Introduction

- C++ is developed by Bjarne Stroustrup.
- Initially this language was known as "C with Classes".
- In 1983 it was renamed to C++.
- The ++ Operator indicated increment in C language.
- C++ is an object oriented programming language.
- An object oriented programming language is a language in which we can completely define a physical or virtual entity in the form of class and can create multiple objects of the class.
- In one object we can store data of one entity and we can perform different operations on that object.

C++ Program Structure | Paradigm

- Paradigm means a model of something, or a very clear and typical example of something
- **Syntax:**

```
preprocessor commands
global declaration
main()
{
        statements
        ....
        ....
}
```

Where,

1. **Preprocessor Directives**

- Preprocessor or C++ Preprocessor is a program which process the source code before the compiler starts compiling the source code.
- The preprocessor performs tasks like replacing the macros, invokes linker to link the required header files, performs task related to conditional compilation, etc.
- The preprocessor produces a new code after making changes or replacements.
- This new code is called as Expanded/Intermediate Source Code.
- Preprocessor directives are not translated into machine level language.
- Every preprocessor directive begins with a hash (#).
- Some preprocessor directives are as follows:

Preprocessor Directive	Description
#include	▪ It includes the specified header file in the C/C++ program. ▪ Example: #include<iostream.h> #include "myheaderfile.h"
#define	▪ It defines a macro. ▪ All the macros are replaced by its definition before the compiler begin compilation. ▪ Example: #define PI 3.14

2. **Global Declaration**

- In this section, we can define variables, functions, structures, enum, union, classes, templates, etc.
- Everything defined here can be used anywhere in the remaining program.

3. **main()**

- Execution of C++ program always begins with function main().
- It also receives the command line arguments.
- The function main() returns an int value which indicates How the program terminated?
- If the program terminated successfully then main() must return 0 to the operating system.
- 0 indicates no error occurred during execution of program otherwise a non-zero value is returned.
- The non-zero value indicated which type of error occurred in the program during execution.

Input-Output objects

Function/Object	Description
cout	▪ The object cout is used to print a text, value of a vriable or output of the program. ▪ With cout we need to use an insertion operator (<<) ▪ The object cout is of class ostream ▪ This object is available in the header file <iostream.h> ▪ **Syntax:** ▪ **cout<<"message";** ▪ **cout<<variable_name;**
cin	▪ The object cin is used to read the values entered by the user. ▪ With cin we need to use an extraction operator (>>) ▪ The object cin is of class istream ▪ This object is available in the header file <iostream.h> ▪ **Syntax:** **cin>>variable1>>variable2>>...;**
'\n' or <<endl	It prints a new line characters. It indicates end of line.

Example: WAP to display a Welcome message.

```
#include<iostream>
using namespace std;
main()
{
        cout<<"Welcome to C++"<<endl;
        cout<<"Amravati"<<endl;
}
```

Output:
Welcome to C++
Amravati

C++ Data-types

- The data-type indicates the type of the value.
- C++ provides different built-in (i.e. readymade) data-types as follows:
- **The memory size and storage capacity of data-types may change as per different compilers.**

1. Data-Types to indicate numbers without decimal point

Data Type Name	Memory Size	Range
short	1 byte	-128 to 127
int	2 bytes	-32,768 to 32,767
long	4 bytes	-35,708 to +65,767

2. Data-Types to indicate numbers with decimal point

Data Type Name	Memory Size	Range
float	4 bytes	-3.4e38 to +3.4e38
double	8 bytes	-1.7e308 to +1.7e308

3. Data-Types to indicate alphabets and symbols

Data Type Name	Memory Size	Range
char	1 byte	-128 to 127

4. void Data-Type

- void is an empty data type that has no value.
- This can be used in functions and pointers.

sizeof() function

- The sizeof() function is used to find the memory space allocated for specified data-type.
- **The memory size and storage capacity of data-types may change in different compilers**
- Example: cout<<"Size of int = "<<sizeof(int)<<" bytes"<<endl;

FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE

Variables

- The variable is a memory location with specified name.
- The variable is used to store values of different data-types.
- All variables must be declared before they are used.
- A variable can be defined as follows:
- **Syntax:** **data_type variable_name;**

Example:
int a=5, b=7;
float pi = 3.1416;
double p, q, r;

Operators

1. Arithmetic Operators:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division (To find quotient)
%	Division (To find remainder)

Operator	Meaning
++	Increment Operator
--	Decrement Operator

2. Assignment, Compound Assignment and Shorthand Operators

Operator	Expression	Meaning
=	int a = 10;	It will assign a right side value to the left-side variable.
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

3. Relational | Comparison Operators:

Operator	Meaning
<	Less than
<=	Less than or equals to
>	Greater than
>=	Greater than or equals to
==	Exact equals to
!=	Not equals to

4. Logical Operators:

Logical operators are used if we want to check multiple conditions simultaneously

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Comments

1. Comments are used to indicate the purpose of code segment in the program.
2. The comments are always ignored by the compiler and comments are not executed.
3. The comments are not converted in the machine level code.
4. The comments can be placed anywhere in the program.
5. The comments can be specified in two ways as follows:
   - Single Line Comment (Syntax: // comment_text)
   - Multi Line Comment (Syntax: /*comment_text*/)

Example: Write a program to read three numbers and find sum and mean (i.e. average) of them.

```
#include<iostream>
using namespace std;
main()
{       int a, b, c;
        cout<<"Enters 3 numbers: ";
        cin>>a>>b>>c;
        int s = a + b + c;
        cout<<"Sum is "<<s<<endl;
        float m = s / 3.0;
        cout<<"Mean is "<<m<<endl;
}
```

Output:
Enters 3 numbers: 5 6 8 <Enter>
Sum is 19
Mean is 6.33333

Example: Write a program to read L & B of a rectangle and find its area and perimeter.

```
#include<iostream>
using namespace std;
main()
{
        int L, B;

        cout<<"Enter L and B of a rectangle: ";
        cin>>L>>B;

        int A = L * B;
        cout<<"Area is "<<A<<endl;

        int P = 2 * (L + B);
        cout<<"Perimeter is "<<P<<endl;
}
```

Output:
Enter L and B of a rectangle: 5 7 <Enter >
Area is 35
Perimeter is 24

Example: WAP to read Date of birth of a person and display it.

```
#include<iostream>
using namespace std;
main()
{
        int D, M, Y;
        cout<<"Enter D.O.B. (e.g. 4/5/2015): ";
        cin>>D;
        cin.ignore();
        cin>>M;
        cin.ignore();
        cin>>Y;
        cout<<"D.O.B. = "<<D<<"."<<M<<"."<<Y<<endl;
}
```

Output:
Enter D.O.B. (e.g. 4/5/2015): 11/5/1988 <Enter>
D.O.B. = 11.5.1988

cin.getline() function:

Function	Description
istream& getline (char* s, int N)	▪ This function is used to read a complete line of text from the console. ▪ It stops reading when N characters are completed or '\n' occurred.
istream& getline (char* s, int N, char delim)	▪ This function is used to read multi-line text from the console. ▪ It stops reading when N characters completed or the specified delimiter and '\n' occurred.
cin.sync();	▪ It clears or flushes the input buffer

Example: What is the use of getline() function in c++? Explain the getline() function with two arguments and three arguments with suitable example.

```
#include<iostream>
using namespace std;
main()
{       char N[50], A[100];
        cout<<"\nEnter your full name: "<<endl;
        cin.getline(N, 50);
        cout<<"\nEnter your address, enter # at end: "<<endl;
        cin.getline(A, 100, '#');
        cout<<"\nName = "<<N<<endl;
        cout<<"\nAddress = "<<A<<endl;
}
```


FOR EDUCATIONAL PURPOSE ONLY, NOT FOR SALE

ASCII Values

- ACSII Stands for: American Standard Codes for Information Interchange.
- ASCII values are invented by ANSI (i.e. American National Standards Institute).
- The ASCII values are as follows:
  A = 65, B = 66, C = 67, D = 68, E = 69, ...
  a = 97, b = 98, c = 99, d = 100, e = 101, ...
  '\n' = 10, 0 = 48, 1 = 49, ...

Type Casting | Type Conversion

- Conversion of one type of value into another data-type is called as type casting.
- C++ supports two types of type casting.
  1. Implicit Type Casting
  2. Explicit Type Casting

1. Implicit Type Casting

- The typecasting which is automatically performed by the C++ Compiler is called as implicit type-casting.

Example:

```
#include<iostream>
using namespace std;
main()
{
        char z = 65;
        cout<<"z = "<<z<<endl;
}
```

Output:
z = A

Example:

```
#include<iostream>
using namespace std;
main()
{
        int a = 3.14;
        cout<<"a = "<<a<<endl;
}
```

Output:
a = 3

2. Explicit Type Casting

- The type-casting specified by the programmer to the compiler is called as explicit type-casting.
- **Syntax: (data_type) variable;**

Example:

```
#include<iostream>
using namespace std;
main()
{
        int z = 65;
        cout<<"z = "<<(char)z<<endl;
        z++;
        cout<<"z = "<<(char)z<<endl;
}
```

Output:
z = A
z = B

Example: Write a program to read two integer values and find the quotient.

```
#include<iostream>
using namespace std;
main()
{
        int a, b;
        cout<<"Enter 2 numbers: ";
        cin>>a>>b;
        float c = a / b;
        cout<<"Quotient is "<<c<<endl;
        float d = a / (float)b;
        cout<<"Quotient is "<<d<<endl;
}
```

Output:
Enter 2 numbers: 9 2 <Enter>
Quotient is 4
Quotient is 4.5

Conditional Statements | Decision making statements

1. if - else statement
2. switch statement

if - else Statement

- The if statement is used to check a condition and execute an appropriate block of code as per the result of the condition.
- A condition can be specified by using relational and/or logical operators.

Working:

- Here first the given condition is checked.
- If the result of the condition is true then the code within the **if block** gets executed and then the program control transfers to the first statement after else block.
- If the result of the condition is false then the code within the **else block** gets executed and then program control transfers to the first statement after else block.
- **The else block is optional.**

- **Syntax:**

```
if(condition)
{
        ....
        ....
}
else
{
        ....
        ....
}
```

Note:

1. In C and C++, 0 (zero) means false and any non-zero value means true. For simplicity 1 is considered for true.
2. The { and } (i.e. opening and closing braces) must be specified if we want to execute more than one line of code inside if or else.
3. The { and } (i.e. opening and closing braces) are optional if we want to execute only one line of code inside if or else.
4. If { and } (i.e. opening and closing braces) are not specified with if or else then by default one line of code is attached with if or else.

Example: Write a program to read age of a person and check the person is eligible to vote or not.

```
#include<iostream>
using namespace std;
main()
{
        int a;
        cout<<"Enter your age: ";
        cin>>a;
        if(a >= 18)
        cout<<"Eligible for vote"<<endl;
        else
        cout<<"Not eligible for vote"<<endl;
}
```

Output:

```
Enter your age: 10
Not eligible for vote
```

Example: Write a program to read three angles and check the triangle is isosceles triangle or not.

```
#include<iostream>
using namespace std;
main()
{
        int a, b, c;
        cout<<"Enter 3 angles in degrees: ";
        cin>>a>>b>>c;
        if((a + b + c == 180) && (a == b || b == c || c == a))
        cout<<"It is an isosceles triangle"<<endl;
        el e
        cout<<"Not an isosceles triangle"<<endl;
}
```

Output:

```
Enter 3 angles in degrees: 30 60 30 <Enter>
Not an isosceles triangle
```

Nested If

- If an **"if" statement** is used **within another "if" block** then such a control structure is called as nested if.

- **Example:**

```
if(condition)
{
        if(condition)
        ....
        if(condition)
        ....
}
else
{
        ....
        ....
}
```


Example: WAP to read a number and check it is positive even number or positive odd number.

```
#include<iostream>
using namespace std;
main()
{
        int n;
        cout<<"Enter a number: ";
        cin>>n;
        if(n > 0)
        {
                if(n % 2 == 0)
                cout<<"It is positive even number."<<endl;
                else
                cout<<"It is positive odd number"<<endl;
        }
        else
        cout<<"Number is negative."<<endl;
}
```

Ladder Structure | if-else-if

- If an **"if" statement** is used **inside the "else"** block then such a control structure is called as ladder structure or if-else-if.

- **Example:**

```
if(condition)
{
        ....
        ....
}
else
        if(condition)
        {
                ....
                ....
        }
```

Example: Write a program to read a number and check the entered number is zero or positive or negative

```
#include<iostream>
using namespace std;
main()
{
        int n;
        cout<<"Enter a number: ";
        cin>>n;

        if(n == 0)
        cout<<"It is zero"<<endl;
        else
                if(n > 0)
                cout<<"It is a positive number"<<endl;
                else
                cout<<"It is a negative number"<<endl;
}
```

Output:

```
Enter a number: -1      <Enter>
It is a negative number
```

Loops | Iterative Statements

- Loops are used to repeatedly execute a block of code.
- C++ provides three loops as follows:
  1. **while loop**
  2. **for loop**
  3. **do-while loop**

while loop

- This loop is used to repeatedly executed a block of code while the specified condition is true.
- The program control exit from the loop when the specified condition becomes false.

Syntax:

```
while(condition)
{
        ....
        ....
}
```

Working:

- Here, first the given condition is checked.
- if the condition is true then the program control enter inside the loop and executes the code present in that loop. After the execution of code, the program control move to check the condition.
- if the condition is true then the above process repeats.
- If the condition is false then the program control exit from the loop.

Note:

- The { and } (i.e. opening and closing braces) must be provided if we want to execute more than one line of code inside the loop.
- The { and } (i.e. opening and closing braces) are optional if we want to execute only one line of code inside any loop.
- If { and } (i.e. opening and closing braces) are not specified with any loop then by default one line of code is attached with that loop.

Example: Write a program to read an integer. If the entered integer is even then print all even numbers in the range 1 to the given integer. If the entered integer is odd then print all odd numbers in the range 1 to the given integer.

```
#include<iostream>
using namespace std;
main()
{
        int n;
        cout<<"Enter a number: ";
        cin>>n;

        int i = 0;
        if(n % 2 == 0)
        i = 2;
        else
        i = 1;

        while(i <= n)
        {
                cout<<i<<"\t";
                i += 2;
        }
}
```

Output:

```
Enter a number: 9       <Enter>
1    3    5    7    9
```

Example: WAP to read a number and check it is palindrome or not

```
#include<iostream>
using namespace std;
main()
{
        int n;
        cout<<"Enter an integer: ";
        cin>>n;
        int p = n;
        int s = 0;

        while(n > 0)
        {
                int z = n % 10;
                s = (s * 10) + z;
                n = n / 10;
        }

        if(s == p)
        cout<<"It is palindrome"<<endl;
        else
        cout<<"It is not palindrome"<<endl;
}
```

Output:

```
Enter an integer: 1221 <Enter>
It is palindrome
```


FOR EDUCATIONAL PURPOSE ONLY, NOT FOR SALE

for Loop

- This loop is also used to repeatedly execute a block of statements until the specified condition becomes false.
- In for loop the initialization, condition and evaluation (i.e. update) is specified at the starting of the loop.
- Also this loop supports multiple initialization and evaluation of variables.

Syntax:

```
for(initialization; condition; update)
{
        statements
        ....
        ....
}
```

Working:

- Here, first the initialization of loop variable takes place only one time.
- Then the given condition is checked.
- If the condition is true then the program control enter inside the loop and executes the code in that loop.
- After the execution of code, the program control move to update the value of the loop variable.
- After updating the value of loop variable it again checks the condition.
- If the condition is true then above process repeats otherwise, the program control exit from the loop.

Note:

- The { and } (i.e. opening and closing braces) must be provided if we want to execute more than one line of code inside the loop.
- The { and } (i.e. opening and closing braces) are optional if we want to execute only one line of code inside any loop.
- If { and } (i.e. opening and closing braces) are not specified with any loop then by default one line of code is attached with that loop.

Example: Write a program to read an integer and find its factorial.

```
#include<iostream>
using namespace std;
main()
{
        int n;
        cout<<"Enter an integer: ";
        cin>>n;
        int f = 1;

        for(int i = 1; i <= n; i++)
        f = f * i;

        cout<<"Factorial is: "<<f<<endl;
}
```

Output:

```
Enter an integer: 5 <Enter>
Factorial is: 120
```

Multiple Initialization and Evaluation of loop variables

- We can initialize and evaluate multiple loop variables with the for loop separated by comma (,) operator.
- Here, it is not compulsory to specify loop condition to all the loop variables.
- We can specify the loop condition to any one loop variable.
- If we want to specify loop condition to all the variables then the condition must be separated by using && (i.e. logical and) operator.

Example:

```
#include<iostream>
using namespace std;
main()
{
for(int i = 1, j = 5; i <= 5 && j >= 1; i++, j--)
cout<<i<<"\t"<<j<<endl;
}
```

OR

```
main()
{
for(int i = 1, j = 5; i <= 5; i++, j--)
cout<<i<<"\t"<<j<<endl;
}
```

Output:

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

Nested Loops

- If a loop is used inside a loop then such a control structure is called as nested loops.
- We can use any loop inside any other or same loop

Example: Write a program to print the following output.

```
1
22
333
4444
55555
```

```
#include<iostream>
using namespace std;
main()
{
        for(int i = 1; i <= 5; i++)
        {
                for(int j = 1; j <= i; j++)
                cout<<i;
                cout<<endl;
        }
}
```

Example: Write a program to print the following output.

```
    1
   212
  32123
 4321234
543212345
```

```
#include<iostream>
using namespace std;
main()
{
        for(int p = 5, q = 1; p >= 1; p--, q++)
        {
                for(int i = 1; i <= p; i++)
                cout<<"-";

                for(int j = q; j >= 1; j--)
                cout<<j;

                for(int k = 2; k <= q; k++)
                cout<<k;

                cout<<endl,
        }
}
```

do-while loop

- This loop is also used to repeatedly execute a block of code while the specified condition is true.
- The program control exit from the loop when the specified condition gets false.
- This loop is specially used when we want to execute a block of code for at least one time.

Working:

- Here, first the program control directly enters in the loop, due to this the code within the loop gets executed.
- Then, it checks the condition, if the condition is true then the program control re-enters in the loop and repeats the above steps.
- If the condition is false then the program control exit from the loop.

Syntax:

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

Example: Write a program to read a number and print all numbers from 1 to the entered integer. Repeat this process while the entered integer is greater than zero.

FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE

```
#include<iostream>
using namespace std;
main()
{       int n;
        do
        {
                cout<<"Enter an integer: ";
                cin>>n;
                for(int i = 1; i <= n; i++)
                cout<<i<<"\t";
                cout<<endl;
        }
        while(n > 0);
}
```

Output:

```
Enter an integer: 5      <Enter>
1    2    3    4    5
Enter an integer: 4      <Enter>
1    2    3    4
Enter an integer: -1
```

Jump Statements

- C++ provides following jump statements:
  1. break
  2. continue
  3. goto

1. break Statement:

- The break statement is used to exit from any loop before the loop condition get false.
- Also, break statement is used to exit from switch statement.
- As soon as the break statement is executed the program control moves to the first statement after the loop or switch statement.

```
while (condition)
{
        ....
        break;
        ....
}
```

```
for (initialization; condition; update)
{
        ....
        break;
        ....
}
```

```
do
{
        ...
        break;
        ...
}
while (condition);
```

```
switch (variable)
{
        case val_1 :
                ....
                ....
                break;
        case val_n :
                ....
                ....
                break;
        default:
                ....
                ....
}
```

Example:

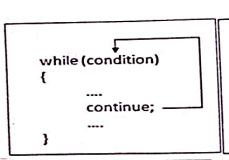
```
#include<iostream>
using namespace std;
main()
{
        for(int i = 1; i <= 5; i++)
        {
                cout<<i<<"\t";
                if(i == 2)
                break;
        }
        cout<<"\nProgram Finished"<<endl;
}
```

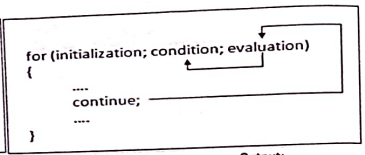
Output:

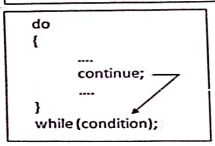
```
1    2
Program Finished
```

2. continue Statement:

- The continue statement is used to start the next iteration of any loop before the execution of all the code in the loop.
- The continue statement transfers the program at different points in different loops as follows.







Example:

```
#include<iostream>
using namespace std;
main()
{
        for(int i = 1; i <= 15; i++)
        {
                if(i % 3 != 0)
                continue;

                cout<<i<<"\t";
        }
}
```

Output:
3 6 9 12 15

3. goto Statement:

- A goto statement transfers program control to anywhere only inside the same function.
- Also, it allows the program control to enter or exit from the loop or switch statement.
- The syntax for goto statement is given below.
- **Syntax: goto label;**
- The identifier label can be defined by using following syntax.
- **Syntax: label:**

Example:

```
#include<iostream>
using namespace std;
main()
{       cout<<"C"<<endl;
        goto ABC;

        PQR:
        cout<<"JAVA"<<endl;
        goto XYZ;

        ABC:
        cout<<"C++"<<endl;
        goto PQR;

        XYZ:
        cout<<"WELCOME"<<endl;
}
```

Output:

```
C
C++
JAVA
WELCOME
```

switch Statement

- The switch statement is used to check an int or char variable to a list of values and to execute an appropriate block of code (case block) as per the value of the variable.

Working:

- Here, the value of the variable is compared with each case block.
- The case block with the value of the variable is matched gets executed.
- If the value of variable is not matched with any case block then the code inside the default block is executed.
- The default block is optional.
- The cases and default block can be specified in any sequence.
- We can use the break statement at the end of each case block to transfer the program control out of the switch.
- If the break statement is not used then it will execute all the code from the matching case block up to the end of the switch statement.

Syntax:

```
switch(variable)
{
        case val_1:     ....
                        break;
        case val_2:     ....
                        break;
        case val_n:     ....
                        break;
        default:    ....
}
```


Example: Write a program to print appropriate message as per the choice entered. **(1 : Hindi, 2 : Marathi, 3 : English)**

```
#include<iostream>
using namespace std;
main()
{       int n;
        cout<<"Enter valid language: ";
        cin>>n;
        switch(n)
        {
            case 1:  cout<<"You Selected Hindi"<<endl; break;
            case 2:  cout<<"You Selected Marathi"<<endl; break;
            case 3: cout<<"You Selected English"<<endl; break;
            default: cout<<"Invalid Option"<<endl;
        }
}
```

Output:

```
Enter valid language: 1
You Selected Hindi
```

Functions

- The function is a special block of code which is defined to perform a specific task.
- The function works on the principle, write once and use whenever required.
- It avoids retyping and repetition of code.
- The functions are of two types as follows:
  1. Function returning value.
  2. Function not returning value.

Function returning value

- Such a function send back its result to the point where the function was called.
- So, with such function we should use the operators like +, -, =, <, >, == etc.
- In simple words, we can say that if the function returns a value then we can perform additional operations with the result of the function.
- For Example: We can print, add or subtract or multiply something with the result.

Example:

```
#include<iostream>
using namespace std;
#include<math.h>
main()
{       double x, y;
        x = sqrt(81);
        cout<<"x = "<<x<<endl;
        y = 100 + pow(2, 3);
        cout<<"y = "<<y<<endl;
}
```

Output:

```
x = 9
y = 108
```

Function not returning value

- Such function perform its task but not send back (i.e. return) its result.
- So, we cannot use the operators like +, -, =, <, >, ==, etc. with such function.

Example:

```
#include<conio.h>
main()
{       clrscr();
        gotoxy(20, 10);
        textcolor(LIGHTRED + BLINK);
        textbackground(WHITE);
        cprintf("Compilers\n\r");
        getch();
}
```

Defining New Function | User-defined Functions

- We can write the reusable code inside a function instead of writing that code again and again at different places.
- Once a function is defined then, we can call that function whenever required any number of times.
- If we want to use the function in an expression then we must define a function which will return value.
- If we don't want to use the function in an expression then we can define a function which will not return value.
- **Syntax:**

```
return_type function_name(data_type arg1, data_type arg2, ... )
{
        ....
        return(result);
}
```

Where:

return_type	▪ It indicates the data_type of the value the function will return as its result. ▪ If the function is not returning value then for such function the return_type must be void. ▪ In C and C++, if the return_type is not specified to a function then the default retyrn-type is int.
return	▪ The return statement is used to send back the result of the function to the point where the function was called. ▪ Whenever the return statement is executed, the program control exit from the function and go back to the point where the function was called. ▪ **Syntax:** 1. **return;** 2. **return(value);**

Example: Design a function print_line() which will print one star and one period (i.e. *.) 20 times.

```
#include<iostream>
using namespace std;
void print_line()
{
        for(int i = 1; i <= 20; i++)
        cout<<"*.";
        cout<<endl;
}
```

```
main()
{
        print_line();
        cout<<"\tWelcome To C++"<<endl;
        cout<<"\tAmravati"<<endl;
        print_line();
}
```

Output:

```
*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.
        Welcome To C++
        Amravati
*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.
```

Parameterized function | Function with arguments

- If required, while calling the function we can pass one or more values (i.e. actual arguments) to the function.
- These values are transferred as formal arguments to the function where according to the formal argument's value the function performs its task.

Examples on Function not returning value

Example: Design a function square() which will calculate & print square of the number which is passed as argument.

```
#include<iostream>
using namespace std;
void square(int n)
{
        int sq = n * n;
        cout<<"Square of "<<n<<" is "<<sq<<endl;
}
```

```
main()
{
        square(9);
        square(10);
}
```

Output:

```
Square of 9 is 81
Square of 10 is 100
```

Example: Write a program to read a number and design a function factorial() which calculate and print factorial of the number which is passed as argument.


```
#include<iostream>
using namespace std;
void factorial(int n)
{
        long int f = 1;
        for(int i = 1; i <= n; i++)
        f = f * i;

        cout<<"Factorial of "<<n<<" is "<<f<<endl;
}
```

```
main()
{
        int N;
        cout<<"Enter a number: ";
        cin>>N;
        factorial(N);
}
```

Output:

```
Enter a number: 5
Factorial of 5 is 120
```

Examples on function returning value

Example: Design a function square() which will return square of the number which is passed as argument.

```
#include<iostream>
using namespace std;

int square(int n)   //Here, variable n is Formal Argument
{
    int sq = n * n;
    return(sq);
}

main()
{
    int x = square(9);  //Here, 9 is Actual Argument
    cout<<"x = "<<x<<endl;
}

Output:
x = 81
```

Example: Design a function isprime() which will reurn true (i.e. 1) if the argumentd number is prime otherwise false (i.e. 0) will be return.

```
#include<iostream>
using namespace std;
int isprime(int x)
{
        int c = 0;
        for(int i=1; i <= x ; i++)
        {
                if(x % i == 0)
                c++;
        }

        if(c == 2)
        return(1);
        else
        return(0);
}
```

```
main()
{
        int n;
        cout<<"Enter a number: ";
        cin>>n;

        if(isprime(n))
        cout<<"It is prime number"<<endl;
        else
        cout<<"It is not prime number"<<endl;
}
```

Nesting of function

- If a function contains call to another function then such mechanism is called as nesting of function.
- It avoids the retyping and repetition of same code in multiple functions.

Example: Design two function sum() and mean() which will return sum and mean (i.e. average) when three numbers are passed as argument.

```
#include<iostream>
using namespace std;
int sum(int a, int b, int c)
{
        return(a + b + c);
}

float mean(int p, int q, int r)
{
        float m = sum(p, q, r) / 3.0;
        return(m);
}
```

```
main()
{
        int x = sum(5, 6, 8);
        cout<<"Sum is "<<x<<endl;

        float y = mean(5, 6, 8);
        cout<<"Mean is "<<y<<endl;
}
```

Output:
Sum is 19
Mean is 6.33

Recursive Function

- If a function contains call to itself then such function is called as recursive function.
- Note that the statement which contains call to the same function must be specified conditionally otherwise the function will call the same function until the buffer memory gets full.

Example: Design a recursive function fact() which returns factorial of the number which is passed as argument.

```
#include<iostream>
using namespace std;
long int fact(int n)
{
        if(n == 1)
        return(1);
        else
        return(n * fact(n - 1));
}
```

```
main()
{
int n;
cout<<"Enter a number: ";
cin>>n;

long int f = fact(n);
cout<<"Factorial of "<<n<<" is "<<f<<endl;
}
```

Output:
Enter a number: 5 <Enter>
Factorial of 5 is 120

Function Overloading

- Defining multiple functions in a program with same name but different number of arguments or different data-types of arguments is called as function overloading.
- In such case an appropriate function is called as per the values passed while calling the function.

Example:

```
#include<iostream>
using namespace std;
void display(int n)
{
        cout<<"n = "<<n<<endl;
}

void display(double n)
{
        cout<<"n = "<<n<<endl;
}
```

```
void display(char n[])
{
        cout<<"n = "<<n<<endl;
}

main()
{
        display(10);
        display(15.34);
        display("JACK");
}
```

Output:
n = 10
n = 15.34
n = JACK

FOR EDUCATIONAL PURPOSE ONLY, NOT FOR SALE

Example: Overload function product(), so that it can be call by passing 4, 3 & 2 arguments.

```
#include<iostream>
using namespace std;
int product(int a, int b, int c, int d)
{
        return(a * b * c * d);
}

int product(int a, int b, int c)
{
        return(a * b * c);
}

int product(int a, int b)
{
        return(a * b);
}
```

```
main()
{
        int x = product(1, 2, 3, 4);
        cout<<"x = "<<x<<endl;

        int y = product(5, 6, 7);
        cout<<"y = "<<y<<endl;

        int z = product(8, 9);
        cout<<"z = "<<z<<endl;
}
```

Output:
x = 24
y = 210
z = 72

Function with Default (i.e. Optional) Arguments

- If a function contains default arguments then in some cases it avoids the need to overload the function resulting to reduce the length of program.
- The function with default argument can be call by passing different number of arguments.
- Note that the default values must be specified in right to left direction.

Example: Design a function product() which can be call by passing 4, 3 and 2 arguments.

```
#include<iostream>
using namespace std;
int product(int a, int b, int c=1, int d=1)
{
        return(a * b * c * d);
}
```

```
main()
{       int x = product(1, 2, 3, 4);
        cout<<"x = "<<x<<endl;

        int y = product(5, 6, 7);
        cout<<"y = "<<y<<endl;

        int z = product(8, 9);
        cout<<"z = "<<z<<endl;
}
```

Output:
x = 24
y = 210
z = 72

Function Prototype | Function Signature

- If a function is called before its definition then we need to define the prototype of that function before the function call.
- The prototype can be define by using following syntax.
- **Syntax:** **return_type function_name (datatype_of_arg1, datatype_of_arg2, ...);**
- **Note:**
- If the function contains default arguments then the default value must specify in the function prototype only.

Example: Design a function product() which can be call by passing 4, 3 and 2 arguments.

```
#include<iostream>
using namespace std;
int product(int, int, int = 1, int = 1);
main()
{       int x = product(1, 2, 3, 4);
        cout<<"x = "<<x<<endl;
        int y = product(5, 6, 7);
        cout<<"y = "<<y<<endl;
        int z = product(8, 9);
        cout<<"z = "<<z<<endl;
}
```

```
int product(int a, int b, int c, int d)
{
        return(a * b * c * d);
}
```

Output:
x = 24
y = 210
z = 72

Object Oriented Programming

Object:

- Every imaginary and non-imaginary thing (i.e. entity) which we can describe programmatically is called as an object.

Object Oriented Programming Language:

- The programming language which provides us features to define objects in the form of class and create objects is called as object oriented programming language.

Features of Object Oriented Language:

1. **Data Abstraction and Encapsulation:**

   Abstraction:
   - This feature indicates that here we can describe any imaginary and non-imaginary entity programmatically and can hide its unnecessary internal details from the user and can reveal only the necessary details to the user.

   Encapsulation:
   - Encapsulation means to bind something in a group.
   - The ability to bind the information and its related functions into a single entity is called as encapsulation.

2. **Inheritance:**
   - Reusability of existing classes without disturbing the existing classes is called as inheritance.

3. **Polymorphism:**
   - Polymorphism indicates similar behavior of objects of different class types.
   - It is a technique to develop a code which will work for the objects of different classes whose behavior is same.

class

- The class is the programatical description of an entity.
- The class is a user-defined data-type where we can bind (i.e. encapsulate) basic information and operations related with that entity / object together.

Syntax:

```
class class_name
{
        access_specifier:
                data_members
                ----
        access_specifier:
                member_functions();
                ----
};
```

Where:

data_member	• It indicates the basic information of an entity. • The data-member is also called as property or field. • The data member can be define by using following syntax. • **Syntax:** `data_type data_member_name;`

member_function	• It indicates the operations which can be perform on the object.
	• The member function can be define by using following syntax.
	• **Syntax:**
	`return_type function_name(data_type arg1, data_type arg2, ... )`
	`{`
	`...`
	`return(value);`
	`}`

Creating Object | Instance

- In one object we can store data of one entity. If we have two entities then we require two objects.
- In other words ,we can say that, one object represents one entity.
- An object can be created as follows.
- **Syntax:** `class_name object_name;`

Note:

- Once a class is defined then we can create multiple objects of that class.
- **When an object is created then the space is created inside each object for all the non-static data members of that class.**

Accessing object members

- The members of the object can be access by using dot ( ) operator.
- The dot operator is also called as period operator.
- **Syntax:** `object_name.member_name;`

Access Specifiers

- Access specifier indicates the accessibility/scope of a class member.
- C++ provides three access specifiers private, public and protected.
- If the access specifier is not specified then it is by default **private**.

private	• If a data-member or member-function is declared as private then such member can be used inside that class only. That means we cannot use such member outside the class. • Generally the members which contain sensitive data are declared as private.
public	• If a data-member or member-function is declared as public then such member can be used inside the class as well as outside the class.
protected	• If a data-member or member-function is declared as protected then such member can only be access inside the class in which it is defined and in all the classes derived from that class.

Example: Design a class Worker with data members name, wage and working days. Define member function payment().

```
#include<string.h>
#include<iostream>
using namespace std;
class Worker
{
        public:
                char N[20];
                int W, WD;
        public:
        void payment()
        {
        int P = W * WD;
        cout<<"\nWorker Name = "<<N<< "\n";
        cout<<"Payment = "<<P<< Rs."<< "\n";
        }
};

main()
{
Worker a, b;

strcpy(a.N, "JACK");
a.W = 100;
a.WD = 10;

strcpy(b.N, "JAMES");
b.W = 200;
b.WD = 20;

a.payment();
/*Find payment by using data of object-a*/

b.payment();
/*Find payment by using data of object-b*/
}
```

Output:

```
Worker Name = JACK
Payment = 1000 Rs.

Worker Name = JAMES
Payment = 4000 Rs.
```

Setter function:

- We know that, we cannot access the private members outside the class so, to set the values of private members we can define setData() function or constructors in the class.

Example: Define a class Worker with data-members name, wage and working-days. Define member f
setdata() and payment();

```
#include<string.h>
#include<iostream>
using namespace std;
class Worker
{
        public:
                char N[20];
                int W, WD;
        public:
        void setdata(char n[], int w, int wd)
        {
                strcpy(N, n);
                W = w;
                WD = wd;
        }

        void payment()
        {
        int P = W * WD;
        cout<<"\nWorker Name = "<<N<<endl;
        cout<<"Payment = "<<P<<" Rs."<<endl;
        }
};
```

```
main()
{
Worker a, b;
a.setdata("JACK", 100, 10);
b.setdata("JAMES", 200, 20);

a.payment();
// Find payment by using data of object-a

b.payment();
// Find payment by using data of object-b
}
```

Output:

```
Worker Name = JACK
Payment = 1000 Rs.

Worker Name = JAMES
Payment = 4000 Rs.
```

Example: Design a class Student with data members Roll No., Name and Department. Define member func
setdata() & showdata().

```
#include<string.h>
#include<iostream>
using namespace std;
class Student
{
        private:
                int RN;
                char SN[20], D[20];
        public:
                void setdata(int rn, char n[], char d[])
                {
                        RN = rn;
                        strcpy(SN, n);
                        strcpy(D, d);
                }

                void showdata()
                {
                        cout<<"\nRoll Number = "<<RN<<endl;
                        cout<<"Student Name = "<<SN<<endl;
                        cout<<"Department = "<<D<<endl;
                }
};
```

```
main()
{
Student a, b;
a.setdata(1111, "JACK", "CSE");
a.showdata();

b.setdata(2222, "JAMES", "IT");
b.showdata();
}
```

Output:

```
Roll Number = 1111
Student Name = JACK
Department = CSE

Roll Number = 2222
Student Name = JAMES
Department = IT
```

Example: Design a class Currency with data-members Rupees and Paise. Define member functions setdata
showdata() and convert(). The convert() function will return data of a Currency object into paise. (1 Rupee = 100 Pai

FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE

```
#include<iostream>
using namespace std;
class Currency
{
        private:
                int R, P;
        public:
        void setdata(int r, int p)
        {
                R = r;
                P = p;
        }

        void showdata()
        {
        cout<<R<<" Rupees "<<P<<" Paise\n";
        }

        int convert()
        {
                return(R*100 + P);
        }};
```

```
main()
{
Currency a;
a.setdata(10, 50);
a.showdata();
int x = a.convert();
cout<<"Data of obj-a is "<<x<<" Paise"<<endl;
}
```

Output:

```
10 Rupees 50 Paise
Data of obj-a is 1050 Paise
```

Example: Design a class Date having data-members day, month & year. Define member function setdata() & showdata()

```
#include<iostream>
using namespace std;
class Date
{
        private:
                int D, M, Y;
        public:
        void setdata(int d, int m, int y)
        {
                D = d;
                M = m;
                Y = y;
        }

        void showdata()
        {
        cout<<D<<"."<<M<<"."<<Y<<endl;
        }
};
```

```
main()
{
        Date a;
        a.setdata(15, 8, 1947);
        a.showdata();

        Date b;
        b.setdata(26, 1, 1950);
        b.showdata();
}
```

Output:

```
15.8.1947
26.1.1950
```

Example: Design a class Book with data-members book name, author name & price. Define member functions setdata() & showdata().

```
#include<string.h>
#include<iostream>
using namespace std;
class Book
{
        private:
                char BN[20], AN[20];
                float P;
        public:
        void setdata(char bn[], char an[], float p)
```

```
main()
{
        Book a;
        a.setdata("C++", "JACK", 199.99);
        a.showdata();
}
```

Output:

```
Book Name = C++
Author Name = JACK
```

```
        Price = 199.99 Rs.
    {
        strcpy(BN, bn);
        strcpy(AN, an);
        P = p;
    }

    void showdata()
    {
    cout<<"\nBook Name = "<<BN<<endl;
    cout<<"Author Name = "<<AN<<endl;
    cout<<"Price = "<<P<<" Rs."<<endl;
    }
};
```

Member Function Overloading

- The process of defining multiple member functions in a class with same name but different number of argume or different data-types of arguments is called as member function overloading.
- An appropriate function will be call as per the values passed while calling the function.

Example: Design a class Circle with data member Radius. Define member-functions setradius(), area() a circumference(). Overload the meber-function setradius() so that it can be call by passing int and double value.

```
#include<iostream>
using namespace std;
class Circle
{
    private:
        double R;
    public:
    void setradius(int r)
    {
        R = r;
    }

    void setradius(double r)
    {
        R = r;
    }

    void area()
    {
        double A = 3.14 * R * R;
        cout<<"Area of circle is "<<A<<endl;
    }

    void circumference()
    {
        double C = 2 * 3.14 * R;
        cout<<"Circumference is "<<C<<endl;
    }
};
```

```
main()
{
    Circle a;
    a.setradius(5);
    a.area();
    a.circumference();

    Circle b;
    b.setradius(6.5);
    b.area();
    b.circumference();
}
```

Output:

```
Area of circle is 78.5
Circumference is 31.4
Area of circle is 132.665
Circumference is 40.82
```

Example: Design a class Distance with data-members Feet & Inches. Define member-functions void setdata(int, int), void setdata(int), void showdata() & int convert().


```
#include<iostream>
using namespace std;
class Distance
{
    private: int F, I;
    public:
    void setdata(int f, int i)
    {
        F = f; I = i;
    }

    void setdata(int f)
    {
        F = f; I = 0;
    }

    void showdata()
    {
    cout<<F<<" Feet "<<I<<" Inches"<<endl;
    }

    int convert()
    {
        return(F * 12 + I);
    }};
```

```
main()
{
Distance a;
a.setdata(5, 2);
a.showdata();
int x = a.convert();
cout<<"Data of obj-a is "<<x<<" inches"<<endl;

Distance b;
b.setdata(5);
b.showdata();
int y = b.convert();
cout<<"Data of obj-b is "<<y<<" inches"<<endl;
}
```

Output:

```
5 Feet 2 Inches
Data of object a is 62 inches

5 Feet 0 Inches
Data of object b is 60 inches
```

Constructors

- Constructors are used to initialize an object.
- That means to store values into an object as soon as the object is created.
- Constructors are automatically called for each object as soon as the object is created.
- Constructors have same name as the class name.
- Constructor does not have any return_type and it cannot return any value.
- If the return type is provided to the constructor then it will cause a compile time error.
- Constructor can have zero or more arguments.
- The constructor with no argument is called as default constructor
- The constructor having one or more argument is called as parameterizd constructor
- It is possible to overload constructor
- It is possible to specify default arguments to the constructor.
- Constructors never inherit in derived class
- The constructor in which space is dynamically allocated for the data-members such a constructor is called as dynamic constructor.

Note:

1. The following functions are always automatically called by C++.
   i) Constructor ii) Destructor iii) Copy Constructor iv) Operator functions in operator overloading.
   v) Operator conversion functions in operator overloading type casting. vi) Assignment Operator Function.
2. The **copy constructor**, **overloaded assignment operator function** and **this** pointer is automatically added in the compile code of the class when the program is compiled.

Example: Design a class Distance having datamembers Feet and Inches i.e. F & I and member functions showdata() whose objects should initializes to zero.

```
#include<iostream>
using namespace std;
class Distance
{
        private:
                int F, I;
        public:
                Distance()
                {
                        F = 0;
                        I = 0;
                }
                void showdata()
                {
                cout<<F<<" Feet "<<I<<" Inches"<<endl;
                }
};

main()
{
        Distance a;
        a.showdata();
        Distance b;
        b.showdata();
}
```

Output:
0 Feet 0 Inches
0 Feet 0 Inches

Parameterized Constructors

- The default constructor will provide fixed/same values for the all the objects.
- While creating the object we can pass one or more values.
- These values are passed as argument to the constructor and the constructors uses these values to initialize the object.

Example: Design a class Distance with data members Feet and Inches. Define member function showdata(), convert(). The convert() function will return the data of the Distance object into inches. The objects of the class must be created by passing data. (1 Feet = 12 Inches).

```
#include<iostream>
using namespace std;
class Distance
{
        private:
                int F, I;
        public:
        Distance(int f, int i)
        {
                F = f;
                I = i;
        }

        void showdata()
        {
        cout<<F<<" Feet "<<I<<" Inches"<<endl;
        }

        int convert()
        {
                return(F*12 + I);
        }
};

main()
{
Distance a(5, 2);
a.showdata();
int x = a.convert();
cout<<"Data of obj-a is "<<x<<" inches"<<endl;
}
```

Output:
5 Feet 2 Inches
Data of obj-a is 62 inches

Example: Design a class Student with data members Roll Number, Name and Department. Define member function showdata(). The objects of the class must be created by passing data.


```
#include<string.h>
#include<iostream>
using namespace std;
class Student
{       private: int RN; char N[20], D[20];
        public:
        Student(int rn, char n[], char d[])
        {
                RN = rn;
                strcpy(N, n);
                strcpy(D, d);
        }
        void showdata()
        {
        cout<<"\nRoll Number = "<<RN<<endl;
        cout<<"Student Name = "<<N<<endl;
        cout<<"Department = "<<D<<endl;
        }
};

main()
{
Student a(1010, "JACK", "CSE");
a.showdata();

Student b(1020, "JAMES", "IT");
b.showdata();
}
```

Output:
Roll Number = 1010
Student Name = JACK
Department = CSE

Roll Number = 1020
Student Name = JAMES
Department = IT

Constructor Overloading

- Defining multiple constructors in the same class with different number of argument or different data-types of argument is called as constructor overloading.
- That means each constructor must have unique signature.
- An appropriate constructor will automatically invoke as per the arguments passed while creating the object.

Example: Design a class Distance with data member function Feet and Inches. Define member function showdata(). The objects of the class must be created by passing 2, 1 or 0 arguments.

```
#include<iostream>
using namespace std;
class Distance
{
        private: int F, I;
        public:
                Distance(int f, int i)
                {
                        F = f;
                        I = i;
                }

                Distance(int f)
                {
                        F = f;
                        I = 0;
                }

                Distance()
                {
                        F = I = 0;
                }

                void showdata()
                {
                cout<<F<<" Feet "<<I<<" Inches"<<endl;
                }
};

main()
{
        Distance a(5, 2);
        a.showdata();

        Distance b(4);
        b.showdata();

        Distance c;
        c.showdata();
}
```

Output:
5 Feet 2 Inches
4 Feet 0 Inches
0 Feet 0 Inches

Constructor with default (i.e. optional) arguments

- It is possible to specify default value to the arguments of a constructor.
- Specifying default value makes that argument optional.
- The default values must be specified in right to left direction.
- The benefit is that, in some cases it avoids the need to overload the constructor.

Note:

- If a class contains multiple constructors and we also want to specify default arguments to the constructor then default values must be specified to the constructor which have minimum number of arguments.

Example: Design a class Distance with data-members Feet and Inches. Define member-function showdata() objects of the class must be created by passing 2, 1 or 0 arguments.

```
#include<iostream>
using namespace std;
class Distance
{
        private:
                int F, I;
        public:
                Distance(int f=0, int i=0)
                {
                        F = f;
                        I = i;
                }

                void showdata()
                {
                cout<<F<<" Feet "<<I<<" Inches"<<endl;
                }
};
```

```
main()
{
        Distance a(5, 2);
        a.showdata();

        Distance b(4);
        b.showdata();

        Distance c;
        c.showdata();
}
```

Output:

```
5 Feet 2 Inches
4 Feet 0 Inches
0 Feet 0 Inches
```

Reference Variable

- A reference variable is used to provide an alternate name (i.e. alias) for an existing variable or object.
- **Syntax:** `data_type &ref_name = existing_variable;`

Example:

```
#include<stdio.h>
#include<iostream>
using namespace std;
main()
{
        int a = 10;
        int &b = a;
        int &c = b;
        cout<<"Address of a = "<<&a<<endl;
        cout<<"Address of b = "<<&b<<endl;
        cout<<"Address of c = "<<&c<<endl;
}
```

Output:

```
Address of a = 0x6ffe3c
Address of b = 0x6ffe3c
Address of c = 0x6ffe3c
```

Example:

```
#include<stdio.h>
#include<iostream>
using namespace std;
main()
{
        int a = 10;
        int &b = a;
        int &c = b;
        cout<<a<<"\t"<<b<<"\t"<<c<<endl;
        a--;
        b--;
        c--;
        cout<<a<<"\t"<<b<<"\t"<<c<<endl;
}
```

Output:

```
10    10    10
7     7     7
```


FOR EDUCATIONAL PURPOSE ONLY, NOT FOR SALE

Copy Constructor

- Copy constructor is used to initialize an object by using data of another object of same class.
- The copy constructor takes reference of an object as argument.
- A copy constructor is automatically called when we pass an object while creating another object of same class or if we assign an object while creating another object of same class.
- Note that the copy constructor, assignment operator function and ***this*** pointer is automatically added in the compile code of class when the program is compiled.
- **Syntax:**

```
class_name(class_name &ref_obj)
{
        ....
        ....
}
```

- **Example:**

```
#include<iostream>
using namespace std;
class Distance
{
        private:
                int F, I;
        public:
        Distance(int f = 0, int i = 0)
        {
                F = f;
                I = i;
        }

        Distance(Distance &z) //Copy constructor
        {
        cout<<"\nThis is a copy constructor"<<endl;
        F = z.F;
        I = z.I;
        }

        void showdata()
        {
        cout<<F<<" Feets "<<I<<" Inches"<<endl;
        }
};
```

```
main()
{
Distance a(5, 3);
a.showdata();

Distance b(a);
b.showdata();

Distance c = b;
c.showdata();

Distance d;
d = c;
/*Overloaded assignment operator function will invoke.*/

d.showdata();
}
```

Output:

```
5 Feets 3 Inches

This is a copy constructor
5 Feets 3 Inches

This is a copy constructor
5 Feets 3 Inches

5 Feets 3 Inches
```

Scope resolution operator (::)

- The scope resolution operator is used for following purposes:

1. To access the global variable when local and global variables have same name.
2. To define the member functions outside the class.
3. To assign value to static data member
4. To use public static-data_memebrs and call static member functions
5. To call base class overrided functions in derived class during the process of inheritance.

Example:

```
#include<iostream>
using namespace std;
int a = 10;        // Global variable
main()
{
        int a = 5;          // Local variable
        cout<<"Local a = "<<a<<endl;
        cout<<"Global a = "<<::a<<endl;
}
```

Output:

```
Local a = 5
Global a = 10
```

Defining members function outside the class

- To increase readability of the program we can define the member functions, constructors, etc. of a class o
  the class.
- Readability of program: The program which is easy to read and understand has the higher readability.
- If we want to define the member function outside the class then we must define the prototype of the me
  function inside the class.
- If the constructor or member-function contains default arguments then the default values can be specified a
  prototype or in the definition of the constructor or member-function.
- The member function can be define outside the class by using following syntax.
- **Syntax:**

```
return_type class_name :: function_name(data_type arg1, data_type arg2, ...)
{
        ....
        ....
        return (value);
}
```

Example: Design a class Distance with data members Feet and Inches. Define member functions showdata(), conve
The convert() will return the data of Distance object into inches (1 Feet = 12 Inches). The objects of the class mus
created by passing 2, 1 or 0 arguments. Define all the constructors and member functions outside the class.

```
#include<iostream>
using namespace std;
class Distance
{
        private:
                int F, I;
        public:
                Distance(int=0, int=0);
                void showdata();
                int convert();
};

Distance :: Distance(int f, int i)
{
        F = f;
        I = i;
}

void Distance :: showdata()
{
        cout<<F<<" Feet "<<I<<" Inches"<<endl;
}

int Distance :: convert()
{
        return(F*12 + I);
}
```

```
main()
{
Distance a(5, 2);
a.showdata();
int x = a.convert();
cout<<"Data of obj-a is "<<x<<" inches"<<endl;

Distance b(4);
b.showdata();
int y = b.convert();
cout<<"Data of obj-b is "<<y<<" inches"<<endl;

Distance c;
c.showdata();
int z = c.convert();
cout<<"Data of obj-c is "<<z<<" inches"<<endl;
}
```

Output:

```
5 Feet 2 Inches
Data of obj-a is 62 inches
4 Feet 0 Inches
Data of obj-b is 48 inches
0 Feet 0 Inches
Data of obj-c is 0 inches
```

Inline function

- Whenever a C++ program is compiled, the compiler produces a unique address for the compile code of ea
  function.
- The Regular Functions are called by jumping on the function address, execute the function code and come bac
  the calling function.
- This process of calling the function by jumping on the function address takes some time and due to this t
  program takes more time for its execution.

- An Inline function behaves like macros in C.
- Inline function are the function for which the complier replaces the function call with the corresponding function's compile code and due to this there is no need to call the function by jumping on the function address and the program takes less time for execution.
- It is beneficial that inline function makes our program faster but the program compile code may take more memory for storage as the function's compile code is inserted at each point where the function is called.
- We can define a function as inline by using the keyword **inline** as a prefix to the function prototype.
- Declaring a function as inline means requesting the compiler.
- Now, it depends on the complier to treat the function as inline or not.
- If the function is small and not so complicated then the function will be treated as inline otherwise it will treated like our regular function.
- If the function is defined inside the class then such a function is automatically treated as inline (if it is small and not complicated).
- **In following situation the compiler ignore the inline declaration.**

1. If the function is returning value and in such function the use of a loop, goto or a switch occurs.
2. If the function have returntype ***void*** and such a function contain ***return*** statement in it.
3. If the function is recursive.
4. If there is a use of a static variable inside the function.

Example: Write a program to illustrate the inline functions.

```
#include<iostream>
using namespace std;
class Distance
{
        private:
                int F, I;
        public:
        void setdata(int f, int i)
        {
                F = f;
                I = i;
        }
        inline void showdata();
        void convert();
};

void Distance :: showdata()
{
cout<<F<<" Feet "<<I<<" inches"<<endl;
}

void Distance :: convert()
{
float z = F * 12 + I;
cout<<"Data of Distance object is "<<z<<" inches"<<endl;
}
```

```
main()
{
Distance a;
a.setdata(2, 4);
/*It will treated as inline because it is defined inside a class*/

a.showdata();
/*It will treated as inline because it is declared as inline*/

a.convert();
/*It will be call by jumping on the function address because it is defined outside the class and not declared as inline.*/
}
```

Output:

```
2 Feet 4 inches
Data of Distance object is 28 inches
```

Static Members

- static Data Member
- static Member Function

1. static Data Member

- **The data members whose value is common (i.e same) for all the objects of that class, such a data member can be declared as static.**
- If a data member is declared as static then for such data member space is not created inside the object.

- For static data member space is created outside the object only one time and all the objects of that class ca that data member.
- The static data members will not destroy until the program terminates.
- For static data member the initial value can be assign after the declaration of the class as follows.
- **Syntax:**

```
data_type class_name :: member_name = value;
```

Note:
- If no initialization is specified then numeric static data members are initializes to zero
- If no initialization is specified then char static data members are initializes to blank
- It is not possible to define a static array
- The static data members can be use inside the non-static member function, static member function, constr and in destructor.

2. static Member Function

- The static member function can be call with the class name.
- That means to call a static member function we don't need to create object of that class.
- In the static member function we can use only other static data-members & static member-function of the sa class.
- But, in the static member function we can use static as well as non-static members of other classes.
- It is NOT POSIIBLE to override the static member function in derived class.
- A static member function can be call by using following syntax.
- **Syntax:**

```
class_name :: function_name(val1,val2, ....);
```

Note:
- static members are also called as class members.
- Non-static members are also called as object members or instance members.
- We can use the static data-members and static member functions inside the the non-static member functio the same class.

Example:

```
#include<iostream>
using namespace std;
class Set
{
        private:
                int x, y;
                static float z;
};
float Set :: z = 0;

main()
{
        Set a, b;
        cout<<"Sizeof object a = "<<sizeof(a)<<" bytes"<<endl;
        cout<<"Sizeof object b = "<<sizeof(b)<<" bytes"<<endl;
}
```

Output:

```
Size of object a = 4 bytes
Size of object b = 4 bytes
```

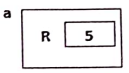
Example: Design a class Circle with data-member R and member functions area() & circumference(). Also define static data-member PI = 3.14 and a static member-function showpi() in the class. The objects of the class must b created by passing data.

```
#include<iostream>
using namespace std;
class Circle
{
        private:
                int R;
                static float PI;
        public:
                Circle(int r)
                {
                        R = r;
                }
                void area()
                {
                        float A = PI * R * R;
                        cout<<"Area of circle is "<<A<<endl;
                }
                void circumference()
                {
                        float C = 2 * PI * R;
                        cout<<"Circumference is "<<C<<endl;
                }
                static void showpi()
                {
                cout<<"PI = "<<PI<<endl;
                }
};
float Circle :: PI = 3.14;

main()
{
        Circle a(5);
        a.area();
        a.circumference();
        Circle::showpi();
}
```

Output:

```
Area of circle is 78.5
Circumference is 31.4
PI = 3.14
```


FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE



Object a

PI 3.14

Space for PI will be created outside the object only one time as PI is declared as static in the class

Friend Function | Object as argument

- We know that, we cannot use the private and protected members of a class outside the class derectly.
- But, if we want to use the private and protected members outside the class then we can use the friend function.
- The friend function is an out-sider function, that means it is not a member function of a class.
- The friend function is mainly used if we want to perform operation by using private or protected members of two or more different classes
- The function can be declare as friend of a class by defining the prototype of the function inside the class preceded with the keyword friend.
- The prototype of function can be write in the private or protected or public section of the class.

Example:

```
#include<iostream>
using namespace std;
class Alpha
{
        private: int A;
                friend int main();
        protected:
                int B;
        public:
                int C;
        public:
                Alpha()
                {
                        A = 0;
                        B = 0;
                        C = 0;
                }
};

main()
{
        Alpha a;
        cout<<"a.A = "<<a.A<<endl;
        cout<<"a.B = "<<a.B<<endl;
        cout<<"a.C = "<<a.C<<endl;
}
```

Output:

```
a.A = 0
a.B = 0
a.C = 0
```

Example: WAP to exchange values of two variables by using friend function. (Call By Reference)

```
#include<iostream>
using namespace std;
class Alpha
{
        private:
        int X, Y;
        friend void exchange(Alpha*);
        public:
        Alpha (int x, int y)
        {
                X = x;
                Y = y;
        }

        void showdata()
        {
                cout<<"X = "<<X<<endl;
                cout<<"Y = "<<Y<<endl;
        }
};
void exchange(Alpha *p)
{
        /*
        int t = p->X;
        p->X = p->Y;
        p->Y = t;
        */
        int t = (*p).X;
        (*p).X = (*p).Y;
        (*p).Y = t;
}
main()
{
        Alpha a(5, 7);
        cout<<"Before Exchange: "<<endl;
        a.showdata();
        exchange(&a);
        cout<<"After Exchange: "<<endl;
        a.showdata();
}
```

Output:

```
Before Exchange:
X = 5
Y = 7
After Exchange:
X = 7
Y = 5
```

Example: WAP to exchange values of two variables by using friend function. (Pass By Reference)

```
#include<iostream>
using namespace std;
class Alpha
{
        private:
        int X, Y;
        friend void exchange(Alpha &);
        public:
        Alpha(int x, int y)
        {
                X = x;
                Y = y;
        }
        void showdata()
        {
                cout<<"X = "<<X<<endl;
                cout<<"Y = "<<Y<<endl;
        }
};
void exchange(Alpha &b)
{
int t = b.X;
b.X = b.Y;
b.Y = t;
}

main()
{
Alpha a(5, 7);
cout<<"Before Exchange: "<<endl;
a.showdata();
exchange(a);
cout<<"After Exchange: "<<endl;
a.showdata();
}
```

Output:

```
Before Exchange:
X = 5
Y = 7
After Exchange:
X = 7
Y = 5
```

Example: WAP to add two numbers by using friend function.

```
#include<iostream>
using namespace std;
class Alpha
{
        private:
                int N;
                friend int sum(Alpha, Alpha);
        public:
                Alpha(int n)
                {
                        N = n;
                }
                void showdata()
                {
                        cout<<"N = "<<N<<endl;
                }
};
int sum(Alpha x, Alpha y)
{
        return(x.N + y.N);
}

main()
{
        Alpha a(5), b(7);
        a.showdata();
        b.showdata();
        int s = sum(a, b);
        cout<<"Sum is "<<s<<endl;
}
```

Output:

```
N = 5
N = 7
Sum is 12
```


FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE

friend Classes

- If a class need to access private or protected members of another class then in such condition we can declare the class as the friend of the another class by declaring the class prototype in the another class preceded with the keyword ***friend***.
- The friend class can use private or protected members of another class.

Example:

```
#include<iostream>
using namespace std;
class Set
{
        private:
                int X, Y;
        public:
        Set(int p, int q)
        {
                X = p;
                Y = q;
        }

        void showdata()
        {
        cout<<"\nX = "<<X<<endl;
        cout<<"Y = "<<Y<<endl;
        }

        friend class Maths;
};
class Maths
{
        public:
        static int max(Set z)
        {
                if(z.X > z.Y)
                return(z.X);
                else
                return(z.Y);
        }

        static int min(Set z)
        {
                if(z.X < z.Y)
                return(z.X);
                else
                return(z.Y);
        }
};
main()
{
Set a(10, 20);
a.showdata();

int g = Maths :: max(a);
cout<<"Greatest is "<<g<<endl;

int s = Maths :: min(a);
cout<<"Smallest is "<<s<<endl;
}
```

Output:

```
X = 10
Y = 20
Greatest is 20
Smallest is 10
```

C++ च्या compiler ला हे माहिती असते कि ++, --, +, -, <, > etc operators चा उ
); built in datatype (int, float etc) साठी काम करायचा. पण C++ च्या compiler ला हे माहिती. म्हणि
operators obj साठी काम करायचे. Operator Overloading म्हणजे compiler ला

Operator Overloading म्हणजे काय op चा use obj साठी करता येतो

- The compiler know how to use operators with built-in (i.e. pre-defined or readymade) data-types.
- But, the compiler didn't know how to use operators with userdefined data-types i.e. objects.
- Operator overloading is the process to inform compiler about how to use an operator with the objects.
- The operators are divided into two categories as follows:

Unary Operators	▪ The operators which can be use with a single variable/operand are called as unary operators. ▪ Example: ++, --, +, -, !, etc.
Binary Operators	▪ The operators which need at least two operands to operate such operators are called as binary operators. ▪ Example: +, -, <, >, ==, +=, -=, etc.

The following operators cannot be overloaded:

Sr. No.	Operators	Description
1	.	Dot Operator
2	::	Scope Resolution Operator
3	?:	Ternary Operator

Sr. No.	Operators	Description
5	sizeof	Size of operator
6	:	Colon
7	.*	

Note: operator + (p1, p2) is equivalent to p1 + p2 for friend functions.

Rules for operator overloading

1. We must not define new operators. We can overload only existing operators.
2. The operators must be used with an object.
3. It is not permitted to change the original meaning of the defined operator i.e. ++ should not perform -- operation
4. In binary operator overloading if the operator function is defined inside the class then the first operand must be an object.

Overloading unary Operators

- To overload unary operators the following general syntax can be used.
- The operator function will be automatically call for the object with the operator is used.

- **General Syntax:**

```
returntype operator op()
{
        ....
        ....
        return(value);
}
```

Overloading prefix and postfix operators

- To overload prefix and postfix ++ and -- operators C++ provides us two separate syntaxes.

Prefix (e.g. ++a, --a) operator function syntax:

```
return_type operator op()
{
        ....
        ....
        return(value);
}
```

Postfix (e.g. a++, a--) operator function syntax:

```
return_type operator op(int)
{
        ....
        ....
        return(value);
}
```

Example: Design a class Distance with data members feet & inches and member function showdata() whose objects can be created by passing data. Overload ++ operator as prefix and postfix which will increment the data of Distance object.

```
#include<iostream>
using namespace std;
class Distance
{       private: int F, I;
        public:
                Distance(int f, int i)
                {
                        F = f;
                        I = i;
                }

                void showdata()
                {
                cout<<F<<" Feet "<<I<<" Inches"<<endl;
                }

                void operator ++ ()     // Prefix
                {
                        ++F;
                        ++I;
                        if(I >= 12)
                        {
                                ++F;
                                I = I - 12;
                        }
                }
                void operator ++ (int)  // Postfix
                {
                        F++;
                        I++;

                        if(I >= 12)
                        {
                                F++;
                                I = I - 12;
                        }
                }
};

main()
{
        Distance a(3, 11);
        ++a;
        a.showdata();

        Distance b(7, 9);
        b++;
        b.showdata();
}
```

Output:

```
5 Feet 0 Inches
8 Feet 10 Inches
```


FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE

Example: Design a class MyString with a data member to store string. Define member function showdata(). The objects can be created by passing data. Overload ~ (i.e. tilde) operator which will reverse the data of MyString object.

```
#include<string.h>
#include<iostream>
using namespace std;
class MyString
{
        private:
                char N[50];
        public:
                MyString(char n[])
                {
                        strcpy(N, n);
                }
                void showdata()
                {
                        cout<<"String = "<<N<<endl;
                }
                void operator ~ ()
                {
                        strrev(N);
                }
};

main()
{
        MyString a("ABCD");
        a.showdata();
        ~a;
        a.showdata();
}
```

Output:

```
String = ABCD
String = DCBA
```

Example: Design a class Time with data members hours & minutes. Define member function showdata(). The objects can be created by passing 2 or 0 arguments. Overload ++ operator as prefix which will increase data of Time object.

```
#include<iostream>
using namespace std;
class Time
{
        private:
                int H, M;
        public:
                Time(int h = 0, int m = 0)
                {
                        H = h;
                        M = m;
                }
                void showdata()
                {
                        cout<<H<<" Hours "<<M<<" Minutes"<<endl;
                }
                Time operator ++ ()
                {
                        Time z;
                        z.H = ++H;
                        z.M = ++M;
                        return(z);
                }
};

main()
{
        Time a(3, 10), b;
        a.showdata();
        b.showdata();
        b = ++a;
        b.showdata();
}
```

Output:

```
3 Hours 10 Minutes
0 Hours 0 Minutes
4 Hours 11 Minutes
```

Overloading Binary Operators

- To overload binary operator the class must contain a special operator function as follows.
- **Syntax:**

```
returntype operator op(datatype arg)
{
    ---
    ---
    return(value);
}
```

Note:

- This binary operator function accepts only one argument.
- To overload binary operators the first operand must be an object.

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

a → 1st operand (Must be an object)
b → 2nd operand (Can be anything)

- This operator function will automatically invoke for the object (1st operand) and the second operand
- transferred as argument to the operator function.
- If the operator function is returning an object then create a temporary object in the operator function, store result in the temporary object and return the temporary object.

Example: Design a class Distance with datamembers Feets and Inches. Define member function showdata() objects of the class can be created by passing data. Overload > (greater than) operator which will compare data of distance object and return true (1) or false (0).

```
#include<iostream>
using namespace std;
class Distance
{
    private:
        int F, I;
    public:
        Distance(int f, int i)
        {
            F = f
            I = i;
        }

        void showdata()
        {
        cout<<F<<" Feet "<<I<<" Inches"<<endl;
        }

        int operator > (Distance z)
        {
            int p = F*12 + I;
            int q = z.F*12 + z.I;

            if(p > q)
            return(1);
            else
            return(0);
        }
};
main()
{
Distance a(3, 2), b(2, 4);
a.showdata();
b.showdata();

if(a > b)
cout<<"Data of obj- a is greatest"<<endl;
else
cou"<<"ata of obj-b is greatest"<<endl;
}
```

Output:

```
3 Feet 2 Inches
2 Feet 4 Inches
Data of obj-a is greatest
```

Example: Design a class Time with data-members hours, minutes and seconds. Define member function showdata The objects can be created by passing data. Overload + operator which will add two Time objects.

FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE

```
#include<iostream>
using namespace std;
class Time
{
    private:
        int H, M, S;
    public:
    Time(int h = 0, int m = 0, int s = 0)
    {
        H = h;
        M = m;
        S = s;
    }

    void showdata()
    {
    cout<<H<<" Hrs. "<<M<<" Mins. "<<S<<" Sec."<<endl;
    }

    Time operator + (Time z)
    {
        Time t;
        t.H = H + z.H;
        t.M = M + z.M;
        t.S = S + z.S;
        if(t.S >= 60)
        {
            t.S -= 60;
            t.M++;
        }

        if(t.M >= 60)
        {
            t.M -= 60;
            t.H++;
        }
        return (t);
    }
};
main()
{
    Time t1(2, 39, 59), t2(2, 40, 54), t3;
    t3.showdata();
    t3 = t1 + t2;
    t3.showdata();
}
```

Output:

```
0 Hrs. 0 Mins. 0 Secs.
5 Hrs. 20 Mins. 53 Secs.
```

Example: Design a class MyString with data member to store string. Define member function showdata(). The objects of the class can be created by passing data. Overload + operator which will concat (join) data of two MyString objects.

```
#include<string.h>
#include<iostream.>
using namespace std;
class MyString
{   private: char N[50];
    public:
        MyString(char n[] = "")
        {
            strcpy(N, n);
        }

        void showdata()
        {
            cout<<"String = "<<N<<endl;
        }

        MyString operator + (MyString z)
        {
            MyString t;
            strcpy(t.N, N);
            strcat(t.N, z.N);
            return(t);
        }
};
main()
{
    MyString a("Raj"), b("kamal");
    MyString c = a + b;
    c.showdata();
}
```

Output:

```
String = Rajkamal
```

Example: Write a program to overload == operator to compare two MyString objects.

```
#include<string.h>
#include<iostream>
using namespace std;
class MyString
{
        private:
                char N[50];
        public:
                MyString(char n[])
                {
                        strcpy(N, n);
                }

                void showdata()
                {
                        cout<<"String = "<<N<<endl;
                }

                int operator == (MyString z)
                {
                        if(strcmp(N, z.N) == 0)
                        return(1);
                        else
                        return(0);
                }
};
```

```
main()
{
        MyString a("JACK"), b("jack");
        if(a == b)
        cout<<"Same"<<endl;
        else
        cout<<"Not Same"<<endl;
}
```

Output:
Not Same

Example: Design a class Distance with data members Feet & Inches. Define member function showdata(). The objects of the class can be created by passing data. Overload += operator which will add n feet and n inches in the Distance object.

```
#include<iostream>
using namespace std;
class Distance
{
        private:
                int F, I;
        public:
                Distance(int f, int i)
                {
                        F = f;
                        I = i;
                }

                void showdata()
                {
                cout<<F<<" Feet "<<I<<
                        " Inches"<<endl;
                }

                Distance operator += (int n)
                {
                        F += n;
                        I += n;

                        if(I >= 12)
                        {
                                F++;
                                I = I - 12;
                        }

                        return(*this);
                }
};
```

```
main()
{
        Distance a(2, 8);
        a.showdata();
        a += 7;
        a.showdata();
}
```

Output:
2 Feet 8 Inches
10 Feet 3 Inches

OR

```
void operator += (int n)
{
        F += n;
        I += n;

        if(I >= 12)
        {
                F++;
                I = I - 12;
        }
}
```


FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE

Example: Write a program to overload += operator to add two complex numbers

```
#include<iostream>
using namespace std;
class Complex
{
        private:
                float real, imag;
        public:
                Complex(float rp, float ip)
                {
                        real = rp;
                        imag = ip;
                }

                void showdata()
                {
                        cout<<" \n "<<real<<" + "<<imag<<"i"<<endl;
                        cout<<"Real Part = "<<real<<endl;
                        cout<<"Imag Part = "<<imag<<endl;
                }

                Complex operator += (Complex z)
                {
                        this->real += z.real;
                        this->imag += z.imag;
                        return(*this);
                }
};
```

```
main()
{
Complex a(1.1, 2.2), b(3.3, 4.4);
a.showdata();
b.showdata();
a+=b;
a.showdata();
}
```

Output:

```
1.1 + 2.2i
Real Part = 1.1
Imag Part = 2.2

3.3 + 4.4i
Real Part = 3.3
Imag Part = 4.4

4.4 + 6.6i
Real Part = 4.4
Imag Part = 6.6
```

Example: WAP to overload subscript(i.e. [ ]) operator

```
#include<iostream>
using namespace std;
const int SIZE = 10;
class MyArray
{
        private:
                int arr[SIZE];
        public:
        MyArray()
        {
                for(int i = 0; i < SIZE; i++)
                arr[i] = i;
        }

        int operator [] (int i)
        {
                if(i >= SIZE)
                {
```

```
main()
{
MyArray a;
cout<< "Value of a[2] = "<<a[2]<<endl;
cout<< "Value of a[5] = "<<a[5]<<endl;
cout<< "Value of a[10] = "<<a[10]<<endl;
}
```

Output:

```
Value of a[2] = 2
Value of a[5] = 5
Error: Invalid Array index
Value of a[10] = 0
```

```
        cout <<"Error: Invalid Array index\n";
        return( -1);
        }
        else
        return arr[i];
    }
};
```

Binary operator overloading by using friend function

- To overload binary operators by using friend function, the operator function will accept only two arguments.
- Here the first operand will come as first argument and second operand will come as second argument.
- It is not necessary that the first operand should be an object.

Example: Design a class Time with data members hours & minutes. Define member function showdata(). The objects of the class can be created by passing 2 or 0 arguments. overload + operator by using friend function which will add hours with the Time object.

```
#include<iostream>
using namespace std;
class Time
{
    private: int H, M;
    public:
        Time(int h = 0, int m = 0)
        {
            H = h;
            M = m;
        }

        void showdata()
        {
        cout<<H<<" Hours "<<M<<" Minutes"<<endl;
        }

        friend Time operator + (Time, int);
        friend Time operator + (int, Time);
};

Time operator + (Time p, int n)
{
    Time z;
    z.H = p.H + n;
    z.M = p.M;
    return(z);
}

Time operator + (int n, Time p)
{
    Time z;
    z.H = p.H + n;
    z.M = p.M;
    return(z);
}

main()
{
Time a( , 30 ), b;
b = a + 3;
// Implicit function call
b.showdata();

Time c(4, 10), d;
d = 3 + c;
// Implicit function call
d.showdata();
}
```

Output:
5 Hours 30 Minutes
7 Hours 10 Minutes

// Explicit function
b = operator + (a,
d = operator + (3,

Example: WAP to overload insertion (<<) and extraction (>>) operator

```
#include<iostream>
using namespace std;
class Distance
{
    private: int F, I;
    public:
    friend void operator >>(istream&, Distance&);
    friend void operator <<(ostream&, Distance&);
};

main()
{
Distance D;
cout<<"Enter Feets and Inches: ";
cin>>D;
cout<<D;
}
```

Output:
Enter Feets and Inches: 5 2
5 Feet 2 Inches


```
void operator >>(istream &in, Distance &d)
{
    in>>d.F>>d.I;
}

void operator <<(ostream &out, Distance &d)
{
    out<<d.F<<" Feet "<<d.I<<" Inches"<<endl;
}
```

Unary operator overloading by using friend function

- It is possible to overload unary operators by using friend function.
- For this purpose the operator function must be define by using following syntax.

Syntax prefix (e.g. ++a, --a):

```
return_type operator op (class_name &ref_name)
{
    ...
    ...
    return(value);
}
```

Syntax postfix (e.g. a++, a--):

```
return_type operator op (class_name &ref_name, int)
{
    ...
    ...
    return(value);
}
```

Note:

1. The above operator function will accept only one argument.
2. The object with the operator is used will go as argument to the operator function.

Example: Design a class Time with data-members hours & minutes. Define member function showdata(). The objects can be created by passing 2 or 0 arguments. Overload ++ operator by using friend function which will increase only hours in the Time object.

```
#include<iostream>
using namespace std;
class Time
{
    private:
        int H, M;
    public:
        Time(int h = 0, int m = 0)
        {
            H = h;
            M = m;
        }

        void showdata()
        {
        cout<<H<<" Hours "<<M<<" Minutes"<<endl;
        }

        friend Time operator ++ (Time);
};

Time operator ++(Time z)
{
    ++z.H;
    return(z);
}

main()
{
    Time a(2, 30), b;
    //Implicit function call
    b = ++a;
    b.showdata();

    Time c(5, 10), d;
    //Explicit function call
    d = operator ++ (c);
    d.showdata();
}
```

Output:
3 Hours 30 Minutes
6 Hours 10 Minutes

Type Casting / Type Conversion with Objects

- There are three possible conversions can be done with object as follows:
  1. Type casting from user-defined to built-in
  2. Type casting from built-in to user-defined
  3. Type casting from user-defined to user-defined

Type casting from user-defined to built-in

- To convert an object into built-in datatype the class must contain a conversion function.
- **This conversion function will automatically call for the object when an object is attempt to assign to a variable of built-in datatype.**
- The conversion function have following general form.
- **Syntax:**

```
operator built_in_datatype()
{
    ....
    ....
}
```

Example: WAP to find length of a string using operator overloading

```
#include<string.h>
#include<iostream>
using namespace std;
class MyString
{
        private: char N[50];
        public:
                MyString(char n[])
                {
                        strcpy(N, n);
                }

                void showdata()
                {
                cout<<"String = "<<N<<endl;
                }

                operator int()
                {
                        return strlen(N);
                }
};
```

```
main()
{
        MyString a("JACK");
        a.showdata();
        int len = a;
        cout<<"Length = "<<len<<endl;
        cout<<"Length = "<<a<<endl;
}
```

Output:

```
String = JACK
Length = 4
Length = 4
```

Example: Design a class Circle with data member R and objects of the class can be created by passing data. Define converter function to convert a circle object to int and float.

```
#include<iostream>
using namespace std;
class Circle
{       private: int R;
        public:
                Circle(int r)
                {
                        R = r;
                }
                operator int ()
                {
                        return(R);
                }
                operator float ()
                {
                        return(3.14*R*R);
                }
};
```

```
main()
{
        Circle a(5);
        int x = a;
        cout<<"Radius = "<<x<<endl;
        float y = a;
        cout<<"Area = "<<y<<endl;
}
```

Output:

```
Radius = 5
Area = 78.5
```


Type casting from built-in to user-defined

- To convert built-in value into an object the class must contain a constructor with single argument.
- **The constructor will automatically invoke for the object when a builtin value is assigned to the object and the value of built-in datatype will be passed as argument to the constructor.**

Example: Design a class Currency with data members rupees & paise. Define member function showdata(). The objects of the class can be create by passing 2 or 0 arguments. Show the process of converting integer into Currency object in the class.

```
#include<iostream>
using namespace std;
class Currency
{
        private:
                int RS, PS;
        public:
                Currency(int rs, int ps)
                {
                        RS = rs;
                        PS = ps;
                }

                Currency()
                {
                        RS = 0;
                        PS = 0;
                }

                Currency(int n)
                {
                        RS = n / 100;
                        PS = n % 100;
                }

                void showdata()
                {
                cout<<RS<<" Rupees "<<PS<<" Paise"<<endl;
                }
};
```

```
main()
{
        Currency a(10, 29), b;
        a.showdata();
        b.showdata();
        int p = 5060;
        b = p;
        b.showdata();
}
```

Output:

```
10 Rupees 29 Paise
0 Rupees 0 Paise
50 Rupees 60 Paise
```

Example:

```
#include<iostream>
using namespace std;
class Circle
{
        private:
                int R;
        public:
                Circle()
                {
                        R = 0;
                }

                Circle(int r)
                {
                        R = r;
```

```
main()
{
        Circle a;
        a.area();       // R = 0
        a = 5;
        a.area();       // R = 5
}
```

Output:

```
Area is 0
Area is 78.5
```

```
    }
    void area()
    {
        float A = 3.14*R*R;
        cout<<"Area is "<<A<<endl;
    }
};
```

Typecasting from user-defined to user-defined

- To convert an object into another object of different class type, the object to which another object is assigned must have a constructor with single argument.
- And the class whose object is assigned must have a conversion function.
- It is supported in the "Turbo C++".

Example:

```
#include<iostream.h>
class MyInteger
{
    private:
        int N;
    public:
        MyInteger(int n)
        {
            N = n;
        }
        operator int()
        {
            return(N);
        }
};
```

```
class Currency
{
    private:
        int RS, PS;
    public:
        Currency(int rs, int ps)
        {
            RS = rs;
            PS = ps;
        }
        Currency()
        {
            RS = PS = 0;
        }
        Currency(int n)
        {
            RS = n / 100;
            PS = n % 100;
        }
        void showdata()
        {
        cout<<RS<<" Rupees "<<PS<<" Paise"<<endl;
        }
};
```

```
main()
{
    MyInteger a(1050);
    Currency b;
    b.showdata();
    b = a;
    b.showdata();
}
```

Output:

```
0 Rupees 0 Paise
10 Rupees 50 Paise
```

Array of objects

- If we require multiple objects of same class then we can create an array of objects.
- **Syntax:** `class_name arr_name[size];`

FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE

Example: Design a class Person having data members name & age. Define member functions readdata() and showdata(). Read data of 5 persons and display it.

```
#include<iostream>
using namespace std;
class Person
{
    private:
        char N[20];
        int A;
    public:
        void readdata()
        {
            cout<<"\nEnter Name and Age: ";
            cin>>N>>A;
        }
        void showdata()
        {
            cout<<"\nName = "<<N<<endl;
            cout<<"Age = "<<A<<" years"<<endl;
        }
};
```

```
main()
{
    Person a[5];
    for(int i = 0; i <= 4; i++)
    {
        a[i].readdata();
        a[i].showdata();
    }
}
```

Array within a class

Example: Design a class Student with data-members Roll Number, Student Name and Marks of 5 subjects. Define member function to read student information, calculate total marks and percentage. Define member function to display Student information.

```
#include<iostream>
using namespace std;
class Student
{
    private:
        int RN;
        char N[20];
        int M[5];
    public:
        void readdata()
        {
        cout<<"Enter RollNo: ";
        cin>>RN;

        cin.sync();
        cout<<"Enter Student Name: ";
        cin.getline(N, 19);

        cout<<"Enter marks of 5 subjects: ";
        for(int i=0; i<=4; i++)
        cin>>M[i];
        }
        void showdata()
        {
        int t = 0;
        for(int i=0; i<=4; i++)
        t = t + M[i];

        float p = t/5.0;

        cout<<"\nRoll Number = "<<RN<<endl;
        cout<<"Student Name = "<<N<<endl;
        cout<<"Total Marks = "<<t<<endl;
        cout<<"Percentage = "<<p<<"%"<<endl;
        }
};
```

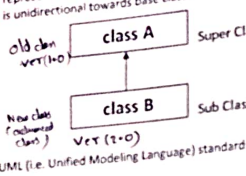
```
main()
{
    Student a;
    a.readdata();
    a.showdata();
}
```

Output:

```
Enter RollNo: 1010
Enter Student Name: JACK
Enter marks of 5 subjects:
60 61 62 63 64 <Enter>

Roll Number = 1010
Student Name = JACK
Total Marks = 310
Percentage = 62%
```

Inheritance

- **The reusability of existing classes without disturbing the existing classes is called as inheritance.**
- By using inheritance we can derive new class from an existing class.
- In this case the existing class is called as **base class / parent class / super class.**
- The newly created class is called as **derived class / child class / sub class**
- **The benefit of inheritance is that, the child class inherits all the properties (i.e. code) of the parent class.**
- **That means we don't need to write code for previous functionalities in the derived class, we only need to write code for new requirements in the derived class.**
- By using inheritance we can create new versions (i.e. advanced versions) of the old classes by adding some new features.
- We can say that the parent class is the version1.0 and child class is ther version2.0 (i.e. advanced version)
- Also we can create new classes which will represent to another entity which is partially same as the previous class.
- Inheritance represents IS-A relationship
- Inheritance is unidirectional towards base class.



Note:
- As per the UML (i.e. Unified Modeling Language) standards the derived class will point to the base class.

Deriving new class from an existing class

- We can derive a new class from an existing class as follows.
- **Syntax:**

```
class derived_class_name : visibility_mode base_class1, visibility_mode base_class2, ...
{
        access_specifier:
                data_members;
        access_specifier:
                member_functions;
};
```

- **Note: If we want to use the members (data-members) of base class in derived class then the members (data-members) in base class must be declared as protected.**

Method Resolution Order (MRO):

- Whenever a member_function is called with the derived class object then first that member_function will be search in the derived class, if not found then it will be search in its immediate base class and so on.

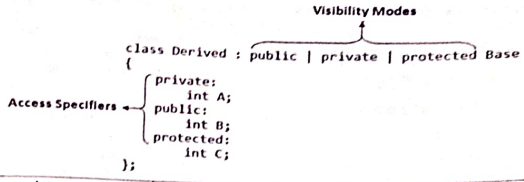
Visibility Modes

- Visibility mode decides the accessibility of base class members in derived classes.
- The visibility mode can be private, public or protected.
- While deriving new class, if the visibility mode is not specified then by default it is private.

Visibility Mode	Description
private	If a base class is inherited as private in the derived class then all the public and protected members of the base class becomes private in the derived class.
protected	If a base class is inherited as protected in the derived class then all the public and protected members of the base class becomes protected in the derived class.
public	If a base class is inherited as public in the derived class then all the public members remains public and all the protected members remains protected in the derived class.

FOR EDUCATIONAL PURPOSE ONLY, NOT FOR SALE

Example:



Base class members	Derived class accessibility		
	If the base class is inherited as private	If the base class is inherited as protected	If the base class is inherited as public
private members of base class	Not Accessible	Not Accessible	Not Accessible
protected members of base class	Becomes private	Becomes protected	Remains protected
public members of base class	Becomes private	Becomes protected	Remains public

Example: Design a class Calculator with data members A and B to store numbers. Define member-functions setdata(), add() and sub(). Derive a new class NewCalculator with additional member-functions mult() and div().

```
#include<iostream>
using namespace std;
class Calculator
{
        protected:
                int A, B;
        public:
        void setdata(int a, int b)
        {
                A = a;
                B = b;
        }
        int add()
        {
                return(A+B);
        }
        int sub()
        {
                return(A-B);
        }
};
```

```
class NewCalculator : public Calculator
{
        public:
        int mult()
        {
                return(A*B);
        }
        float div()
        {
                return((float)A/B);
        }
};
```

```
main()
{
NewCalculator a;
a.setdata(5, 7);
cout<<"Add = "<<a.add()<<endl;
cout<<"Sub = "<<a.sub()<<endl;
cout<<"Mul = "<<a.mult()<<endl;
cout<<"Div = "<<a.div()<<endl;
}
```

Example: Design a class Distance with data members Feet and Inches. Define member-functions setdata() & showdata(). Derive a class MyDistance from the class Distance having additional member-function convert() which will convert the data of MyDistance object into inches. (1 Feet = 12 Inches)

```
#include<iostream>
using namespace std;
class Distance
{
        protected:
                int F, I;
        public:
        void setdata(int f, int i)
```

```
class MyDistance : public Distance
{
        public:
        void convert()
        {
        int c = F * 12 + I;
        cout<<"Data is "<<c<<" inches"<<endl;
        }
```

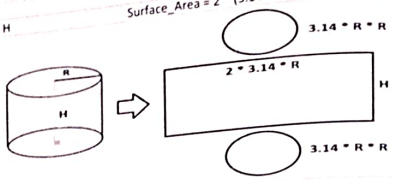
```
    {
        F = f;
        I = i;
    }

    void showdata()
    {
    cout<<F<<" Feets "<<I<<" Inches"<<endl;
    }
};
main()
{
    Distance a;
    a.setdata(5, 2);
    a.showdata();

    MyDistance b;
    b.setdata(2, 10);
    b.showdata();
    b.convert();
}
```

Output:
5 Feets 2 Inches
2 Feets 10 Inches
Data is 34 inches

Example: Design a class Cylinder with data-members radius & height. Define member-function setdata() and
Derive a class MyCylinder from the class Cylinder with additional member-function surfacearea().

Formulae:
Volume = 3.14 * R * R * H

Surface_Area = 2 * (3.14 * R * R) + (2 * 3.14 * R) * H




```
#include<iostream>
using namespace std;
class Cylinder
{
    protected:
        int R, H;
    public:
    void setdata(int r, int h)
    {
        R = r;
        H = h;
    }

    void volume()
    {
    float V = 3.14 * R * R * H;
    cout<<"Volume is "<<V<<endl;
    }
};
```

```
class MyCylinder : public Cylinder
{
    public:
    void surfacearea()
    {
    float A = 2*(3.14*R*R) + (2*3.14*R)*H;
    cout<<"Surfacearea is "<<A<<endl;
    }
};
main()
{
    Cylinder a;
    a.setdata(2, 3);
    a.volume();

    MyCylinder b;
    b.setdata(4, 5);
    b.volume();
    b.surfacearea();
}
```

Output:
Volume is 37.68
Volume is 251.2
Surfacearea is 226.08

Function Overriding

- **The process of redefining a base class function in derived class with exactly same prototype/signature** (i.e. same name, same number of arguments, same data-types of arguments and same return_type) **as the base function is called as function overriding.**
- **When the base class function does not work as per the requirements of derived class object then such function needs to redefine in the derived class.**

Example: Design a class Cylinder with data-members radius & height. Define member-functions setdata(), volume() & surfacearea(). Derive a class OpenCylinder (i.e. Cylinder with no top surface) from the class Cylinder having overridden member function surfacearea().

```
#include<iostream>
using namespace std;
class Cylinder
{
    protected:
        int R, H;
    public:
        void setdata(int r, int h)
        {
            R = r;
            H = h;
        }

        void volume()
        {
        float V = 3.14 * R * R * H;
        cout<<"Volume is "<<V<<endl;
        }

        void surfacearea()
        {
        float S = 2 * (3.14 * R * R) + (2 * 3.14 * R) * H;
        cout<<"Surfacearea of Cylinder is "<<S<<endl;
        }
};
```


```
class OpenCylinder : public Cylinder
{
    public:
    void surfacearea()
    {
    float S = (3.14 * R * R) + (2 * 3.14 * R) * H;
    cout<<"Surfacearea of OpenCylinder is "<<S<<endl;
    }
};
```

```
main()
{
    Cylinder a;
    a.setdata(4, 5);
    a.volume();
    a.surfacearea();

    OpenCylinder b;
    b.setdata(4, 5);
    b.volume();
    b.surfacearea();
}
```

Output:
Volume is 251.2
Surfacearea of Cylinder is 226.08
Volume is 251.2
Surfacearea of OpenCylinder is 175.84

Example: Design a class Person with data-members name & age. Define member-functions setdata() & showdata(). Derive a new class Citizen from the class Person having additional data-member nationality and member-functions setdata() & showdata().

```
#include<string.h>
#include<iostream>
using namespace std;
class Person
{   protected: char N[50]; int A;
    public:
    void setdata(char n[], int age)
    {
        strcpy(N, n);
        A = age;
    }
```

```
class Citizen : public Person
{   private: char NAT[20];
    public:
    // Overloaded function
    void setdata(char n[], int age, char nat[])
    {
        strcpy(N, n);
        A = age;
        strcpy(NAT, nat);
    }
```

```
    void showdata()
    {
    cout<<"\nName = "<<N<<endl;
    cout<<"Age = "<<A<<" years"<<endl;
    }
};
```

```
    void showdata()// Overrided function
    {
    cout<<"\nName = "<<N<<endl;
    cout<<"Age = "<<A<<" years"<<endl;
    cout<<"Nationality = "<<NAT<<endl;
    }
};
```

```
main()
{
        Person a;
        a.setdata("JACK", 20);
        a.showdata();

        Citizen b;
        b.setdata("JAMES", 22, "American");
        b.showdata();
}
```

Output:
Name = JACK
Age = 20 years

Name = JAMES
Age = 22 years
Nationality = American

Types of Inheritance:

- We can derive our classes in different ways as follows.

Sr. No.	Inheritance Type	Diagram
1	**Single Level Inheritance:** If a class is derived from a single independent class then it is called as single level inheritance.	Class A — Base Class; Class B — Derived Class
2	**Multi-Level Inheritance:** If a class is derived from a derived class then such type of inheritance is called as multilevel inheritance.	Class A — Base Class; Class B — Derived Class / Base Class; Class C — Derived Class
3	**Hierarchical Inheritance:** If two or more classes are derived from the same base class then such type of inheritance is called as hierarchical inheritance.	Class A — Base class; Class C1, Class C2, Class C3 — Derived Class

FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE

4	**Multiple Inheritance:** If a class is derived from multiple base classes then such type of inheritance is called as multiple inheritance.	Base class: Class A, Class B — Derived class: Class C
5	**Hybrid Inheritance:** If the mixture of inheritance is used then it is called as hybrid inheritance.	Class A; Class B, Class C; Class D

Calling base class overrided function in derived class

- To avoid retyping and repetition of the existing code we can call the base class function in derived class as follows.
- **Syntax :**
  base_class_name :: function_name(arg1, arg2, ...);

Example:

Single Level Inheritance

class Person

Name, Age
void readdata(void); void showdata (void);

class Citizen

Nationality
void readdata(void); void showdata (void);

```
#include<iostream>
using namespace std;
class Person
{
        protected: char N[50]; int A;
        public:
                void readdata()
                {
                cout<<"\nEnter Name and Age: ";
                cin>>N>>A;
                }
                void showdata()
                {
                cout<<"\nName = "<<N<<endl;
                cout<<"Age = "<<A<<" Yrs."<<endl;
                }
};
```

```
class Citizen : public Person
{
        private:
                char NAT[20];
        public:
        void readdata()
        {
        Person::readdata();
        cout<<"Enter Nationality: ";
        cin>>NAT;
```

```
main()
{
Person a;
a.readdata();
a.showdata();

Citizen b;
b.readdata();
b.showdata();
}
```

a: N JACK, A 20

b: N JAMES, A 20, NAT

Output:
Enter Name and Age: JACK 10

Name = JACK
Age = 10 Yrs.

Enter Name and Age: JAMES 20
Enter Nationality: American

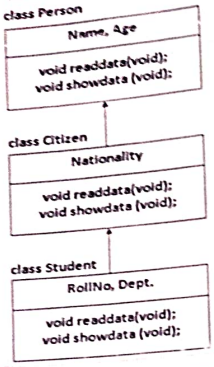
Name = JAMES

Age = 20 Yrs.
Nationality = American

```
    }
    void showdata()
    {
    Person::showdata();
    cout<<"Nationality = "<<NAT <<endl;
    }
};
```

Example:

Multi Level Inheritance



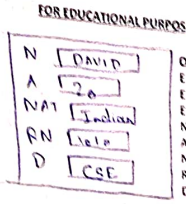

```
#include<iostream>
using namespace std;
class Person
{
        protected:
                char N[50];
                int A;
        public:
                void readdata()
                {
                cout<<"\nEnter Name and Age: ";
                cin>>N>>A;
                }
                void showdata()
                {
                cout<<"\nName = "<<N<<endl;
                cout<<"Age = "<<A<<" Yrs."<<endl;
                }
};
```

```
class Citizen : public Person
{
        protected:
                char NAT[20];
        public:
        void readdata()
        {
        Person::readdata();
        cout<<"Enter Nationality: ";
        cin>>NAT;
        }
        void showdata()
        {
        Person::showdata();
        cout<<"Nationality = "<<NAT <<endl;
        }
};
```

```
class Student : public Citizen
{
        private:
                int RN; char D[20];
        public:
        void readdata()
        {
        Citizen::readdata();
        cout<<"Enter Roll No. and Dept: ";
        cin>>RN>>D;
        }
        void showdata()
        {
        Citizen::showdata();
        cout<<"Roll No. = "<<RN<<endl;
        cout<<"Department = "<<D <<endl;
        }
};
```



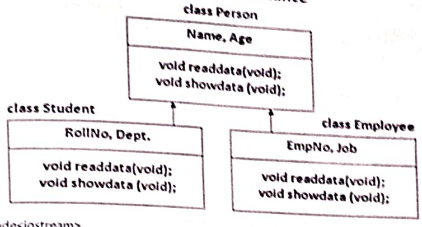
```
main()
{
        Student a;
        a.readdata();
        a.showdata();
}
```



Output:
Enter Name and Age: DAVID 20
Enter Nationality: Indian
Enter Roll No. and Dept: 1090 CSE
Name = DAVID
Age = 20 years
Nationality = Indian
Roll Number = 1090
Department = CSE

Example:

Hierarchical Inheritance

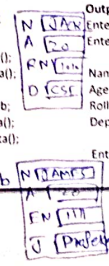



```
#include<iostream>
using namespace std;
class Person
{
        protected:
                char N[50]; int A;
        public:
        void readdata()
        {
        cout<<"\nEnter Name and Age: ";
        cin>>N>>A;
        }
        void showdata()
        {
        cout<<"\nName = "<<N<<endl;
        cout<<"Age = "<<A<<" Yrs." <<endl;
        }
};
```

```
class Student : public Person
{
        private:
                int RN; char D[20];
        public:
                void readdata()
                {
                Person::readdata();
                cout<<"Enter Roll No and Dept: ";
                cin>>RN>>D;
                }
                void showdata()
                {
                Person::showdata();
                cout<<"Roll No. = "<<RN<<endl;
                cout<<"Dept. = "<<D<<endl;
                }
};
```

```
class Employee : public Person
{
        private:
                int EN; char J[20];
        public:
        void readdata()
        {
        Person::readdata();
        cout<<"Enter EmpNo and Job: ";
        cin>>EN>>J;
```

```
main()
{
Student a;
a.readdata();
a.showdata();

Employee b;
b.readdata();
b.showdata();
}
```



Output:
Enter Name and Ag:- JACK 20
Enter Roll No and Dept: 1090 CSE

Name = JACK
Age = 20 Yrs.
Roll No. = 1090
Dept. = CSE

Enter Name and Age: JAMES 22

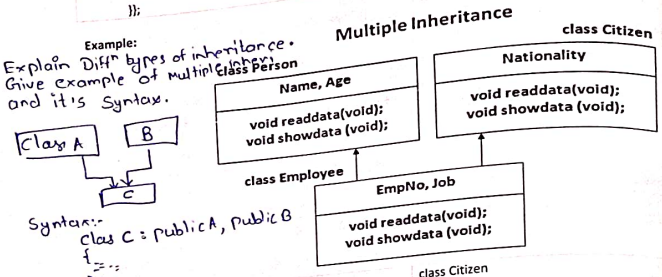
```
}
void showdata()
{
Person::showdata();
cout<<"Emp. No. = "<<EN <<endl;
cout<<"Job = "<<J<<endl;
});
```

```
Enter EmpNo and Job: 2010 Professor

Name = JAMES
Age = 22 years
Emp. No. = 2010
Job = Professor
```

Example:

Multiple Inheritance

Explain Diff^n types of inheritance. Give example of multiple inheri. and it's Syntax.



Syntax:-

```
Class C : public A, public B
{
:::
};
```

```
#include<iostream>
using namespace std;
class Person
{
        protected:
                char N[50];
                int A;
        public:
        void readdata()
        {
        cout<<"\nEnter Name and Age: ";
        cin>>N>>A;
        }

        void showdata()
        {
        cout<<"\nName = "<<N<<endl;
        cout<<"Age = "<<A<<" Yrs."<<endl;
        }
};
```

```
class Citizen
{
        protected:
                char NAT[20];
        public:
        void readdata()
        {
        cout<<"Enter Nationality: ";
        cin>>NAT;
        }

        void showdata()
        {
        cout<<"Nationality = "<<NAT<<endl;
        }
};
```

```
class Employee : public Person, public Citizen
{
        private:
                int EN;
                char J[20];
        public:
                void readdata()
                {
                Person::readdata();
                Citizen::readdata();
                cout<<"Enter EmpNo and Job: ";
```

```
main()
{
        Employee a;
        a.readdata();
        a.showdata();
}
```

Output:

```
Enter Name and Age: DAVID 22
Enter Nationality: Indian
Enter EmpNo and Job: 1090 Engineer
```



```
cin>>EN>>J;
}

void showdata()
{
Person::showdata();
Citizen::showdata();
cout<<"Emp. No. = "<<EN<<endl;
cout<<"Job = "<<J<<endl;
}
};
```

```
Name = DAVID
Age = 22 Yrs.
Nationality = Indian
Emp.No. = 1090
Job = Engineer
```

Ambiguity in Multiple Inheritance

- Suppose, two base classes have a function with same name which is not overridden in derived class.
- Now, if we try to call the function with the object of the derived class then, compiler shows error.
- This is because compiler will confuse which function to call.

Example:

```
#include<iostream>
using namespace std;
class Base1
{
        public:
        void display()
        {
        cout<<"Base1- display"<<endl;
        }
};

class Base2
{
        public:
        void display()
        {
        cout<<"Base2- display"<<endl;
        }
};
```

```
class Derived : public Base1, public Base2{};

main()
{
        Derived a;
        a.Base1::display();
        a.Base2::display();
}
```

Output:

```
Base1- display
Base2- display
```

Note:

```
Derived a;
a.display(); //It will cause error
```

Calling function of grand-parent:

Example:

```
#include<iostream>
using namespace std;
class Alpha
{
        public:
        void display()
        {
        cout<<"Alpha: display"<<endl;
        }
};
```

```
class Beta : public Alpha
{
        public:
        void display()
        {
        cout<<"Beta: display\n";
        }
};
```

```
main()
{
Beta a;
a.display();
a.Alpha::display();
}
```

Output:

```
class Beta: display function
class Alpha: display function
```

Virtual Base Class IMP

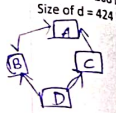
- If the hybrid inheritance involves multiple inheritance in it, then there is possibility that a base class is ineherited in derived class multiple times, and **if this happened then it will create multiple copies of base class data-members in the derived class.**
- These multiple copies of base class data-members introduce ambiguity (i.e. confusing results).

- These multiple copies of base class data-members can be avoided by inheriting the base class virtually in the derived class.
- If a base class is inherited virtually in the derived class then C++ takes special care to see that only one copy of base class data-members to be inherited in the derived class.
- **Example:**

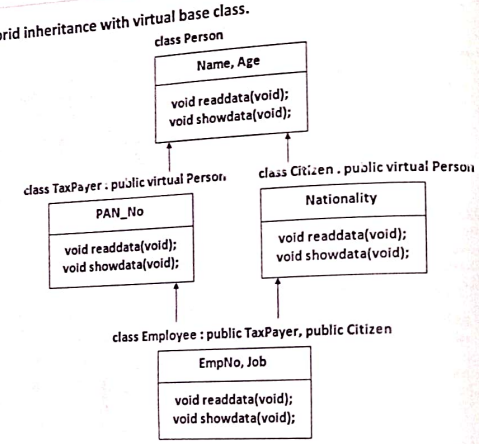
```
#include<iostream>
using namespace std;
class A
{
        protected:
                char x[100];
};
class B : public virtual A
{
        protected:
                char y[100];
};
class C : public virtual A
{
        protected:
                char z[100];
};
class D : public B, public C
{
protected:
        char p[100];
};
main()
{
A a;
cout<<"Size of a = "<<sizeof(a)<<" bytes"<<endl;
B b;
cout<<"Size of b = "<<sizeof(b)<<" bytes"<<endl;
C c;
cout<<"Size of c = "<<sizeof(c)<<" bytes"<<endl;
D d;
cout<<"Size of d = "<<sizeof(d)<<" bytes"<<endl;
}
```

Output:

```
Size of a = 100 bytes
Size of b = 208 bytes
Size of c = 208 bytes
Size of d = 424 bytes
```



Example: Hybrid inheritance with virtual base class.




```
#include<iostream>
using namespace std;
class Person
{
        protected:
                char N[50];
                int A;
        public:
        void readdata()
        {
        cout<<"\nEnter Name and Age: ";
        cin>>N>>A;
        }

        void showdata()
        {
        cout<<"\nName = "<<N<<endl;
        cout<<"Age = "<<A<<" Yrs."<<endl;
        }
};
class TaxPayer : public virtual Person
{
        protected:
                char PAN_NO[20];
        public:
        void readdata()
        {
        Person::readdata();
        cout<<"Enter PAN number: ";
        cin>>PAN_NO;
        }

        void showdata()
        {
        Person::showdata();
        cout<<"PAN No. = "<<PAN_NO<<endl;
        }
};
class Citizen : public virtual Person
{
        protected:
                char NAT[20];
        public:
        void readdata()
        {
        Person::readdata();
        cout<<"Enter Nationality: ";
        cin>>NAT;
        }

        void showdata()
        {
        Person::showdata();
        cout<<"Nationality = "<<NAT<<endl;
        }
};
class Employee : public TaxPayer, public Citizen
{
        private:
                int EN;
                char J[20];
        public:
                void readdata()
                {
                TaxPayer::readdata();
                cout<<"Enter Nationality: ";
                cin>>NAT;
                cout<<"Enter EmpNo and Job: ";
                cin>>EN>>J;
                }

                void showdata()
                {
                TaxPayer::showdata();
                cout<<"Nationality = "<<NAT<<endl;
                cout<<"Emp. No. = "<<EN<<endl;
                cout<<"Job = "<<J<<endl;
                }
};
main()
{
        Employee a;
        a.readdata();
        a.showdata();
}
```

Output:

```
Enter Name and Age: JACK 22
Enter PAN number: ABCD1234
Enter Nationality: Indian
Enter EmpNo and Job: 1090 Engineer

Name = JACK
Age = 22 Yrs.
PAN Number = ABCD1234
Nationality = Indian
Emp. No. = 1090
Job = Engineer
```

Constructors and Destructors in inheritance

- If a base class and derived class both contains default constructor then when we create the derived t class object then both the constructors are executed.
- First the constructor of base class is executed and then the constructor of derived class is executed.

Behavior of default constructor in inheritance.

Example-1:

```
#include<iostream>
using namespace std;
class Base
{
        public:
        Base()
        {
        cout<<"Base: constructor"<<endl;
        }
};
class Derived : public Base {};

main()
{
        Derived a;
}
```

Output:
Base: constructor

Example-2:

```
#include<iostream>
using namespace std;
class Base
{
        public:
        Base()
        {
        cout<<" Base: constructor"<< "\n";
        }
};
class Derived : public Base
{
        public:
        Derived()
        {
        cout<<"Derived: constructor"<< "\n";
        }
};
main()
{
Derived a;
}
```

Output:
Base: constructor
Derived: constructor

Example-3:

```
#include<iostream>
using namespace std;
class Base
{
        public:
        Base()
        {
        cout<<"Base: Default constructor"<<endl;
        }
};

class Derived : public Base
{
        public:
        Derived(int a)
        {
        cout<<"Derived class: Parameterized constructor"<<endl;
        }
};
main()
{
        Derived a(10);
}
```

Output:
Base class: default constructor
Derived class: Parameterized constructor

- **Note:** In the case of derived class object the constructors are always invoke in the exactly same sequence in which the classes are inherited in the derived class and destructors are executed in exactly reverse sequence of classes inherited in derived class.

Example:

```
#include<iostream>
using namespace std;
class Alpha
{
        public:
        Alpha()
        {
        cout<<"Alpha: Constructor"<<endl;
        }

        ~Alpha()
        {
        cout<<"Alpha: Destructor"<<endl;
        }
};
class Beta
{
        public:
        Beta()
        {
        cout<<"Beta: Constructor"<<endl;
        }

        ~Beta()
        {
        cout<<"Beta: Destructor"<<endl;
        }
};
class Gamma : public Beta, public Alpha
{
        public:
        Gamma()
        {
        cout<<"Gamma: Constructor"<<endl;
        }

        ~Gamma()
        {
        cout<<"Gamma: Destructor"<<endl;
        }
};
main()
{
        Gamma *p;
        p = new Gamma();
        delete p;
}
```

Output:
Beta: Constructor
Alpha: Constructor
Gamma: Constructor
Gamma: Destructor
Alpha: Destructor
Beta: Destructor

Calling base class parameterized constructor from derived class constructor

- **To avoid retyping and repetition of code we can call the base class constructor in derived class.**
- If a base class contains a constructor with arguments then such a base class constructor is not automatically invoke.
- **Syntax:**

```
derived_class_constructor(datatype arg1, ....) : base1(agr1, arg2,...), base2(arg1, arg2,...), ....
{
    ....
    ....
}
```

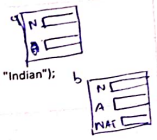
Example: Design a class Person with data members Name & Age. Define member function showdata(). The object of the class must created by passing data. Derive a new class Citizen from the class Person having additional data member Nationality. Define member function showdata and object of the class must be created by passing data.

```
#include<string.h>
#include<iostream>
using namespace std;
class Person
{
        protected: char N[50]; int A;
        public:
        Person(char n[], int age)
        {
                strcpy(N, n);
                A = age;
        }
class Citizen : public Person
{
        protected:
                char NAT[20];
        public:
        Citizen(char n[], int age, char nat[]) : Person(n, age)
        {
                strcpy(NAT, nat);
        }

        void showdata()
```

```
}
void showdata()
{
cout<<"\nName = "<<N<<endl;
cout<<"Age = "<<A<<" years"<<endl;
}
};
```

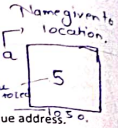
```
{
Person::showdata();
cout<<"Nationality = "<<NAT<<endl;
}
};
```

```
main()
{
    Person a("JACK", 10);
    a.showdata();
    Citizen b("JAMES", 20, "Indian");
    b.showdata();
}
```



Output:
Name = JACK
Age = 10 years

Name = JAMES
Age = 20 years
Nationality = Indian



Pointers

- **Variable:**
  - ✓ The variable is a memory location with specified name.
  - ✓ Every memory location have a unique address, that means every variable have a unique address.
  - ✓ The variable is used to store values.

- **Pointer:**
  The pointer is a variable which can be used to store address of a variable or a memory location or an object.
- ★★★ The pointer always points to the variable or memory location or object whose address is currently present in the pointer.
- **The pointers are mainly used to reserve and release the memory dynamically (i.e. during the execution of program or at run-time)**
- Also, pointers are used to perform following operations:
  - ✓ To pass array as argument
  - ✓ To return multiple values from a function.
- A pointer can be defined by using following syntax:
  Syntax: datatype *pointer_name;

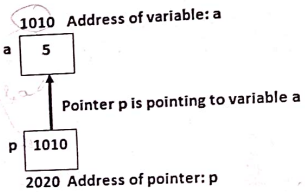
Storing address of a variable in the pointer

- The "address_of" (i.e. &) operator returns address of a variable.
- The address of a variable can be store in the pointer by using following syntax.

Syntax:
pointer_variable = &variable;

Example:

```
int a = 5, *p;
p = &a;
    1010
```



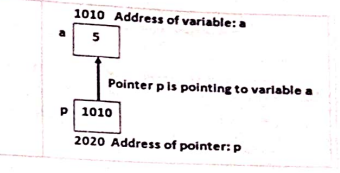
Accessing value by using pointer

- If a pointer is representing | pointing to a variable or memory location then the value of the variable or memory location can be get | access by using pointer as follows:
- **Syntax:**
  ***pointer_name;**

Example:

```
#include<iostream>
using namespace std;
main()
{
int a = 5, *p;
p = &a;

cout<<"a = "<<a<<endl;
cout<<"&a = "<<&a<<endl;
cout<<"p = "<<p<<endl;
cout<<"&p = "<<&p<<endl;
cout<<"*p = "<<*p<<endl;
cout<<"*&a = "<<*&a<<endl;
}
```

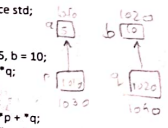
Output:
a = 5
&a = 1010
p = 1010
&p = 2020
*p = 5
*&a = 5



Example

```
#include<iostream>
using namespace std;
main()
{
    int a = 5, b = 10;
    int *p, *q;
    p = &a;
    q = &b;
    int x = *p + *q;
    cout<<"x = "<<x<<endl;
    int y = *p * *q;
    cout<<"y = "<<y<<endl;
}
```

Output:
x = 15
y = 50



Example

```
#include<iostream>
using namespace std;
main()
{
    int a = 5, *p = &a;
    cout<<"a = "<<a<<endl;
    cout<<"*p = "<<*p<<endl;
    (*p)++;
    cout<<"a = "<<a<<endl;
    cout<<"*p = "<<*p<<endl;
}
```

Output:
a = 5
*p = 5
a = 6
*p = 6

Example: Find output of the following program

```
#include<iostream>
using namespace std;
main()
{
    int a = 5, b = 10;
    int *p = &a;
    int *q = &b;
    *p = (*p) * 2;
    *q = (*q) * (*p);
    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
    cout<<"*p = "<<*p<<endl;
    cout<<"*q = "<<*q<<endl;
}
```

Output:
a = 10
b = 100
*p = 10
*q = 100

Example: Find output of the following program

```
#include<iostream>
using namespace std;
main()
{
int a = 2, *p, *q;
p = &a;
q = p;
cout<<a<<"\t"<<*p<<"\t"<<*q<<endl;

a++;
(*p)++;
(*q) *= 2;
cout<<a<<"\t"<<*p<<"\t"<<*q<<endl;
}
```

Output:
2 2 2
6 6 6

Pointer Arithmetic

We can use only +, -, ++, --, += and -= operators with the pointer variable itself.

- Adding 1 with pointer instruct the pointer to point to the next value in the memory.
- Similarly, subtracting 1 from the pointer instruct the pointer to point to the previous value in the memory.
- That's why:

1. If we add 1 with integer pointer then the address in the pointer will be increase by 2, and if we subtract 1 from integer pointer then the address in the pointer will be decrease by 2. Because the next or previous integer value is 2 bytes away from its current location.
2. If we add 1 with float pointer then the address in the pointer will be increase by 4, and if we subtract 1 from float pointer then the address in the pointer will be decrease by 4. Because the next or previous float value is 4 bytes away from its current location.
3. Same for remaining datatypes.

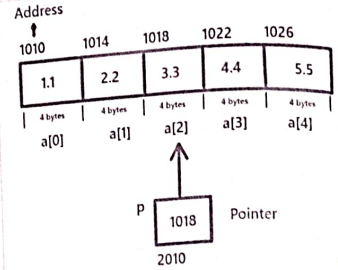
- **Note:** The size and capacity of data-types may change in different softwares.

Example:

```
#include<stdio.h>
#include<iostream>
using namespace std;
main()
{
        float a[] = {1.1, 2.2, 3.3, 4.4, 5.5};
        float *p;
        p = &a[2];
        cout<<"*p = "<<*p<<endl;
        printf("p = %u\n", p);
        p-=2;
        cout<<"*p = "<<*p<<endl;
        printf("p = %u\n", p);
}
```

Output:

```
*p = 3.3
p = 1018
*p = 1.1
p = 1010
```



Call by value, Call by reference, Pass by reference

- These concepts are used to check that the function can change the value of the actual arguments or not.
- And what are the reasons behind the result.

Call by value

1. Calling a function by passing a variable or value is called as call by value.
2. In call by value the copy of the value of the actual argument is passed as formal argument to the function.
3. The formal arguments are local variables, so they created each time when the function is called and gets destroyed as soon as the function is finished.
4. The function performs all its operations on or using the formal argument. That means the function cannot perform any operation on the actual argument.
5. Hence, in call by value the function cannot change the value of the actual argument.

Example:

```
#include<iostream>
using namespace std;
// Here, variable z is Formal argument
void square(int z)
{       z = z * z;
        cout<<"z = "<<z<<endl;
}
main()
{
        int z = 9;
        cout<<"z = "<<z<<endl;
        square(z);
        // Here, variable z is Actual argument
        cout<<"z = "<<z<<endl;
}
```

Output:

```
z = 9
z = 81
z = 9
```


FOR EDUCATIONAL PURPOSE ONLY, NOT FOR SALE

Call by reference | Call by address | Pass by pointers

1. Calling function by passing address of a variable is called as call by reference.
2. In call by reference the address of the variable is transferred as formal argument to the function.
3. The address of each variable is unique. That means two variable cannot have same address.
4. As the function receives address of a variable as argument so, the function can directly access the actual argument and can change value of the actual argument.
5. Hence, in call by reference the function can change the value to change the actual arguments.

Example:

```
#include<iostream>
using namespace std;
void square(int *p)
{
        *p = (*p) * (*p);
        cout<<"*p = "<<*p<<endl;
}
main()
{
        int z = 9;
        cout<<"z = "<<z<<endl;
        square(&z);
        cout<<"z = "<<z<<endl;
}
```

Output:

```
z = 9
*p = 81
z = 81
```

Pass by reference

- If the formal arguments of a function are reference variables then a new storage location is not created for these formal arguments.
- The reference variables is an alternate name (i.e. alias) to the actual argument and represent the same memory location of the actual arguments.
- Hence in pass by reference the function can change the value of the actual arguments.

Example:

```
#include<iostream>
using namespace std;
void test(int &b)
{
        cout<<"&b = "<<&b<<endl;
        b++;
}
main()
{
        int a = 5;
        cout<<"&a = "<<&a<<endl;
        cout<<"a = "<<a<<endl;
        test(a);
        cout<<"a = "<<a<<endl;
}
```

Output:

```
&a = 1090
a = 5
&b = 1090
a = 6
```

this pointer | Self referencing pointer

- **"this" is the name given to a pointer variable.**
- The "this" pointer contains address of the object for which this function or constructor or destructor is currently called.
- "this" points to the current object inside the class only.
- **By using the "this" pointer the class determines data of which object to be used for processing.**
- The expressions in the class are replaced by using "this" pointer in the compile code as follows:
  Example:
  float A = 3.14 * R * R;
  The above statement will be replace by following statement.
  float A = 3.14 * this->R * this->R;
- It is generally used to access the members of the object inside the class.
- Also it is used to return the object itself.
- **Note:** The copy constructor, overloaded assignment (=) operator function and "this" pointer is automatically added in the compile code of the class when the C++ program is compiled.

Example: To check whether this contains address of the current object or not.

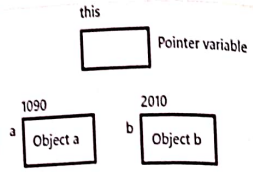
```
#include<iostream>
using namespace std;
class Alpha
{
        public:
        Alpha()
        {
        cout<<"Constructor: this = "<<this<<endl;
        }
main()
{
Alpha a;
cout<<"Address of obj-a = "<<&a<<endl;

Alpha b;
cout<<"Address of obj-b = "<<&b<<endl;
```

```
}

void display()
{
cout<<"Function: this = "<<this<<endl;
}

~Alpha()
{
cout<<"Destructor: this = "<<this<<endl;
}
};
```

```
a.display();
b.display();
}
```

Output:

```
Constructor: this = 1090
Address of obj-a = 1090

Constructor: this = 2010
Address of obj-b = 2010

Function: this = 1090
Function: this = 2010

Destructor: this = 1090
Destructor: this = 2010
```



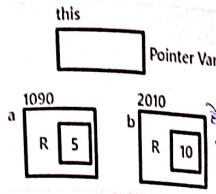
Example:

```
#include<iostream>
using namespace std;
class Circle
{
        private:
                int R;
        public:
        Circle(int r)
        {
        cout<<"\nConstructor, this = "<<this<<endl;
        this->R = r;
        }

        void area()
        {
        cout<<"\narea fun., this = "<<this<<endl;
        float A = 3.14 * this->R * this->R;
        cout<<"Area is "<<A<<endl;
        }

        void circumference()
        {
        cout<<"\ncircum. fun. this = "<<this<<endl;
        float C = 2 * 3.14 * this->R;
        cout<<"Circumference is "<<C<<endl;
        }
};
```

```
main()
{
        Circle a(5);
        cout<<"Address of a = "<<&a<<endl;
        Circle b(10);
        cout<<"Address of b = "<<&b<<endl;
        a.area();
        b.area();
        a.circumference();
        b.circumference();
}
```

Output:

```
Constructor, this = 1090
Address of a = 1090

Constructor, this = 2010
Address of b = 2010

area fun., this = 1090
Area is 78.5

area fun., this = 2010
Area is 314

circum. fun. this = 1090
Circumference is 31.4
```


```
circum. fun. this = 2010
Circumference is 62.8
```

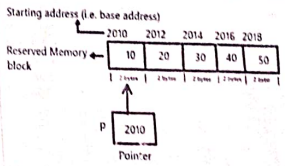
Dynamic Memory Management

- Dynamic means runtime (i.e. when the program is executing).
- Dynamic memory management means reserving and releasing memory during the execution of program.
- To reserve and release memory at runtime C++ provides us two special operators ***new*** and ***delete***.

new	▪ The new operator reserves a memory block for specified datatype and returns the starting address of reserved memory otherwise NULL be return. ▪ **Syntax: pointer_name = new datatype[int_size];**
delete	▪ The delete operator instructs the operating system to releases the memory block which was reserved by using new operator. ▪ **Syntax: delete pointer_name;**

Example: WAP to read N numbers and find their sum and mean

```
#include<iostream>
using namespace std;
main()
{
int N;
cout<<"How many integers you want to add?: ";
cin>>N;

int *p = new int[N];

cout<<"Enter "<<N<<" integers: ";

for(int i = 0; i < N; i++)
cin>>*(p+i);

int sum = 0;

for(int i = 0; i < N; i++)
sum = sum + *(p+i);

delete p;
cout<<"Sum = "<<sum<<endl;

float m = sum / (float)N;
cout<<"Mean = "<<m<<endl;
}
```



Output:

```
How many integers you want to add?: 5 <Enter>
Enter 5 integers: 10 20 30 40 50 <Enter>
Sum is 150
Mean is 30
```

Object and Pointer:

- If required we can store address of an object in the pointer.
- **Syntax:**

```
class_name object_name;
class_name *ptr_name = &object_name;
```

Accessing object members by using pointer:

- If the pointer is pointing to an object then the members of the object can be access by using following way.
- **Syntax:**
  - **1) ptr_name -> member_name;**
  - **2) (*ptr_name).member_name;**

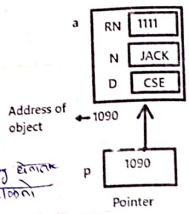
Example:

```
#include<string.h>
#include<iostream>
using namespace std;
class Student
{
        private:
                int RN;
                char N[20], D[20];
        public:
                Student(int rn, char n[], char d[])
                {
                        RN = rn;
                        strcpy(N, n);
                        strcpy(D, d);
                }

                void showdata()
                {
                cout<<"\nRoll Number = "<<RN<<endl;
                cout<<"Student Name = "<<N<<endl;
                cout<<"Department = "<<D<<endl;
                }
};
```

```
main()
{
        Student a(1111, "JACK", "CSE");
        Student *p;
        p = &a;

        p -> showdata();
        (*p).showdata();

}
```

Output:

```
Roll Number = 1111
Student Name = JACK
Department = CSE

Roll Number = 1111
Student Name = JACK
Department = CSE
```



Normal obj मधून structure Destroy होतात Pointer obj साठी वेगळे Destroy करा लागतो

Creating objects dynamically| Dynamic initialization of object| Pointer to object

- If the object is created dynamically then, we can destroy the object at run time if the object is no more required.
- The object can be create & destroy dynamically as follows:

Description	Syntax
To create object dynamically	**class_name *ptr_name = new class_name(val1, val2, ...);**
To destroy object dynamically	**delete ptr_name;**

Destructors | Deconstructors

- The destrcutor is a special member function in the class which is automatically invoked just before the object is going to destroy.
- **The destructor is used to truncate the object.** That means to release all the resources hold by the object just before the object is going to destroy.
- Destructor have same name as the class name with tilde (~) operator preceded.
- Destructor is automatically called just before the object is going to destroyed.
- Destructor cannot have any return_type. Hence cannot return value.
- Destructor cannot accept any argument hence not possible to overload.

- Destructor cannot inherit in derived classes.
- In case of derived class the destructors invoke in exactly reverse order in which the class is inherited.
- Destructor is used to release the resources (memory resources) hold by the object before the object is going to destroy.
- **Note:** A destructor is called automatically in following cases:
  1. When the function containing local objects ends
  2. When a block containing local objects ends
  3. When delete operator used.

Dynamic Constructor

- The constructor which runtime reserves memory for the data-members of the class is called as dynamic constructor.

Example: The following example contains, use of new & delete, destructor, dynamic constructor (dynamic initialization of object), and Pointer to object.

```
#include<iostream>
using namespace std;
#include<string.h>
class MyString
{       private: char *str;
        public:
        MyString(char s[]) // Dynamic constructor
        {
        str = new char[strlen(s) + 1];
        strcpy(str, s);
        }

        void showdata()
        {
        cout<<"String = "<<str<<endl;
        }

        ~MyString() // Destructor
        {
        delete str;
        cout<<"Object members truncated.\n";
        }
};
```

```
main()
{
MyString *ptr = new MyString("JACK");
ptr -> showdata();
delete ptr;
}
```

OR

```
main()
{
        MyString a("JACK");
        a.showdata();
}
```

Output:

```
String = JACK
Object members truncated..
```

Polymorphism

- Polymorphism indicates the similar behavior of objects of different classes.
- Similar behavior can be maintain in different classes by using inheritance i.e. by deriving the classes from same base class.
- The polymorphism can be achieved only for those classes whose base class is same and behavior is same.
- By using polymorphism we can develop a code which will work for different objects of different classes whose behavior is same.
- Such code is called as polymorphic code and the process to develop such code is called as polymorphism.
- Polymorphism is of two types as follows:
  1) Compile time polymorphism
  2) Run time polymorphism

Compile time polymorphism:

- **In compile time polymorphism compiler decides the function call at compile time (i.e. before the program begin is execution) as per the type of pointer.**
- The compile time polymorphism is also called as ***early binding, Static linking*** or ***static binding***.
- **C++ by default performs the Compile Time Polymorphism.**

Run time polymorphism:

- In run time polymorphism compiler decides the function call during the execution of program (i.e. at runtime) as per the type of object created.
- The run time polymorphism is also called as ***Late binding, Dynamic linking or Dynamic binding.***

Important:

- **C++ by default performs the Compile Time Polymorphism.**
- If we want to achieve run time polymorphism then, C++ provides us a special feature and the concept of virtual functions.
- **The Special feature is as follows:**
  - ✓ **"By using a base class pointer it is possible to create an object of derived class dynamically".**
  - ✓ But by using base class pointer we can call only the inherited and overrided functions for the derived class object.
  - ✓ That means by using base class pointer it is not possible to call the newly defined functions in derived class.

Example: To check the special feature and confirm the default polymorphism

```
#include<iostream>
using namespace std;
class Base
{
        public:
        void fun1()
        {
        cout<<"Base: fun1"<<endl;
        }

        void fun2()
        {
        cout<<"Base: fun2"<<endl;
        }
};
class Derived : public Base
{
        public:
        // Overrided
        void fun1()
        {
        cout<<"Derived: fun1"<<endl;
        }

        // Not Inherited or Not overrided function
        void fun3()
        {
                cout<<"Derived: fun3"<<endl;
        }
};
main()
{
        Base *p = new Derived();        // Valid
        p -> fun1();    // Overrided function
        p -> fun2();    // Inherited function
        //p -> fun3();  // Error: Because this is not inherited or overrided
        delete p;
}
```

Output:

```
Base: fun1
Base: fun2
```

Example-1: To check default polymorphism and special feature

```
#include<iostream>
using namespace std;
class Unit
{
        public:
        void showdata()
        {
        cout<<"Unit class showdata function\n";
        }

        void convert()
        {
        cout<<"Unit class convert function\n";
        }
};
class Distance : public Unit
{
        private:
                int F,I;
        public:
        Distance(int f, int i)
        {
                F = f;
                I = i;
        }

        void showdata()
        {
                cout<<F<<" Feet "<<I<<" Inches\n";
        }

        void convert()
        {
        int z = F*12 + I;
        cout<<"Data is "<<z<<" inches\n";
        }
};
class Time : public Unit
{
        private:
                int H, M;
        public:
        Time(int h, int m)
        {
                H = h;
                M = m;
        }

        void showdata()
        {
        cout<<H<<" Hours "<<M<<" Minutes"<<endl;
        }

        void convert()
        {
        int z = H*60 + M;
        cout<<"Data is "<<z<<" minutes"<<endl;
        }
};
main()
{
        Unit *p;
        p = new Distance(5, 2);
        p->showdata();
        p->convert();
        delete p;

        Unit *q;
        q = new Time(4, 10);
        q->showdata();
        q->convert();
        delete q;
}
```

Output:

```
Unit class showdata function
Unit class convert function
Unit class showdata function
Unit class convert function
```

Note: See the output of above program carefully, here only the functions of class Unit are executed. This happened because C++ by default performs Compile Time Polymorphism.

Virtual functions

- If we want to override a function in derived class then such a function should always be declared as virtual in base class even if it is not required.
- **If an overrided function is declared as virtual in base class then C++ decides the function call at the time of program execution as per the type of the object and ignoring the type of the pointer.**
- A base class function can be declared as virtual by preceding the function prototype with the keyword ***virtual***.

Example-2: Virtual function and runtime polymorphism.

```
#include<iostream>
using namespace std;
class Unit
{
        public:
        virtual void showdata()
        {
        cout<<"Unit class showdata function\n";
        }

        virtual void convert()
        {
        cout<<"Unit class convert function\n";
        }
};
class Distance : public Unit
{
        private:
                int F,I;
        public:
        Distance(int f, int i)
        {
                F = f;
                I = i;
        }
        void showdata()
        {
        cout<<F<<" Feet "<<I<<" Inches"<<endl;
```


FOR EDUCATIONAL PURPOSE ONLY, NOT FOR SALE

```
    }
};
```

```
    }
    void convert()
    {
    int z = F*12 + I;
    cout<<"Data is "<<z<<" inches"<<endl;
    }
};
```

```
class Time : public Unit
{
    private:
        int H, M;
    public:
    Time(int h, int m)
    {
        H = h;
        M = m;
    }

    void showdata()
    {
    cout<<H<<" Hours "<<M<<" Minutes"<<endl;
    }

    void convert()
    {
    int z = H*60 + M;
    cout<<"Data is "<<z<<" minutes"<<endl;
    }
};
```

```
main()
{
        Unit *p;
        p = new Distance(5, 2);
        p->showdata();
        p->convert();
        delete p;

        Unit *q;
        q = new Time(4, 10);
        q->showdata();
        q->convert();
        delete q;
}
```

Output:

```
5 Feet 2 Inches
Data is 62 inches
4 Hours 10 Minutes
Data is 250 minutes
```

Example-3: What if the virtual function is not overrided in the derived classes?

```
#include<iostream>
using namespace std;
class Unit
{
    public:
    virtual void showdata()
    {
    cout<<"Unit class showdata function"<<endl;
    }

    virtual void convert()
    {
    cout<<"Unit class convert function"<<endl;
    }
};
```

```
class Distance : public Unit
{
    private:
        int F,I;
    public:
        Distance(int f, int i)
        {
            F = f;
            I = i;
        }
};
```

```
class Time : public Unit
{
    private:
        int H, M;
    public:
        Time(int h, int m)
        {
            H = h;
            M = m;
```

```
main()
{
        Unit *p;
        p = new Distance(5, 2);
        p->showdata();
        p->convert();
        delete p;

        Unit *q;
```

Output:

```
Unit class showdata function
Unit class convert function
Unit class showdata function
Unit class convert function
```



```
    }
};
```

```
        q = new Time(4, 10);
        q->showdata();
        q->convert();
        delete q;
}
```

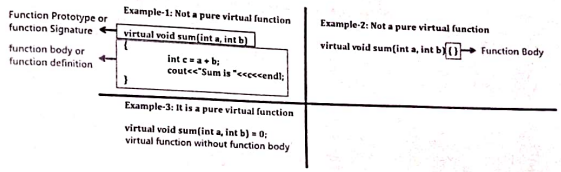
- **Note:** In above program it is not compulsory to override the base class functions in derived classes but if we want to make compulsion to override the base class function in derived classes then the base class virtual functions must be declared as pure virtual.

Pure virtual functions / abstract functions

- A virtual function without function definition (i.e. function body) is called as pure virtual function.
- The pure virtual functions are also called as abstract functions.
- A pure virtual function can be defined as follows.
- **Syntax:**

  virtual return_type function_name(data_type_of_arg1, data_type_of_ arg2, ...) = 0;

Example:



Abstract classes

- If a class contains at least one pure virtual function then such a class becomes an abstract class.
- An abstract class cannot be instantiated. Means it is not possible to create objects of an abstract class but, it is possible to create pointers of the abstract class.
- It is possible to derive new classes from an abstract class but if a class is derived from an abstract class then in the derived class we need to override all the pure virtual functions inherited from the base class otherwise the newly derived class will also become an abstract class.
- **The abstract class is used to forcefully override some functions in the derived classes.**
- **Example:**

```
#include<iostream>
using namespace std;
// Abstract class
class Rectangle
{   protected: int L, B;
    public:
    void setdata(int l, int b)
    {
            L = l;
            B = b;
    }

    virtual void area()
    {
            int A = L * B;
            cout<<"Area is "<<A<<endl;
    }
```

```
class MyRectangle : public Rectangle
{
    public:
    void perimeter()
    {
    int P = 2*(L + B);
    cout<<"Perimeter is "<<P<<endl;
    }
};

main()
{
    MyRectangle a;
    a.setdata(9, 3);
    a.area();
    a.perimeter();
}
```

```
        // pure virtual or abstract function
        virtual void perimeter() = 0;
};
```

Output:
Area is 27
Perimeter is 24

Example-4: After defining abstract functions in the base-class

```
#include<iostream>
using namespace std;
class Unit
{
        public:
        virtual void showdata()=0;
        virtual void convert()=0;
};

class Distance : public Unit
{
        private:
                int F,I;
        public:
                Distance(int f, int i)
                {
                        F = f;
                        I = i;
                }

                void showdata()
                {
                cout<<F<<" Feet "<<I<<" Inches"<<endl;
                }

                void convert()
                {
                int z = F*12 + I;
                cout<<"Data is "<<z<<" inches"<<endl;
                }
};
```

```
class Time : public Unit
{
        private:
                int H, M;
        public:
                Time(int h, int m)
                {
                        H = h;
                        M = m;
                }

                void showdata()
                {
                cout<<H<<" Hours "<<M<<" Minutes"<<endl;
                }

                void convert()
                {
                int z = H*60 + M;
                cout<<"Data is "<<z<<" minutes"<<endl;
                }
};
```

```
main()
{
        Unit *p;
        p = new Distance(5, 2);
        p->showdata();
        p->convert();
        delete p;

        Unit *q;
        q = new Time(4, 10);
        q->showdata();
        q->convert();
        delete q;
}
```

Output:
5 Feet 2 Inches
Data is 62 inches
4 Hours 10 Minutes
Data is 250 minutes

Example-5: To generate polymorphic code

```
main()
{
        Unit *p[5];
        p[0] = new Distance(5,2);
        p[1] = new Time(10,5);
```

Output:
5 Feet 2 Inches
Data is 62 inches

10 Hours 5 Minutes

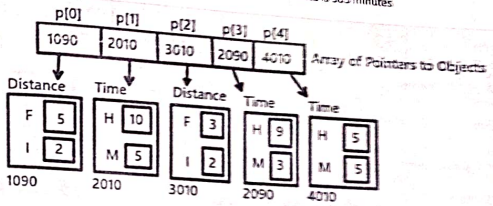

```
        p[2] = new Distance(3,2);
        p[3] = new Time(9,3);
        p[4] = new Time(5,5);

        for(int i=0;i<=4;i++)
        {
                p[i]->showdata();
                p[i]->convert();
                delete p[i];
        }
}
```

Data is 605 minutes

3 Feet 2 Inches
Data is 38 inches

9 Hours 3 Minutes
Data is 543 minutes

5 Hours 5 Minutes
Data is 305 minutes



Container classes

(| Containership | Object Composition | Collection | Nested Classes | Member classes | Aggregation)

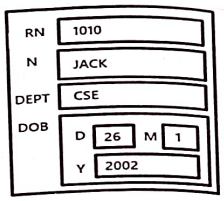
- If a class contains object as data_member (i.e. object of another class as data member) then such a mechanism is called as containership..
- Containership represents HAS-A relationship

Example: Design a class student with data-members RollNo, Name, Dept & DOB. Define member functions readdata() & showdata(). To store DOB, design a class Date with data members Day, Month & Year. Define member functions readdata() & showdata().

```
#include<iostream>
using namespace std;
class Date
{
        private:
                int D, M, Y;
        public:
        void readdata()
        {
        //cout<<"Enter Date (DD/MM/YYYY): ";
        cin>>D;
        cin.ignore();
        cin>>M;
        cin.ignore();
        cin>>Y;
        }

        void showdata()
        {
        cout<<D<<"."<<M<<"."<<Y<<endl;
        }
};
```

```
class Student
{
        private:
                int RN; char N[20], DEPT[20]; Date DOB;
        public:
        void readdata()
        {
        cout<<"\nEnter Roll No. Name & Dept: ";
        cin>>RN>>N>>DEPT;
        cout<<"Enter DOB (DD/MM/YYYY): ";
        DOB.readdata();
        }

        void showdata()
        {
        cout<<"\nRoll Number = "<<RN<<endl;
        cout<<"Student Name = "<<N<<endl;
        cout<<"Department = "<<DEPT<<endl;
        cout<<"DOB = ";
        DOB.showdata();
        }
};
```

```
main()
{
        Student a;
        a.readdata();
        a.showdata();
}
```



Output:
Enter Roll No. Name & Dept: 1010 JACK CSE <Enter>
Enter DOB (DD/MM/YYYY): 26/1/2002 <Enter>

Roll Number = 1010
Student Name = JACK
Department = CSE
DOB = 26.1.2002

IS-A Relationship and HAS-A Relationship

- There are two ways we can use to reuse the code as follows.
  1. Inheritance (IS-A relationship)
  2. Object composition (HAS-A relationship).

IS-A Relationship

- Inheritance represents IS-A type of relationship.
- Inheritance is uni-directional towards base class.
- For example:
  Suppose we have a base class Person and two childs of Person are class Student and class Employee. Then we can say that every Student is a Person but every Person is not a Student.

HAS-A Relationship

- Containership represents HAS-A relationship.
- It indicates that an object contains another object/s in it.
- Example: class Student in above example of Container classes.

Namespace

```
#include<iostream>
using namespace std;
namespace n1
{
        int x = 10;
}
namespace n2
{
        int x = 20;
}
main()
{
        cout<<"n1::x = "<<n1 :: x<<endl;
        cout<<"n2::x = "<<n2 :: x<<endl;
}
```

Output:
n1::x = 10
n2::x = 20

Exception Handling

- The exception means run-time error.
- If the execptions are not handelled then the program causes abnormal termination.
- That means the program stops its execution at the point where the run-time error was occured.
- The aim of exception handling is that, if any runtime error occured then the program must not stop its execution, rather the program must handle the runtime error and must continue the execution of remaining program.
- The C++ Exception handling mechanism is consist of following elements:
  1. try block
  2. throw statement
  3. catch block

try block

- The code whose execution may generate runtime error, such code can be write inside the try block.
- If any runtime error is detected then the programmer can throw a value or object to represent that runtime error.
- After throwing the value or object, the program control is transfered in the appropriate catch block where the runtime error gets handelled.
- **Syntax:**

```
try
{
        code which may cause runtime error
        ...
        ...
}
```

throw statement

- The throw statement is used to transfers the program control from try block to appropriate catch block.
- With the throw statement we need to specify a value or object to indicate the type of runtime error occurred.
- As per the value (i.e. error number) or object thrown an appropriate catch block gets executed.
- If the value or object thrown is not matched with any catch block then the default catch block gets executed.
- **Syntax:**
  1. **throw object;**
  2. **throw value;**

catch block

- The code to handle the runtime error can be write in the catch block.
- After a try block we can specify multiple catch blocks to handle different types of exceptions.
- As per the value or object thrown an appropriate catch block gets executed.
- If the value or object thrown is not matched with any catch block then the default catch block gets executed.
- The default catch block is optional.
- **Syntax:**

General catch block

```
catch(datatype arg)
{
        statements
        ....
        ....
}
```

Default catch block

```
catch(...)
{
        statements
        ....
        ....
}
```

Note: The default catch block have only 3 dots in the round brackets.

Syntax:

```
try
{
        ....
        throw value;
        ....
}
catch(data_type var)
{
        ....
        ....
}
catch(...)              // default catch block
{
        ....
        ....
}
```

Example: Write a program to read two integers and find their quotient. Compute result with necessary privileges.

```
#include<iostream>
using namespace std;
main()
{
        try
        {
                int a, b;
                cout<<"Enter 2 integers: ";
                cin>>a>>b;

                if(b == 0)
                throw 0;

                float c = a / b;
                cout<<"Quotient is "<<c<<endl;
        }
        catch(int ex)
        {
                if(ex == 0)
                cout<<"Cannot divide by zero"<<endl;
        }
        catch(...)
        {
                cout<<"This is default catch block"<<endl;
        }

        int x=10, y=20;
        int z = x + y;
        cout<<"Sum is "<<z<<endl;
}
```

Output-1:
```
Enter 2 integers: 10   0 <Enter>
Cannot divide by zero
Sum is 30
```

Output-2:
```
Enter 2 integers: 10   2 <Enter>
Quotient is 5
Sum is 30
```

Templates

- Templates are used to define **shareable** functions and classes for builtin|generic data-types in C++.
- If we want to perform same or common operation on values of different datatypes (like int, float, char etc.) then instead of writing the same logic again and again we can define a function template or template class. That means when the logic is same but only the datatype is different then we can define function template or class template. For example sort(), max(), min(), search(), etc.
- The advantage of templates is that once the function template or template class is defined then the same function or class will work for different data-types.
- The template definition is automatically replaced as per the data type specified.
- A template can be defined by using following syntax:
  Syntax: **template<class name1, class name2, ... >**
- The template is an example of polymorphism as the same template function or template class can work for different datatypes.

Function Template

- A function can be define by using template as follows:
- **Syntax:**

```
template<class template_name>
return_type function_name(template_name arg1, template_name arg2, ...)
{
    ...
    ...
    return (value) ;
}
```


- **Syntax to call function:**
  var = function_name <builtin_data_type> (val1, val2, ...);

Example: Design a function template smallest() which will return smallest of 2 chars or integers, or double values.

```
#include<iostream>
using namespace std;
template<class new_datatype>
new_datatype smallest(new_datatype a, new_datatype b)
{
        if(a < b)
                return(a);
        else
                return(b);
}
main()
{
char x = smallest<char>('P', 'Q');
cout<<"Smallest is "<<x<<endl;

int y = smallest<int>(20, 10);
cout<<"Smallest is "<<y<<endl;

double z = smallest<double>(1.1, 2.2);
cout<<"Smallest is "<<z<<endl;
}
```

Output:
```
Smallest is P
Smallest is 10
Smallest is 1.1
```

Example:

```
#include<iostream>
using namespace std;
template<class mydatatype>
void print_data(mydatatype x[])
{
        for(int i=0; i<=2; i++)
        cout<<x[i]<<"\t";
        cout<<endl;
}
main()
{
        int a[] = {10, 20, 30};
        print_data(a);

        float b[] = {1.1, 2.2, 3.3};
        print_data (b);

        char c[] = {'A', 'B', 'C'};
        print_data (c);
}
```

Output:
```
10    20    30
1.1   2.2   3.3
A     B     C
```

Example: Default values. Design a function sum() which can be called by passing 3 or 2 values.

```
#include<iostream>
using namespace std;
template<class mydatatype>
mydatatype sum(mydatatype a, mydatatype b, mydatatype c=0)
{
        mydatatype s = a + b + c;
        return(s);
}
main()
{ int x = sum<int>(10, 20, 30);
cout<<"x = "<<x<<endl;

double p = 1.1, q = 2.2, r = 3.3;
double y = sum<double>(p, q, r);
cout<<"y = "<<y<<endl;

x = sum<int>(40, 50);
cout<<"x = "<<x<<endl;

y = sum<double>(4.4, 5.5);
cout<<"y = "<<y<<endl;
}
```

Output:
```
x = 60
y = 6.6
x = 90
y = 9.9
```

Class Template

- The class which is specially designed for generic datatypes, that means whose data-members are of template type also the member function are function template, such a class is called as class template.

- **Syntax:**

```
template<class template_name>
class class_name
{
    template_datamembers;
    ....
    template_functions;
    ....
};
```

Creating an object of class template

- An object of class template can be created as follows.
- **Syntax:**
  class_name <generic_data_type> object_name(val1, val2, ...);

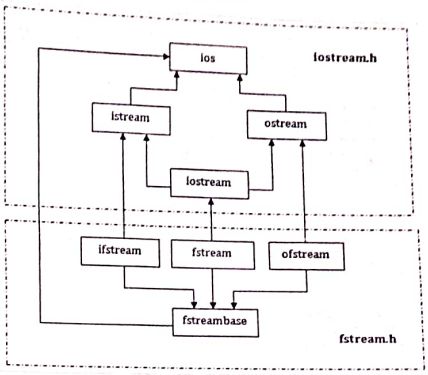
Example: Design a template class Math having data_members X & Y and member functions max(), min() and sum() whose objects can be created by passing data.

```
#include<iostream>
using namespace std;
template<class mydatatype >
class Maths
{
        private:
                mydatatype X,Y;
        public:
        Maths(mydatatype x, mydatatype y)
        {
                X=x;
                Y=y;
        }

        mydatatype max()
        {
                if(X>Y)
                return(X);
                else
                return(Y);
        }

        mydatatype min()
        {
                if(X<Y)
                return(X);
                else
                return(Y);
        }
        mydatatype sum()
        {
                mydatatype S = X + Y;
                return(S);
        }
};

main()
{
Maths <int> a(5,9);
cout<<"\nGreatest = "<<a.max()<<endl;
cout<<"Lowest = "<<a.min()<<endl;
cout<<"Sum = "<<a.sum()<<endl;

Maths <float> b(1.1,10.10);
cout<<"\nGreatest = "<<b.max()<<endl;
cout<<"Lowest = "<<b.min()<<endl;
cout<<"Sum = "<<b.sum()<<endl;

Maths <char> c('A','B');
cout<<"\nGreatest = "<<c.max()<<endl;
cout<<"Lowest = "<<c.min()<<endl;
}
```

Output:

```
Greatest = 9
Lowest = 5
Sum = 14

Greatest = 10.1
Lowest = 1.1
Sum = 11.2

Greatest = B
Lowest = A
```


Stream classes hierarchy



Console I/O

Functions of class ios

- The class ios provides us different functions to display data in different format on console.
- As this functions displays data on console and class ostream is inherited from class ios and is interested to display data on console so, these functions can be used with our standard output object cout of class ostream.
- Some of these functions are as follows.

1. precision(int n)
It displays the specified number of digits after the decimal point(for floating numbers only).

Example:

```
#include<iostream.h>
main()
{
        float z = 3.145611;
        cout<<"z = "<<z<<endl;
        cout.precision(2);
        cout<<"z = "<<z<<endl;
}
```

Output:

```
z = 3.145611
z = 3.15
```

2. int width(int w)

- It defines a block of specified width and manages the output in the specified block.

3. char fill(char c)
This function fills the blank spaces of width function by the specified character.

Example:

```
#include<iostream.h>
main()
{
        float z = 3.1416;
        cout.width(10);
        cout<<z<<endl;
        cout.width(10);
        cout.fill('#');
        cout<<z<<endl;
}
```

Output:

```
      3.1416
######3.1416
```

4. setf(int flag)

- This function sets the different formatting flags according to which the data will be displayed. Multiple flags can be set by separating binary OR (|)operator.

a) **ios :: dec** -Displays the integer values in decimal format.
b) **ios :: oct** - Displays the integer values in octal format.
c) **ios :: hex** -Displays the integer values in hexadecimal format.

Example:

```
#include<iostream.h>
#include<conio.h>
main()
{
        clrscr();
        int z=10;
        cout<<"z = "<<z<<endl;
        cout.setf(ios::hex);
        cout<<"z = "<<z<<endl;
        cout.setf(ios::oct);
        cout<<"z = "<<z<<endl;
        cout.setf(ios::dec);
        cout<<"z = "<<z<<endl;
        getch();
}
```

Output:

```
z = 10
z = a
z = 12
z = 10
```

d) ios :: scientific

- It displays the floating point number in the scientific format.

e) ios::uppercase

- It displays the character of hexadecimal & scientific format in the block letters.

f) ios::showpos

Show '+' with positive number.

g) ios::internal

- Displays + sign before fill character for positive numbers.

Example:

```
#include<iostream.h>
#include<conio.h>
main()
{
        clrscr();
        float z=3.1416;
        cout.width(15);
        cout.fill('#');
        cout.setf(ios::scientific | ios::showpos | ios::uppercase);
        cout<<z<<endl;

        cout.width(15);
        cout.fill('#');
        cout.setf(ios::internal);
        cout<<z<<endl;
        getch();
}
```

Output:

#	#	#	#	+	3	.	1	4	1	6	E	+	0	0

+	#	#	#	#	3	.	1	4	1	6	E	+	0	0

Example:

```
#include<iostream.h>
#include<conio.h>
main()
{
        clrscr();
        int z=10;
        cout<<"z = "<<z<<endl;
        cout.setf(ios::hex|ios::uppercase);
        cout<<"z = "<<z<<endl;
        getch();
}
```

Output:

```
z = 10
z = A
```

h) ios::showpoint

Shows the decimal point for the floating-point output though not necessary.

Example:

```
#include<iostream.h>
#include<conio.h>
main()
{
        clrscr();
        float z = 3;
        cout<<"z = "<<z<<endl;
        cout.width(15);
        cout.fill('#');
        cout.setf(ios::showpos | ios::showpoint);
        cout<<z<<endl;
        getch();
}
```

Output:

```
z = 3
```

#	#	#	#	#	#	+	3	.	0	0	0	0	0

i) ios::right

It displays the output right aligned.

j) ios::left

It displays the output left aligned.

Example:

```
#include<iostream.h>
#include<conio.h>
main()
{
        clrscr();
        char N[50]="KUSHAL";
        cout.width(10);
        cout.fill('#');
        cout.setf(ios::right);
        cout<<N<<endl;

        cout.width(10);
        cout.fill('#');
        cout.setf(ios::left);
        cout<<N<<endl;
        getch();
}
```

Output:

#	#	#	#	K	U	S	H	A	L

K	U	S	H	A	L	#	#	#	#

Manipulators

- Manipulators are the overloaded insertion operators.
- It is the alternate way for some function | flags of class ios.
- These manipulators are defined in the header file <iomanip.h>.
- Some of these manipulators are as follows.

Manipulator	Description
setw(int w)	It defines a block of specified width and manages the output in the specified block. It is equivalent to int width(int).
setfill(char c)	This function fills the blank spaces of width function by the specified character. It is equivalent to fill(char c).
setprecision(int n)	It displays the specified number of digits after the decimal point (for floating point number only). It is equivalent to precision(int n).
dec	Displays the integer value in the decimal format.
oct	Displays the integer value in the octal format.
hex	Displays the integer value in the hexadecimal format.
setiosflags(long flag)	It sets the ios flags of setf() function to display data in different format.
resetiosflags(long flag)	It clears the specified ios flag.
endl	It inserts a new line & flushes the stream.

Example:
```
#include<iostream>
#include<iomanip>
using namespace std;
main()
{
        int z=10;
        cout<<"z = "<<z<<endl;
        cout<<hex<<setiosflags(ios::uppercase)<<"z = "<<z<<endl;
        cout<<oct<<"z = "<<z<<endl;
        cout<<dec<<"z = "<<z<<endl;
}
```

Output:
```
z = 10
z = A
z = 12
z = 10
```

Example:
```
#include<iostream>
#include<iomanip>
using namespace std;
main()
{
        float z = 3.1416;
        cout<<setw(15)<<setfill('#')<<z<<endl;
        cout<<setw(15)<<setfill('#')<<setiosflags(ios::scientific)<<z<<endl;
        cout<<setprecision(2)<<z<<endl;
        cout<<resetiosflags(ios::scientific)<<z<<endl;
}
```

Output:
```
########3.1416
#####3.1416e+00
3.14e+00
3.14
```

File Handling

- A file is used to store general or business information.
- C++ provides us different predefined classes to perform operations on files.
- These classes are class ifstream, class ofstream and class fstream.
- All these classes are available in the header file <fstream.h>.
- There are two types of data storage and retrieval approaches as follows:

Sequential file access	▪ In sequential file access we can write or read data in a linear sequence. ▪ That means without writing/ reading 1st char or line we cannot write / read 2nd char or line.
Random file access	▪ Writing and reading fixed length data in a file is called as Random file access. ▪ In random file access we can access the data randomly i.e. we can access any record either first or middle or last.

class ofstream

- An object of this class is used to create a new file and write data into it.
- This class is available in the header file <fstream.h>

Constructors:

Constructor	Description
ofstream(char* file_name)	▪ It create a new file with specified name to write data into the file. ▪ It will return true if the file created and opened successfully otherwise it will return false.

Member Functions:

Function	Description
void open(char* file_name)	It create a new file to write data into the file.
void close()	It closes the file opened by the object.
int put(int c)	It will write the given character in the file.

class ifstream

- An object of this class is used to open and read data from an existing file.
- This class is available in the header file <fstream.h>

Constructors:

Constructor	Description
ifstream(char* file_name)	▪ It opens an existing file to read data from it. ▪ It will return true if the file opened successfully otherwise it will return false.

Member functions:

Member Function	Description
void open(char* file_name)	It opens an existing file to read data from it.
void close()	It closes the file opened by the object
int get()	▪ This function reads a single character from a file and return its int value. ▪ It always read the next character. ▪ This function returns EOF (-1) at the end of file.

Sequential File Access System

Example: IMP
WAP to create a new file and write a string into it.
```
#include<iostream>
#include<fstream>
using namespace std;
main()
{
        ofstream f("C:/my_folder/sample.txt");
        if(f)
        {
        f<<"C++"<<endl;
        f<<"Object Oriented Programming"<<endl;
        f.close();
        cout<<"Task completed.."<<endl;
        }
        else
        cout<<"file not created.."<<endl;
}
```

Example:
WAP to read lines from a file
```
#include<fstream>
#include<iostream>
using namespace std;
main()
{
        ifstream f1("C:/my_folder/sample.txt");
        char line1[50], line2[50];
        if(f1)
        {
        f1.getline(line1, 49);
        // f1>>line2;
        f1.getline(line2, 49);
        cout<<line1<<endl;
        cout<<line2<<endl;
        f1.close();
        }
        else
        cout<<"File not exist.."<<endl;
}
```
Output:
```
C++
Object Oriented Programming
```

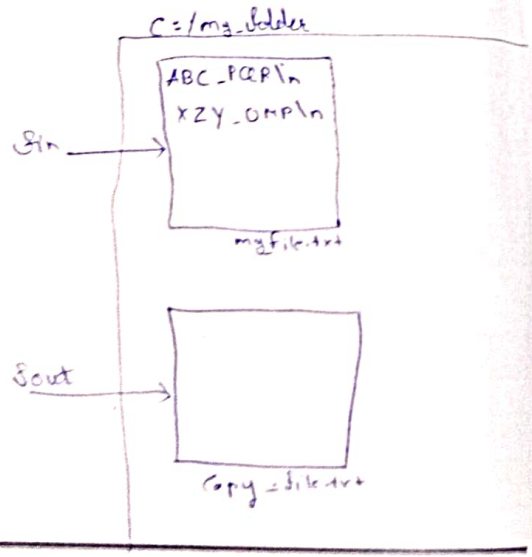
IMP

Example: WAP to display contents of a file. Also count total number of characters, words and lines in the file. Also display the file size in bytes.
```
#include<fstream>
#include<iostream>
using namespace std;
main()
{
        ifstream f1("C:/my_folder/myfile.txt");

        if(f1)
        {
                // blank space = 32
                // \n = 10
```

```
        int chars = 0, words = 0, lines = 0;

        while(1)
        {
                int ch = f1.get();
                if(ch == -1)
                break;

                cout<<(char)ch;
                chars++;

                if(ch == 32 || ch == 10)
                words++;

                if(ch == 10)
                lines++;
        }
        f1.close();

        cout<<"\n";
        cout<<"Number of chars approx = "<<chars<<endl;
        cout<<"Number of words approx. = "<<words<<endl;
        cout<<"Number of lines approx. = "<<lines<<endl;
        cout<<"File Size = "<<chars<<" bytes"<<endl;
    }
    else
    cout<<"File not exist.."<<endl;
}
```

Example: Write a program to copy content of the file "C:/my_folder/myfile.txt" into a new file "C:/my_folder/copy.txt".

```
#include<iostream>
#include<fstream>
using namespace std;
main()
{
        ifstream f1("C:/my_folder/myfile.txt");
        ofstream f2("C:/my_folder/copy.txt");

        if(f1 && f2)
        {
                int ch;

                while(1)
                {
                        ch = f1.get();

                        if(ch == EOF)
                        break;

                        f2.put(ch);
                }

                f1.close();
                f2.close();
                cout<<"File Copied.."<<endl;
        }
```




```
        else
        cout<<"Error: Task not completed.."<<endl;
}
```

class fstream

- The class fstream provides features to both read and write data into a file.
- This class is available in the header file <fstream.h>

Constructors:

Constructor	Description
fstream(char *file_name, int mode)	It will opens the specified file in specified mode.

Member Functions:

Sr.No.	Member Function and description
1	**void open(char *file_path, int mode);** ▪ This function opens the specified file in specified mode.The different modes are present in the following table. **Modes** / **Description** ios :: in — It will open the file to read data. ios :: out — It will create a new file to write data into it. ios :: app — ▪ This is append mode. ▪ It will add data to end of a file if the file is already present. ▪ If the file is not exist then it will create a new file. ios :: binary — if we want to write or read fixed length data from the file then the file must be open in binary mode. (For Random Access File System Only) **Note:** ▪ Multiple modes can be separated by using '|' (i.e. pipe) operator ▪ **Example: object.open(char *file_name, ios::app | ios::binary);**
2	**void close()** This function closes the opened file.
3	**write((char*)&object_name, int object_size);** **write((char*)&variable_name, int value_size);** ▪ This function writes fixed length data into a file. ▪ While writing or reading fixed length data the file should be open in binary mode ▪ To find memory size of datatype or object we can use the sizeof() operator
4	**read((char*)&object_name, int object_size);** **read((char*)&variable_name, int size);** ▪ This function reads fixed length data from a file and store it in the given object or variable. ▪ At the end of file it will return false. ▪ To find memory size of datatype or object we can use the sizeof() operator

5 **seekg(int move_num_bytes, int given_position);**

- It moves the get pointer/file cursor while reading data from a file to the specified position.
- To move cursor in forward direction the bytes value must be positive and to move cursor in reverse direction the bytes value must be negative.
- This function will work for random access file system only.
- The GET pointer/cursor exist when the file is opened to read data.
- The PUT pointer/cursor exist when the file is opened to write or append data.

Where:

move_num_bytes	It indicates the number of bytes the pointer to be moved from the given position.
given_position	• It indicates the position from which the pointer to be moved. • The standard position of file cursor/pointer can be any of the following.

ios::beg	Indicates the start of the file
ios::cur	Indicates current position of the pointer/cursor
ios::end	Indicates the end of the file

This function has one of the following form.

Form	Meaning
seekg(0, ios :: beg)	Go to start
seekg(0, ios :: cur)	Stay at the current position
seekg(0, ios :: end)	Go to the end of file
seekg(n, ios :: beg)	Move to (n+1)th byte in the file from the beginning
seekg(n, ios :: cur)	Go in forward direction by n bytes
seekg(-n, ios :: cur)	Go in reverse direction by n bytes
seekg(-n, ios :: end)	Go in reverse direction by n bytes

6 **seekp(int move_num_bytes, int given_position);**

- It will move the put pointer to the specified position from the given position while writing data into a file.
- It will work for random access file system only.
- The given_position is same as above.

7 **int tellg();**

- It returns the current position to the get pointer.
- It will work for ranuom access file system only.

8 **int t-llp(**

- It returns the current position to the put pointer.
- It will work for random access file system only.

9 **int get()**

- This function reads a single character from a file.
- It will always read next character from the file.
- This function returns EOF (-1) at the end of file.
- It will work for sequential access file system only.

10 **int put(int c)**

- Writes the given character in the file.
- It will work for sequential access file system only.

IMP

Example: Write a program to copy contents of the file "C:/my_folder/myfile.txt" into a new file "C:/ my_folder /copy.txt" by using class fstream.

```
#include<fstream>
#include<iostream>
using namespace std;
main()
{
        char *source_file = "c:/ my_folder/sample.txt";
        char *dest_file = "c:/ my_folder/copy.txt";
```


(.dat)


```
        fstream fin, fout;
        fin.open(source_file, ios::in);
        fout.open(dest_file, ios::out);

        if(fin && fout)
        {
                while(1)
                {
                        int ch = fin.get();
                        if(ch == -1)
                        break;
                        else
                        fout.put(ch);
                }
                fin.close();
                fout.close();
                cout<<"Task completed.."<<endl;
        }
        else
        cout<<"Task not completed.."<<endl;
}
```

File Inclusion | Multi-File Program

- If we want to define the same functions, structures, unions, enums, classes, template classes, etc. in different programs then instead of defining them again and again, we can write them only once in a file and can include that file in different programs.
- It will avoid the retyping and repetition of code and will save the coding time also.
- In big projects, writing the code in different files keeps the project more organized.
- Also, it is easy to find the code file if we want to make any change.
- Also, it will speed up the compile time. Suppose, if everything defined or written in a single file, and if we make any change in the code then the complete file will be recompiled. That means, the code which is not changed that code will also recompiled and it will increase the compile time. If the code is written in different files then after making changes, only one small file will be compiled and it will reduce the compile time.
- A C++ file can be included in another C++ program as follows.
- **Syntax:**
  1. **#include "file_path"**
  2. **#include<file_path>**

Example:

Steps:

1. Create and save "Employee.cpp"
2. Compile the file "Employee.cpp"
3. Create and save "Sample.cpp"
4. Compile and run "Sample.cpp"

Employee.cpp

```
#include<iostream.h>
class Employee
{
        private
                int empno;
                char ename[20], job[20];
                float sal;
        public:
        void readdata()
        {
```

Sample.cpp

```
#include<iostream.h>
#include "Employee.cpp"
main()
{
        Employee a;
        a.readdata();
        a.showdata();
}
```

Output:

```
        cout<<"\nEnter Emp. No., name, job & salary: ";
        cin>>empno>>ename>>job>>sal;
        }

        void showdata()
        {
        cout<<"\nEmp. Number = "<<empno<<endl;
        cout<<"Emp. Name = "<<ename<<endl;
        cout<<"Job = "<<job<<endl;
        cout<<"Salary = "<<sal<<" Rs."<<endl;
        }

        int get_empno()
        {
        return(empno);
        }
};
```

```
Enter Emp. No., name, job & salary: 1010 JACK
ENGINEER 30000 <Enter>

Employee Number = 1010
EmployeeName = JACK
Job = ENGINEER
Salary = 30000 Rs.
```

Random Access File System

- While writing or reading fixed length data (i.e. Block I/O) the file must be open in the binary mode (ios :: binary)

Example: Write a program (WAP) to save the objects of class Employee in a file "mydatabase.dat".
Note: The file "mydatabase.dat" will be created in the current "bin" directory.

```
#include<iostream.h>
#include<fstream.h>
#include "Employee.cpp"
main()
{
        fstream f;
        f.open("mydatabase.dat", ios::app|ios::binary);

        if(f)
        {
                Employee a;
                a.readdata();
                f.write((char*)&a, sizeof(a));
                f.close();
                cout<<"Record Saved.."<<endl;
        }
        else
        cout<<"Error: File not found.."<<endl;
}
```

Example: WAP to display all the record from the file "Employee.dat".

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
#include"Employee.cpp"
main()
{
        fstream f;
        f.open("mydatabase.dat", ios::in|ios::binary);
        if(f)
        {
                Employee a;
                while(f.read((char*)&a, sizeof(a)))
```



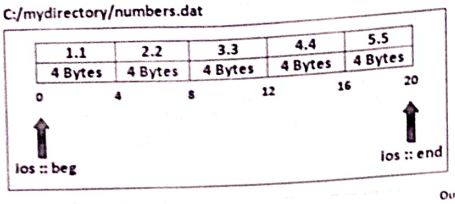
```
                a.showdata();
                f.close();
        }
        else
        cout<<"Error: File not found.."<<endl;
}
```

Example: Write a program to read employee number and display the information of that employee if its record is available in the file "mydatabase.dat" otherwise display appropriate message.

```
#include<iostream.h>
#include<fstream.h>
#include"Employee.cpp"
main()
{
        int eno;
        cout<<"Enter Employee Number: ";
        cin>>eno;

        fstream f;
        f.open("mydatabase.dat", ios::in|ios::binary);

        if(f)
        {
                int c = 0;
                Employee a;
                while(f.read((char*)&a,sizeof(a)))
                {
                        if(eno == a.get_empno())
                        {
                                a.showdata();
                                c++;
                                break;
                        }
                }
                f.close(),

                if(c == 0)
                cout<<"Record not found.."<<endl;
        }
        else
        cout<<"Error: File not found.."<<endl;
}
```

Example: WAP to store 5 float values in a file "C:/mydirectory/numbers.dat" and display position of put pointer after writing each value.

C:/mydirectory/numbers.dat




```
#include<iostream>
#include<fstream>
using namespace std;
main()
{
fstream f1("C:/mydirectory/numbers.dat", ios::out|ios::binary);
        if(f1)
        {
                float a[] = {1.1, 2.2, 3.3, 4.4, 5.5};

                cout<<"Put Pointer Position = "<<f1.tellp()<<endl;
                for(int i=0; i<=4; i++)
                {
                f1.write((char*)&a[i], sizeof(float));
                cout<<"Put Pointer Position = "<<f1.tellp()<<endl;
                }
                f1.close();
                cout<<"Numbers saved.."<<endl;
        }
        else
        cout<<"File not exist"<<endl;
}
```

Output:

```
Put Pointer Position = 0
Put Pointer Position = 4
Put Pointer Position = 8
Put Pointer Position = 12
Put Pointer Position = 16
Put Pointer Position = 20
Numbers saved..
```

Example: WAP to display all the numbers from the file "C:/mydirectory/numbers.dat" also display the position of get pointer after reading every value. Also find sum and mean of all the numbers.

```
#include<iostream>
#include<fstream>
using namespace std;
main()
{
        fstream f1("C:/mydirectory/numbers.dat", ios::in|ios::binary);
        if(f1)
        {
                float n, s=0;
                cout<<"Get Pointer Position = "<<f1.tellg()<<endl;

                while(f1.read((char*)&n, sizeof(float)))
                {
                cout<<"n = "<<n<<endl;
                cout<<"Get Pointer Position = "<<f1.tellg()<<endl;
                s = s + n;
                }
                f1.close();
```

Output:

```
Get Pointer Position = 0
n = 1.1
Get Pointer Position = 4
n = 2.2
Get Pointer Position = 8
n = 3.3
Get Pointer Position = 12
n = 4.4
Get Pointer Position = 16
n = 5.5
Get Pointer Position = 20
Sum is 16.5
Mean is 3.3
```



```
                cout<<"Sum is "<<s<<endl;
                float m = s/5.0;
                cout<<"Mean is "<<m<<endl;
        }
        else
        cout<<"File not exist"<<endl;
}
```

Example: WAP to display 2nd and 4th and 1st number from the file C:/mydirectory/numbers.dat

```
#include<iostream>
#include<fstream>
using namespace std;
main()
{
        fstream fin;
        fin.open("C:/mydirectory/numbers.dat", ios::in|ios::binary);
        if(fin)
        {
                float n;
                fin.seekg(4, ios::beg);
                fin.read((char*)&n, sizeof(float));
                cout<<"n = "<<n<<endl;

                fin.seekg(-8, ios::end);
                fin.read((char*)&n, sizeof(float));
                cout<<"n = "<<n<<endl;

                fin.seekg(-16, ios::cur); // fin.seekg(0, ios::beg);
                fin.read((char*)&n, sizeof(float));
                cout<<"n = "<<n<<endl;
                fin.close();
        }
        else
        cout<<"File not found.."<<endl;
}
```

Output:

```
n = 2.2
n = 4.4
n = 1.1
```