       // point coordinates

public:             // begin interface section
  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};

  Point apoint;
```

This declares a class *Point* and defines an object *apoint*. You can think of a class definition as a structure definition with functions (or ``methods''). Additionally, you can specify the *access rights* in more detail. For example, *_x* and *_y* are **private**, because elements of classes are private as default. Consequently, we explicitly must ``switch'' the access rights to declare the following to be **public**. We do that by using the keyword public followed by a colon: Every element following this keyword are now accessible from outside of the class.

We can switch back to private access rights by starting a private section with private:. This is possible as often as needed:

```
class Foo {
  // private as default ...
public:
  // what follows is public until ...
private:
  // ... here, where we switch back to private ...
public:
  // ... and back to public.
};
```

Recall that a structure struct is a combination of various data elements which are accessible from the outside. We are now able to express a structure with help of a class, where all elements are declared to be public:

```
class Struct {
public:       // Structure elements are public by default
  // elements, methods
};
```

This is exactly what C++ does with struct. Structures are handled like classes. Whereas elements of classes (defined with class) are private by default, elements of structures (defined with struct) are public. However, we can also use private: to switch to a private section in structures.

Let's come back to our class *Point*. Its interface starts with the public section where we define four methods. Two for each coordinate to set and get its value. The set methods are only declared. Their actual functionality is still to be defined. The get methods have a function body: They are defined *within* the class or, in other words, they are *inlined methods*.

This type of method definition is useful for small and simple bodies. It also improve performance, because bodies of inlined methods are ``copied" into the code wherever a call to such a method takes place.

On the contrary, calls to the set methods would result in a ``real" function call. We define these methods outside of the class declaration. This makes it necessary, to indicate to which class a method definition belongs to. For example, another class might just define a method *setX()* which is quite different from that of *Point*. We must be able to define the *scope* of the definition; we therefore use the scope operator ``::":

```
void Point::setX(const int val) {
  _x = val;
}

void Point::setY(const int val) {
  _y = val;
}
```

Here we define method *setX()* (*setY()*) within the scope of class *Point*. The object *apoint* can use these methods to set and get information about itself:

```
Point apoint;
apoint.setX(1);     // Initialization
apoint.setY(1);
//
// x is needed from here, hence, we define it here and
// initialize it to the x-coordinate of apoint
//

int x = apoint.getX();
```

The question arises about how the methods ``know" from which object they are invoked. This is done by implicitly passing a pointer to the invoking object to the method. We can access this pointer within the methods as this. The definitions of methods *setX()* and *setY()* make use of class members *_x* and *_y*, respectively. If invoked by an object, these members are ``automatically" mapped to the correct object. We could use this to illustrate what actually happens:

```
void Point::setX(const int val) {
  this->_x = val;   // Use this to reference invoking
                    // object
}

void Point::setY(const int val) {
  this->_y = val;
}
```

Here we explicitly use the pointer this to explicitly dereference the invoking object. Fortunately, the compiler automatically ``inserts" these dereferences for class members, hence, we really can use the first definitions of *setX()* and *setY()*. However, it sometimes make sense to know that there is a pointer this available which indicates the invoking object.

60. program- to print add of 2 numbers

```
#include<conio.h>
class A
{
public:
void add()
{
int a,b,c;
cout<<"enter 2 nos.:";
cin>>a>>b;
c=a+b;
cout<<"addition is"<<c;
}
};
void main()
{
A ob;
clrscr();
ob.add();
getch();
}
```

61. program- to display a data

```
# include <iostream.h>
class smallobj
{
clrscr();
private:
  int somedata;
public:
  void setdata(int d)
    { somedata = d;}
  void showdata()
    { cout << "\nData is " << somedata;}
};
void main()
  {
  smallobj s1, s2;
  s1.setdata(1066);
  s2.setdata(1776);
  s1.showdata();
  s2.showdata();
  }
getche();
}
```

62.program- todisplay multifunction using class

```
#include<iostream.h>
#include<conio.h>
class A
{
 int a,b,c;
  public:
 void add()
 {
     cout<<"enter the values of a and b";
     cin>>a>>b;
     c= a+b;
     cout<<"the add is "<<c;
 }
 void sub()
 {
     cout<<"enter the values of a and b";
     cin>>a>>b;
     c= a-b;
     cout<<"the add is "<<c;
 }
 void mul()
  {
      cout<<"enter the values of a and b";
      cin>>a>>b;
      c= a*b;
      cout<<"the add is "<<c;
  }
 void div()
  {
      cout<<"enter the values of a and b";
      cin>>a>>b;
      c= a/b;
      cout<<"the add is "<<c;
  }
};
void main();
{
   A a;
    a.add();
    a.sub();
     a.mul();
     a.div();
}
```

63. write a program to print area of three triangle (hint:create 3 objects)?

SCOPE RESOLUTION-VARIABLE

To find which function belongs to which class was not possible in c.this problem was solved in c++ by using **scope resolution operator(:).**it is used to find which variable or method belongs to which class .

64. **scope resolution using variable**

```
#include<iostream.h>
#include<conio.h>
int m=10;
void main()
{
int m=20;
clrscr();
{
int k=4;
int m=30;
cout<<"inner block"<<"\n";
cout<<" k= "<<k<<"\n";
cout<<"m= "<<m<<"\n";
cout<<"::m= "<<::m<<"\n";
}
cout<<"outer block";
cout<<"m= "<<m<<"\n";
cout<<"::m= "<<::m<<"\n";
getch();
}
```

SCOPE RESOLUTION-FUNCTION

65. scope resolution using function

```
#include<iostream.h>
#include<conio.h>
class A
{
int a,b,c;
public:
void add(void);
};
void A::add(void)
{
cout<<"enter the no.:";
cin>>a>>b;
c=a+b;
cout<<c;
}
void main()
{
A a1;
clrscr();
a1.add();
getch();
}
```

66. program- to print the item

```
#include<iostream.h>
#include<conio.h>
class item
{
float base,height;
public:
void getdata(float b,float h);
void putdata(void)
{
float a;
cout<<"base:"<<base<<"\n";
cout<<"height:"<<height<<"\n";
a=0.5*base*height;
cout<<"area is:"<<a;
}
};
void item:: getdata(float b,float h)
{
base=b;
height=h;
}
void main()
{
item x;
clrscr();
cout<<"\n object x"<<"\n";
x.getdata(100,299.95);
x.putdata();
item y;
cout<<"\n  object y"<<"\n";
y.getdata(200,799);
y.putdata();
item z;
cout<<"\n  object z"<<"\n";
z.getdata(300,499);
z.putdata();
getch();
}
```

67. Assignment-to display the data

```
#include<iostream.h>
#include<conio.h>
class item
{
int number;
float cost;
public:
void getdata(int a,float b);
void putdata(void)
{
cout<<"number:"<<number<<"\n";
cout<<"cost:"<<cost<<"\n";
}
};
void item:: getdata(int a,float b)
{
number=a;
cost=b;
}
void main()
{
item x;
clrscr();
cout<<"\n object x"<<"\n";
x.getdata(100,299.95);
x.putdata();
item y;
cout<<"\n  object y"<<"\n";
y.getdata(200,799);
y.putdata();
getch();
}
```

68.Assignment- to print largest value

```
#include<iostream.h>
#include<conio.h>
class set
{
int m,n;
public:
void input(void);
void display(void);
int largest(void);
};
int set::largest(void)
{
if(m>=n)
return(m);
else
return(n);
}
void set::input(void)
{
cout<<"input values of m & n:";
cin>>m>>n;
}
void set::display(void)
{
cout<<"largest
value="<<largest()<<"\n";
}
void  main()
{
set A;
clrscr();
A.input();
A.display();
getch();
}
```

69. to display complex values using friend

```
#include<iostream.h>
#include<conio.h>
class complex
{
float x,y;
public:
void input(float real,float img)
{
x=real;
y=img;
}
friend complex sum(complex,complex);
void  show(complex);
 };
complex sum(complex c1,complex c2)
{
complex c3;
c3.x=c1.x+c2.x;
c3.y=c1.y+c2.y;
return(c3);
}
void complex::show(complex c)
{
cout<<c.x<<"+j"<<c.y<<"\n";
}

void main()
{
clrscr();
complex A,B,C;
A.input(3.1,5.26);
B.input(2.75,1.2);
C=sum(A,B);
cout<<"A=";
A.show(A);
cout<<"B=";
B.show(B);
cout<<"C=";
C.show(C);
getch();
}
```

manipulators: they are mainly to set the width of a line or to format the output . we have two manipulators setw and endl. **endl** is end of line it act like same as ("\n") whereas **setw** is used to set the width of character with some space.

70. example

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main()
{
int basic=950,allowance=95,total=1045;
cout<<setw(10)<<"Basic"<<setw(10)<<basic<<endl<<setw(10)<<"Allowance"<<setw(10)<<all
owance<<endl<<setw(10)<<"Total"<<setw(10)<<total<<endl;
getch();
}
```

inline functions

71. example of inline function

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
inline float mul(float x,float y)
{
return(x*y);
}
inline double div(double p,double q)
{
return(p/q);
}

void main()
{
float a=12.345;
float b=9.82;
clrscr();
cout<<mul(a,b)<<"\n";
cout<<div(a,b)<<"\n";
getch();
//return 0;
}
```

STATIC

72. to represent static function

```
#include<iostream.h>
#include<conio.h>
class test
{
static int count;
int code;
public:
void setcode(void)
{
code =++count;
}
void showcode(void)
{
cout<<"object no:"<<code<<"\n";
}
static void showcount(void)
{
cout<<"count:"<<count<<"\n";
}
};
int test::count;
int main()
{
test t1,t2;
clrscr();
t1.setcode();
t2.setcode();
test::showcount();
test t3;
t3.setcode();
test::showcount();
t1.showcode();
t2.showcode();
t3.showcode();
getch();
}
```

73. to represent static variable

```
#include<iostream.h>
#include<conio.h>
class item
{
static int count;
int number;
public:
void getdata(int a)
{
number=a;
count++;
}
void getcount(void)
{
cout<<"count:";
cout<<count<<"\n";
}
};
int item::count;
void main()
{
item a,b,c;
clrscr();
a.getcount();
b.getcount();
c.getcount();
a.getdata(100);
b.getdata(200);
c.getdata(300);
cout<<"after reading:"<<"\n";
a.getcount();
b.getcount();
c.getcount();
getch();
}
```

Constructors : They have same name like class. they don't have return type. they are invoked through object. they are used to initialize private variable.

74. simple constructor

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
A()
{
cout<<"inside constructor";
}
};
void  main()
{
clrscr();
A a1;
getch();
}
```

Constructors are methods which are used to initialize an object at its definition time. We extend our class *Point* such that it initializes a point to coordinates (0, 0):

```
class Point {
  int _x, _y;

public:
  Point() {
    _x = _y = 0;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
```

Constructors have the same name of the class (thus they are identified to be constructors). They have no return value. As other methods, they can take arguments. For example, we may want to initialize a point to other coordinates than (0, 0). We therefore define a second constructor taking two integer arguments within the class:

```
class Point {
  int _x, _y;

public:
  Point() {
    _x = _y = 0;
  }
  Point(const int x, const int y) {
    _x = x;
    _y = y;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
```

Constructors are implicitly called when we define objects of their classes:

```
Point apoint;          // Point::Point()
Point bpoint(12, 34);  // Point::Point(const int, const int)
```

With constructors we are able to initialize our objects . We are now able to define a class *List* where the constructors take care of correctly initializing its objects.

If we want to create a point from another point, hence, copying the properties of one object to a newly created one, we sometimes have to take care of the copy process. For example, consider the class List which allocates dynamically memory for its elements. If we want to create a second list which is a copy of the first, we must allocate memory and copy the individual elements. In our class Point we therefore add a third constructor which takes care of correctly copying values from one object to the newly created one:

```
class Point {
int _x, _y;

public:
  Point() {
    _x = _y = 0;
  }
  Point(const int x, const int y) {
    _x = x;
    _y = y;
  }
  Point(const Point &from) {
    _x = from._x;
    _y = from._y;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
```

The third constructor takes a constant reference to an object of class *Point* as an argument and assigns *_x* and *_y* the corresponding values of the provided object.

This type of constructor is so important that it has its own name: *copy constructor*. It is highly recommended that you provide for each of your classes such a constructor, even if it is as simple as in our example. The copy constructor is called in the following cases:

```
Point apoint;          // Point::Point()
Point bpoint(apoint);  // Point::Point(const Point &)
Point cpoint = apoint; // Point::Point(const Point &)
```

With help of constructors we have fulfilled one of our requirements of implementation of abstract data types: Initialization at definition time. We still need a mechanism which automatically ``destroys" an object when it gets invalid (for example, because of leaving its scope). Therefore, classes can define *destructors*.

constructor-with parameters

75. Example of constructor with parameters

```
#include<iostream.h>
#include<conio.h>
class code
{
int id;
public:
code() { }
code(int a) { id=a;}
code (code &x)
{
id=x.id;
}
void display(void)
{
cout<<id;
}
};
void main()
{
clrscr();

code A(100);
code B(A);
code C=A;
code D;
D=A;
cout<<"\nid of A:";
A.display();
cout<<"\nid of B:";
B.display();
cout<<"\nid of C:";
C.display();
cout<<"\nid of D:";
D.display();
getch();
}
```

76. Example

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
A()
{
cout<<"inside constructor";
}
A(int a)
{
cout<<"\none parameter";
}
A(int a,int b)
{
cout<<"\ntwo parameter";
}
};
void  main()
{
clrscr();
A a1,a2(5),a3(2,4);
getch();
}
```

77. Assignment

```
#include<iostream.h>
#include<conio.h>
class integer
{
int m,n;
public:
integer(int,int);
void display(void)
{
cout<<"m= "<<m<<"\n";
cout<<"n= "<<n<<"\n";
}
};
integer:: integer(int x,int y)
{
m=x;
n=y;
}
void main()
{
clrscr();
integer int1(0,100);
integer int2=integer(25,75);
cout<<"\nobject1"<<"\n";
int1.display();
cout<<"\nobject2"<<"\n";
int2.display();
getch();
}
```

Destructors

Consider a class *List*. Elements of the list are dynamically appended and removed. The constructor helps us in creating an initial empty list. However, when we leave the scope of the definition of a list object, we must ensure that the allocated memory is released. We therefore define a special method called *destructor* which is called once for each object at its destruction time:

```
void foo() {
  List alist;     // List::List() initializes to
                  // empty list.
  ...             // add/remove elements
}                 // Destructor call!
```

Destruction of objects take place when the object leaves its scope of definition or is explicitly destroyed. The latter happens, when we dynamically allocate an object and release it when it is no longer needed.

Destructors are declared similar to constructors. Thus, they also use the name prefixed by a tilde (~) of the defining class:

78. Example of destructor

```
#include<iostream.h>
int count=0;
class alpha
{
public:
alpha()
{
count++;
cout<<"\n no. of objects created"<<count;
}
~alpha()
{
cout<<"\n no. of objects destroyed"<<count;
count--;
}
};
void main()
{
clrscr();
cout<<"\n enter main\n";
alpha a1,a2,a3,a4;
{
cout<<"\n\nenter block1\n";
alpha a5;
}
{
cout<<"\n\nenter block2\n";
alpha a6;
}
cout<<"\n\nre-enter block";
}
 class Point {
   int _x, _y;

 public:
   Point() {
     _x = _y = 0;
   }
   Point(const int x, const int y) {
     _x = xval;
     _y = yval;
   }
   Point(const Point &from) {
     _x = from._x;
     _y = from._y;
   }

   ~Point() { /* Nothing to do! */ }
    void setX(const int val);
   void setY(const int val);
   int getX() { return _x; }
   int getY() { return _y; }
 };
```

Destructors take no arguments. It is even invalid to define one, because destructors are implicitly called at destruction time: You have no chance to specify actual arguments.

79.Assignment- to display destructor and to print objects destroyed

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
A()
{
cout<<"inside constructor";
}
A(int a)
{
cout<<"\ninside a"<<a;
}
A(int a,int b)
{
cout<<"\ninside a & b"<<a<<b;
}
~A()
{
cout<<"\ndestructor";
}
};
void  main()
{
clrscr();
A a1,a2(2,4),a3(5);
getch();
}
```

Inheritance

In our pseudo language, we formulate inheritance with ``inherits from". In C++ these words are replaced by a colon. As an example let's design a class for 3D points. Of course we want to reuse our already existing class *Point*. We start designing our class as follows:

```
class Point3D : public Point {
  int _z;

public:
  Point3D() {
    setX(0);
    setY(0);
    _z = 0;
  }
  Point3D(const int x, const int y, const int z) {
    setX(x);
    setY(y);
    _z = z;
  }

  ~Point3D() { /* Nothing to do */ }

  int getZ() { return _z; }
  void setZ(const int val) { _z = val; }
};
```

Types of Inheritance

You might notice again the keyword public used in the first line of the class definition (its *signature*). This is necessary because C++ distinguishes two types of inheritance: *public* and *private*. As a default, classes are privately derived from each other. Consequently, we must explicitly tell the compiler to use public inheritance.

The type of inheritance influences the access rights to elements of the various superclasses. Using public inheritance, everything which is declared private in a superclass remains private in the subclass. Similarly, everything which is public remains public. When using private inheritance the things are quite different as is shown.

The leftmost column lists possible access rights for elements of classes. It also includes a third type protected. This type is used for elements which should be directly usable in subclasses but which should not be accessible from the outside. Thus, one could say elements of this type are between private and public elements in that they can be used within the class hierarchy rooted by the corresponding class.

Table : Access rights and inheritance.

	Type of Inheritance	
	private	public
private	private	private
protected	private	protected
public	private	public

The second and third column show the resulting access right of the elements of a superclass when the subclass is privately and publically derived, respectively.

Construction

When we create an instance of class *Point3D* its constructor is called. Since *Point3D* is derived from *Point* the constructor of class *Point* is also called. However, this constructor is called *before* the body of the constructor of class *Point3D* is executed. In general, prior to the execution of the particular constructor body, constructors of every superclass are called to initialize their part of the created object.

When we create an object with

```
Point3D point(1, 2, 3);
```

the second constructor of *Point3D* is invoked. Prior to the execution of the constructor body, the constructor *Point()* is invoked, to initialize the *point* part of object *point*. Fortunately, we have defined a constructor which takes no arguments. This constructor initializes the 2D coordinates *_x* and *_y* to 0 (zero). As *Point3D* is only derived from *Point* there are no other constructor calls and the body of *Point3D(const int, const int, const int)* is executed. Here we invoke methods *setX()* and *setY()* to explicitly override the 2D coordinates. Subsequently, the value of the third coordinate *_z* is set.

This is very unsatisfactory as we have defined a constructor *Point()* which takes two arguments to initialize its coordinates to them. Thus we must only be able to tell, that instead of using the *default constructor Point()* the paramterized *Point(const int, const int)* should be used. We can do that by specifying the desired constructors after a single colon just before the body of constructor

Point3D():

```
class Point3D : public Point {
 ...
public:
  Point3D() { ... }
  Point3D(
    const int x,
    const int y,
    const int z) : Point(x, y) {
    _z = z;
  }
 ...
};
```

If we would have more superclasses we simply provide their constructor calls as a comma separated list. We also use this mechanism to create contained objects. For example, suppose that class *Part* only defines a constructor with one argument. Then to correctly create an object of class *Compound* we must invoke *Part()* with its argument:

```
class Compound {
  Part part;
  ...
public:
  Compound(const int partParameter) : part(partParameter) {
    ...
  }
  ...
};
```

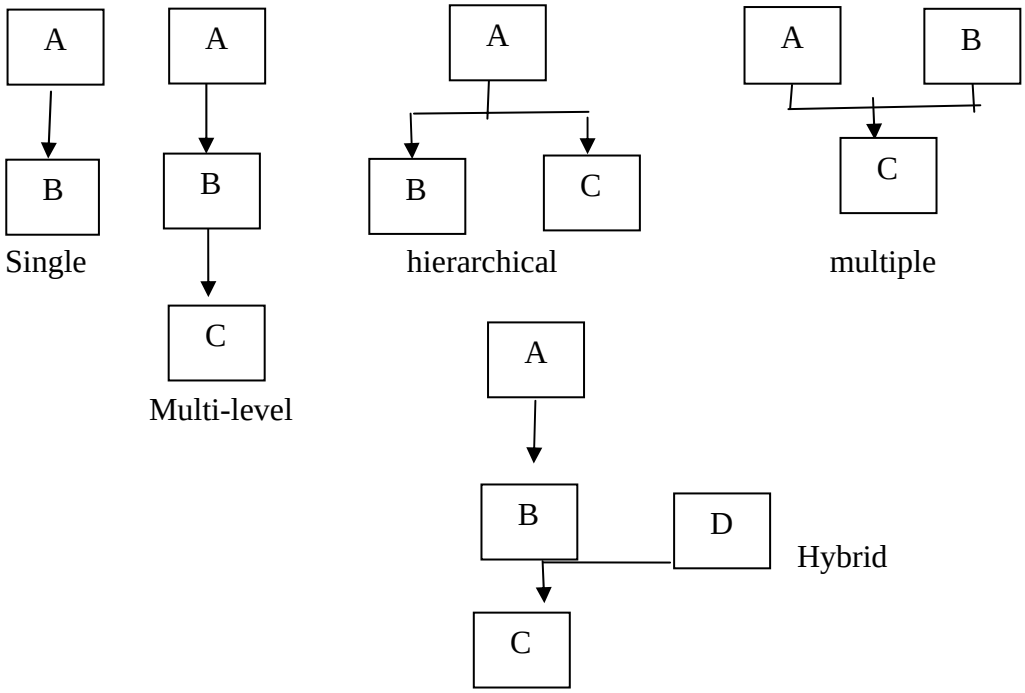
This dynamic initialization can also be used with built-in data types. For example, the constructors of class *Point* could be written as:

```
Point() : _x(0), _y(0) {}
Point(const int x, const int y) : _x(x), _y(y) {}
```

You should use this initialization method as often as possible, because it allows the compiler to create variables and objects correctly initialized instead of creating them with a default value and to use an additional assignment (or other mechanism) to set its value.

Destruction

If an object is destroyed, for example by leaving its definition scope, the destructor of the corresponding class is invoked. If this class is derived from other classes their destructors are also called, leading to a recursive call chain.



Multiple Inheritance

C++ allows a class to be derived from more than one superclass, as was already briefly mentioned in previous sections. You can easily derive from more than one class by specifying the superclasses in a comma separated list:

```
class DrawableString : public Point, public DrawableObject {
  ...
public:
  DrawableString(...) :
    Point(...),
    DrawableObject(...) {
      ...
  }
  ~DrawableString() { ... }
  ...
};
```

We will not use this type of inheritance in the remainder of this tutorial. Therefore we will not go into further detail here.

inheritance can be classified into following types:

80.single inheritance

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
void add()
{
int a,b,c;
cout<<"enter the values";
cin>>a>>b;
c=a+b;
cout<<"the addition is"<<c;
}
};
class B:public  A
{
public:
void sub()
{
int x,y,z;
cout<<"enter the values";
cin>>x>>y;
z=x-y;
cout<<"the subtraction is"<<z;
}
};
void main()
{
clrscr();
B a1;
a1.add();
a1.sub();
getch();
}
```

81.multiple inheritance

```
#include<iostream.h>
class A
{
public:
void add()
{
int a,b,c;
cout<<"enter the values";
cin>>a>>b;
c=a+b;
cout<<"the addition is"<<c;
}
};
class B
{
public:
void sub()
{
int x,y,z;
cout<<"enter the values";
cin>>x>>y;
z=x-y;
cout<<"the subtraction is"<<z;
}
};
class C:public A,public B
{
public:
void mul()
{
int s,t,q;
cout<<"enter the values";
cin>>s>>t;
q=s*t;
cout<<"the multi is"<<q;
}
};
void main()
{
C c1;
c1.mul();
c1.add();
c1.sub();
}
```

82.hirarical inheritance

```
#include<iostream.h>
class A
{
public:
void add()
{
int a,b,c;
cout<<"enter the values";
cin>>a>>b;
c=a+b;
cout<<"the addition is"<<c;
}
};
class B:public A
{
public:
void sub()
{
int x,y,z;
cout<<"enter the values";
cin>>x>>y;
z=x-y;
cout<<"the subtraction is"<<z;
}
};
class C:public A
{
public:
void mul()
{
int s,t,q;
cout<<"enter the values";
cin>>s>>t;
q=s*t;
cout<<"the multi is"<<q;
}
};
void main()
{
B b;
C c;
b.sub();
b.add();
c.mul();
c.add();
}
```

83.multilevel inheritance

```
#include<iostream.h>
class A
{
public:
void add()
{
int a,b,c;
cout<<"enter the values";
cin>>a>>b;
c=a+b;
cout<<"the addition is"<<c;
}
};
class B:public  A
{
public:
void sub()
{
int x,y,z;
cout<<"enter the values";
cin>>x>>y;
z=x-y;
cout<<"the subtraction is"<<z;
}
};
class C:public B
{
public:
void mul()
{
int s,t,q;
cout<<"enter the values";
cin>>s>>t;
q=s*t;
cout<<"the multi is"<<q;
}
};
void main()
{
C a1;
a1.add();
a1.sub();
a1.mul();
}
```

84.hybrid inheritance

```
#include<iostream.h>
class A
{
public:
void add()
{
int a,b,c;
cout<<"enter the values";
cin>>a>>b;
c=a+b;
cout<<"the addition is"<<c;
}
};
class B:public A
{
public:
void sub()
{
int x,y,z;
cout<<"enter the values";
cin>>x>>y;
z=x-y;
cout<<"the subtraction is"<<z;
}
};
class D
{
public:
void div()
{
float x,y,z;
cout<<"enter the values";
cin>>x>>y;
z=x/y;
cout<<"the division is"<<z;
}
};
class C:public B,public D
{
public:
void mul()
{
int s,t,q;
cout<<"enter the values";
cin>>s>>t;
q=s*t;
cout<<"the multi is"<<q;
}
};
void main()
{
C c1;
c1.mul();
c1.sub();
c1.div();
c1.add();
}
```

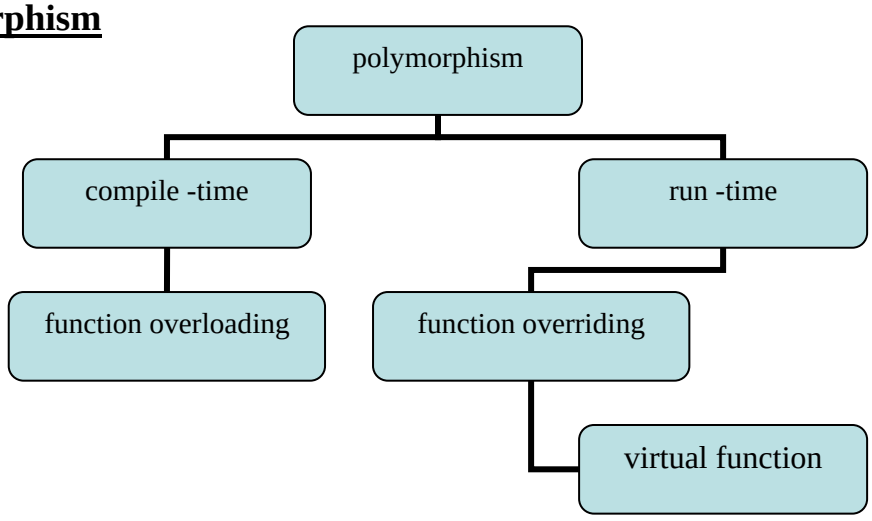
85. Example

```
#include<iostream.h>
#include<conio.h>
class student
{
protected:
int roll_no;
public:
void get_no(int a)
{
roll_no=a;
}
void put_no(void)
{
cout<<"roll no:"<<roll_no<<"\n";
}
};
class test:public student
{
protected:
float part1,part2;
public:
void get_marks(float x,float y)
{
part1=x;
part2=y;
}
void put_marks(void)
{
cout<<"marks
obtained:"<<"\n"<<"part1="<<part1<<"
part2="<<part2<<"\n";
}
};
class sports
{
protected:
float score;
public:
void get_score(float a)
{
score=a;
}
void put_score(void)
{
cout<<"sports wt:"<<score<<"\n\n";
}
};
class result:public test,public sports
{
float total;
public:
void display(void);
};
void result::display(void)
{
total=part1+part2+score;
put_no();
put_marks();
put_score();
cout<<"total score:"<<total<<"\n";
}
void main()
{
clrscr();
result s1;
s1.get_no(123);
s1.get_marks(27.5,33.0);
s1.get_score(6.0);
s1.display();
getch();
}
```

86.Example

```
#include<iostream.h>
#include<conio.h>
class STUD
{
public:
void stud()
{
int roll;
cout<<"enter the roll no.:";
cin>>roll;
cout<<"the roll no. is"<<roll;
}
};
class TEST:public STUD
{
public:
void test()
{
int x,y,z,t;
cout<<"enter the marks in 3subs.:";
cin>>x>>y>>z;
t=x+y+z;
cout<<"total marks are"<<t;
}
};
class SPORT
{
public:
void sport()
{
char x[20];
char g;
cout<<"enter the name of sport & grade";
cin>>x>>g;
cout<<"U got "<<g<<" grade in sport "<<x;
}
};
class RESULT:public TEST,public SPORT
{
public:
void result()
{
void stud();
void test();
void sport();
}
};
void main()
{
clrscr();
RESULT c1;
c1.result();
c1.test();
c1.sport();
//c1.student();
getch();
}
```

Polymorphism



polymorphism –many forms
compile time-related to compile time
run time- period related to run time
function overloading –one class with diff signatures of function
function overriding-two class with same signatures of function

In our pseudo language we are able to declare methods of classes to be virtual, to force their evaluation to be based on object content rather than object type. We can also use this in C++:

```
class DrawableObject {
public:
  virtual void print();
};
```

Class *DrawableObject* defines a method *print()* which is virtual. We can derive from this class other classes:

```
class Point : public DrawableObject {
  ...
public:
  ...
  void print() { ... }
};
```

Again, *print()* is a virtual method, because it inherits this property from *DrawableObject*. The function *display()* which is able to display any kind of drawable object, can then be defined as:

```
void display(const DrawableObject &obj) {
  // prepare anything necessary
  obj.print();
}
```

When using virtual methods some compilers complain if the corresponding class destructor is not declared virtual as well. This is necessary when using pointers to (virtual) subclasses when it is time to destroy them. As the pointer is declared as superclass normally its destructor would be called. If the destructor is virtual, the destructor of the actual referenced object is called (and then, recursively, all destructors of its superclasses).

```
class Colour {
public:
  virtual ~Colour();
};

class Red : public Colour {
public:
  ~Red();      // Virtuality inherited from Colour
};

class LightRed : public Red {
public:
  ~LightRed();
};
```

Using these classes, we can define a *palette* as follows:

```
Colour *palette[3];
palette[0] = new Red;   // Dynamically create a new Red object
palette[1] = new LightRed;
palette[2] = new Colour;
```

The newly introduced operator new creates a new object of the specified type in dynamic memory and returns a pointer to it. Thus, the first new returns a pointer to an allocated object of class *Red* and assigns it to the first element of array *palette*. The elements of *palette* are pointers to *Colour* and, because *Red* is-a *Colour* the assignment is valid.

The contrary operator to new is delete which explicitly destroys an object referenced by the provided pointer. If we apply delete to the elements of *palette* the following destructor calls happen:

```
delete palette[0];
// Call destructor ~Red() followed by ~Colour()
delete palette[1];
// Call ~LightRed(), ~Red() and ~Colour()
delete palette[2];
// Call ~Colour()
```

The various destructor calls only happen, because of the use of virtual destructors. If we would have not declared them virtual, each delete would have only called ~ *Colour()* (because *palette[i]* is of type pointer to *Colour*).

87. function overloading

```
#include<iostream.h>
#include<conio.h>
class A
{
float v,r,h;
public:
void volume(float r)
{
v=(4*3.14*r*r*r)/3;
cout<<"volume of sphere is"<<v;
}
void volume(float r,float h)
{
v=3.14*r*r*h;
cout<<"volume of cylinder is"<<v;
}
};
void main()
{
A v1;
clrscr();
v1.volume(8);
v1.volume(8,6);
getch();
}
```

88. function overriding with virtual function

```
class A
{
public:
virtual void display()
{
cout<<"a\n";
}
};
class B:public A
{
public:
void display()
{
cout<<"b\n";
}
};
class C:public B
{
public:
void display()
{
cout<<"c\n";
}
};
void main()
 {
A a1,*p;
B b1;
C c1;
clrscr();
p=&a1;
p->display();
p=&b1;
p->display();
p=&c1;
p->display();
getch();
}
```

Note: program gives output as a a a. to get output we use virtual function which runs at late binding and we get output as a b c

Abstract Classes

Abstract classes are defined just as ordinary classes. However, some of their methods are designated to be necessarily defined by subclasses. We just mention their *signature* including their return type, name and parameters but not a definition. One could say, we omit the method body or, in other words, specify ``nothing''. This is expressed by appending ``= 0'' after the method signatures:

```
class DrawableObject {
  ...
public:
  ...
  virtual void print() = 0;
};
```

This class definition would force every derived class from which objects should be created to define a method *print()*. These method declarations are also called *pure methods*.

Pure methods must also be declared virtual, because we only want to use objects from derived classes. Classes which define pure methods are called *abstract classes*.

Operator Overloading

If we recall the abstract data type for complex numbers, *Complex*, we could create a C++ class as follows:

```
class Complex {
  double _real,
         _imag;
  public:
    Complex() : _real(0.0), _imag(0.0) {}
    Complex(const double real, const double imag) :
      _real(real), _imag(imag) {}
    Complex add(const Complex op);
    Complex mul(const Complex op);
    ...
  };
```

We would then be able to use complex numbers and to ``calculate'' with them:

```
Complex a(1.0, 2.0), b(3.5, 1.2), c;
c = a.add(b);
```

Here we assign *c* the sum of *a* and *b*. Although absolutely correct, it does not provide a convenient way of expression. What we would rather like to use is the well-known ``+" to express addition of two complex numbers. Fortunately, C++ allows us to *overload* almost all of its operators for newly created types. For example, we could define a ``+" operator for our class *Complex*:

```
class Complex {
  ...
public:
  ...
  Complex operator +(const Complex &op) {
    double real = _real + op._real,
           imag = _imag + op._imag;
    return(Complex(real, imag));
  }
  ...
};
```

In this case, we have made operator + a member of class *Complex*. An expression of the form

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

is translated into a method call

```
c = a.operator +(b);
```

Thus, the binary operator + only needs one argument. The first argument is implicitly provided by the invoking object (in this case *a*).

However, an operator call can also be interpreted as a usual function call, as in

```
c = operator +(a, b);
```

In this case, the overloaded operator is **not** a member of a class. It is rather defined outside as a normal overloaded function. For example, we could define operator + in this way:

```
class Complex {
  ...

public:
  ...

  double real() { return _real; }
  double imag() { return _imag; }

  // No need to define operator here!
};
Complex operator +(Complex &op1, Complex &op2) {
```

```
  double real = op1.real() + op2.real(),
        imag = op1.imag() + op2.imag();
  return(Complex(real, imag));
}
```

In this case we must define access methods for the real and imaginary parts because the operator is defined outside of the class's scope. However, the operator is so closely related to the class, that it would make sense to allow the operator to access the private members. This can be done by declaring it to be a *friend* of class *Complex*.

89.operator overloading(-)

```
#include<iostream.h>
#include<conio.h>
class A
{
int x,y,z;
public:
void getno(int a,int b,int c)
{
x=a;
y=b;
z=c;
}
void putno()
{
cout<<x<<"\t"<<y<<"\t"<<z;
}
operator -()
{
x=-x;
y=-y;
z=-z;
}
};
void main()
{
A a1;
clrscr();
a1.getno(10,20,30);
-a1;
a1.putno();
getch();
}
```

90. operator overloading(+)

```
#include<iostream.h>
#include<conio.h>
class complex
{
float x,y;
public:
complex(){  }
complex(float real,float img)
{
x=real;
y=img;
}
complex operator+(complex);
void display(void);
};
complex complex::operator+(complex c)
{
complex temp;
temp.x=x+c.x;
temp.y=y+c.y;
return(temp);
}
void complex::display(void)
{
cout<< x<<"+j"<<y<<"\n";
}
void main()
{
complex c1,c2,c3;
clrscr();
c1=complex(2.5,3.5);
c2=complex(1.6,2.7);
c3=c1+c2;
cout<<"c1= ";
c1.display();
cout<<"c2= ";
c2.display();
cout<<"c3= ";
c3.display();
}
```

91.program to overload != operator

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

class vectors
{
 int a,b,c;
 public:
 vectors(int x,int y,int z)
 {
 a=x;
 b=y;
 c=z;
 }
 int operator !=(vectors v)
 {
  if(a!=v.a||b!=v.b||c!=v.c)
  {
  return 1;
  }
  else
  {
  return 0;
  }
 }

};
 void main()
 {
 clrscr();
 vectors v1(10,20,30),v2(10,-20,30);
 if(v1!=v2)
 {
 cout<<"They are not equal"<<endl;
 }
 else
 {
 cout<<"They are equal"<<endl;
 }
 getch();
 }
```

Friends

We can define functions or classes to be friends of a class to allow them direct access to its private data members. For example, in the previous section we would like to have the function for operator + to have access to the private data members *_real* and *_imag* of class *Complex*. Therefore we declare operator + to be a friend of class *Complex*:

```
class Complex {
  ...

public:
  ...

  friend Complex operator +(
    const Complex &,
    const Complex &
  );
};

Complex operator +(const Complex &op1, const Complex &op2) {
  double real = op1._real + op2._real,
         imag = op1._imag + op2._imag;
  return(Complex(real, imag));
}
```

You should not use friends very often because they break the data hiding principle in its fundamentals. If you have to use friends very often it is always a sign that it is time to restructure your inheritance graph.

92.friend function -examples

```
#include<iostream.h>
#include<conio.h>
class sample
{
int a;
int b;
public:
void setvalue()
{
a=25,b=40;
}
friend float mean(sample s);
 };
 float mean(sample s)
 {
 return float(s.a+s.b)/2.0;
 }
void main()
{
sample x;
clrscr();
x.setvalue();
cout<<"mean value:"<<mean(x)<<"\n";
getch();
}
```

93. Example

```
#include<iostream.h>
#include<conio.h>
class abc;
class xyz
{
int x;
public:
void setvalue(int i)
{
x=i;
}
friend void max(xyz,abc);
 };
 class abc
{
int a;
public:
void setvalue(int i)
{
a=i;
}
 friend void max(xyz,abc);
 };
 void max(xyz m,abc n)
 {
 if(m.x>=n.a)
 {
 cout<<m.x;
 }
 else
 cout<<n.a;
 }
void main()
{
clrscr();
abc ob;
ob.setvalue(10);
xyz ob1;
ob1.setvalue(20);
max(ob1,ob);
getch();
}
```

94.Assignment- to print complex values

```
#include<iostream.h>
#include<conio.h>
class complex
{
float x,y;
public:
void input(float real,float img)
{
x=real;
y=img;
}
friend complex sum(complex,complex);
void  show(complex);
 };
complex sum(complex c1,complex c2)
{
complex c3;
c3.x=c1.x+c2.x;
c3.y=c1.y+c2.y;
return(c3);
}
void complex::show(complex c)
{
cout<<c.x<<"+j"<<c.y<<"\n";
}

void main()
{
clrscr();
complex A,B,C;
A.input(3.1,5.26);
B.input(2.75,1.2);
C=sum(A,B);
cout<<"A=";
A.show(A);
cout<<"B=";
B.show(B);
cout<<"C=";
C.show(C);
getch();
}
```