C++

1. Scope Resolution Operator

We cannot access the global variable inside the function. Scope resolution operator is used to solve this problem

Example

```
#include<iostream.h>
#include<conio.h>
int m=30; //global variable
main(){
int m=10; //local variable
cout<<"Local m=  "<<m;
cout<<"Global m= "<<::m;
::m=50; //reinitialize global variable
cout<<"Local Variable = "<<m;
cout<<"Global variable = "<<::m;
}
```

Output

```
Local m = 10
Global m = 30
Local m =10
Global m =50
```

2. Passing Member \ Accessing Functions outside the class using scope resolution operator

Member function can be defined outside the class using the scope resolution operator.

Example

```
#include <iostream.h>
#include<conio.h>
class distance{
```

```
private:
int feet;
float inches;
public:
distance(){
float=0;
inches=0.0;
}
void readdata(){
cout<<"enter feet & inches ";
cin>>feet>>inches;
}
void showdata();
};
void distance::showdata(){
cout<<" feet = "<<feet;
cout<<" inches = "<<inches;
}

int main(){
distance d1;
d1.readdata();
d1.showdata();
getch();
}
```

3. Reference Variable

Reference variable provides an alternative name for the previous defined variable

For Example If we make the variable "sum" a reference to the "total" then both "sum" and "total" can be used interchangeably to represent that variable

```
float total = 100;

float &sum = total;
```

Dynamic allocation using reference variable

```
#include<iostream.h>
#include<conio.h>
void main(){
int *arr;
int size;
cout<<"Enter size of array";
cin>>size;
arr=new int [size];
cout<<"Dynamic allocation done";
delete arr;
getch();
}
```

Output

```
Enter size of array : 5
Dynamic allocation done
```

4. C++ Manipulators

These are the operators available in c++ which are used to format the display of o/p.

1. End of line : cout<<"hello"endl<<"hy";
2. Set w(d) : sets the width of display
   Eg : int x= 20
   Coout<<x;
   _ _ _ _ _ _ _ _ 20
   Variable value will be right justify

5. Memory Management Operator

C++ defines two unary operators new and delete that perform the task of allocating & freeing the memory in a better & easier way. These are also known as free store operators.

The new operator can be used to create objects of any type. It allocates sufficient memory to hold a data object.

Example

```
int *p = new int;
float *q =new float;
*p = 25;
*q = 7.5;
```

We also initialize the memory using new operator

```
pointer-variable = new datatype(value);
Int *p = new int(25);
Float *q = new float(7.5);
```

New can also be used to create a memery space for any datatype including user defined datatype such as array.

```
pointer-variable = new datatype
Int *p = new int[10];
```

Example

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

int main(){
int *arr;
int size;
clrscr();
cout<<"Enter the size of array ";
cin>>size;
arr = new int [size];
cout<<"\nDynamic allocation done \n";

for (int i = 0; i < size; i++)
```

```
{
cout<<"Enter no ";
cin>>arr[i];
}

for(i=0;i<size;i++){
cout<<arr[i]<<endl;
}

delete arr;
getch();
}
```

Delete Operator : When a data objects is no longer needed, it is destroyed to release the memory space for reuse the general form of its use:

delete pointer-variable

delete p;

if we want to free a dynamically allocated array

delete [size] pointer-variable;

delete [10] p;

6. Escape sequences

These are the sequences which are used to change the format of the o/p on the screen.

“\n”	New line
“\t”	Tap
“\b”	Backspace
“\a”	Bell (Beep sound)
“\’”	For single code
“\””	For double codes

7. Enumerated datatype

An enumerated datatype is anther user defined data type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code.

The enum keyword automatically enumerates a list of words by assigning them values 0,1,2,3 & soon.

```
enum{
circle; // auto value 0
rectangle; //1
triangle; //2
}
```

Example

```
#include <iostream.h>
#include<conio.h>
enum shape{
circle;
rectangle;
triangle;
}
main(){
int code;
cout<<"Enter shape code : ";
cin>>code;
while(code>=circle&&code<=triangle)
{
switch(code){
case circle : cout<<"hello";
case rectangle : cout<<"hi";
case triangle : cout<<"good";
break;
```

```
}
break;
}
cout<<"bye";
getch();
}
```

Output

```
Enter shape code : 1
Hibye
Enter shape code : 0
Hello bye
Enter shape code : 2
Good bye
```

8. Structure

It is user defined datatype composed of data items that may be of different datatypes

The size of structure type is equal to the sum of the sizes of the individual member types.

It is defined by using the keyword struct.

```
Struct structure_name{
Datatype variable1;
Datatye vartiable2;
};
```

Accessing structure members can be accessed by creating a structure variable & using dot. Operator.

Employee e; e.age;	'Employee' is structure name, 'e' is variable. 'age' is member name.

Example

```
#include <iostream.h>
#include<conio.h>
main(){
struct Employee{
char name[20];
int age;
float salary;
};
Employee e;
cout<<"Enter name : ";
cin>>e.name;
cout<<"Enter age : ";
cin>>e.age;
cout<<"Enter salary : ";
cin>>e.salary;

cout<<"\n Your entered :- \n";
cout<<e.name<<endl;
cout<<e.age<<endl;
cout<<e.salary;
}
```

Structure can also be initialized when it is defined

Employee e = {"Amit",27,26000}

9. Union

Union is user defined datatypes they are similar to structure as they allow to group together dissimilar type elements inside a single unit.

Size of union is equal to the size of its largest member element

struct / union{ **char name[20];** **int age;** **float salary;**	Size of structure is : 20+2+4 = 26	Size of union is : Largest size value = 20

}		

10. char* datatype

```
#include<iostream.h>
#include<conio.h>
int main(){
char* c;
char b;
cin>>c;
cin>>b;

cout<<c<<endl<<b;
getch();
}
```

Output

```
Mohit
Kumar
Mohit
K
```

Concept of OOPS

1. Object

An object can be defined as an instance of a class, and there can be multiple instances of an class in a program. An Object contains both the data(member) and the function(member function). For example - chair, bike, marker, pen, table, car, etc.

2. Class

The class is a group of similar entities. It is only an logical component and not the physical entity. For example,

if you had a class called “Expensive Cars” it could have objects like Mercedes, BMW, Toyota, etc.

‘Collection of objects is called class. It is a logical entity.’

3. Data encapsulation

The wrapping up of data and function into a single unit(called class) is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.

4. Data Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.
A java program is also a great example of abstraction. In java, we use abstract class and interface to achieve abstraction.

5. Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Base class : - The class whose features (qualities) are inherited by another class is called base class. Eg. Person

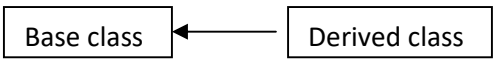
Derived class : The class which inherits the features of another class is called derived class. Eg. employee.

There are various type of Inheritance

(i) Single inheritance
(ii) Multiple inheritance

(iii) Multilevel inheritance
(iv) Hierarchical Inheritance

Single Inheritance : The situation in which a derived class has only one base class.




```
#include<iostream.h>
#include<conio.h>
#include<string.h>
//Base Class
class person{
protected:
char name[20];
int age;

public:
person(){
strcpy(name, "");
age=0;
}
void readdata(){
cout<<"Enter name and age : ";
cin>>name>>age;
}
void showdata(){
cout<<"Name = "<<name<<endl;
cout<<"Age = "<<age<<endl;
}
};

//Derived class
class employee : public person{
private:
int eno;
```

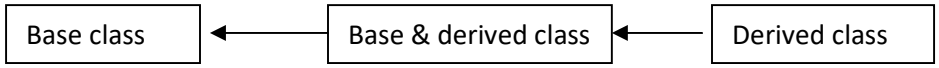
```
float salary;

public:
employee():person(){
eno = 0;
salary = 0;
}
void readdata(){
person::readdata();
cout<<"Enter eno and salary : ";
cin>>eno>>salary;
}
void showdata(){
person::showdata();
cout<<"eno = "<<eno<<endl;
cout<<"salary = "<<salary<<endl;
}
};
void main(){
employee e;
e.readdata();
e.showdata();
getch();
}
```

Output

```
Enter name and age : mohit
21
Enter eno and salary : 127
50000
Name = mohit
Age = 27
Eno = 127
Salary = 50000
```

Multilevel Inheritance : If we inherit a base class [x] into derived class [y] & then [y] to another derived class [z] & so on then this type of inheritance.




```
#include<iostream.h>
#include<conio.h>
#include<string.h>
//Base Class
class person{
protected:
char name[20];
int age;

public:
person(){
strcpy(name, "");
age=0;
}
void readdata(){
cout<<"Enter name and age : ";
cin>>name>>age;
}
void showdata(){
cout<<"Name = "<<name<<endl;
cout<<"Age = "<<age<<endl;
}
};

//Derived class and base class
class employee : public person{
protected:
int eno;
float salary;

public:
employee():person(){
eno = 0;
salary = 0;
}
void readdata(){
```

```
person::readdata();
cout<<"Enter eno and salary : ";
cin>>eno>>salary;
}
void showdata(){
person::showdata();
cout<<"eno = "<<eno<<endl;
cout<<"salary = "<<salary<<endl;
}
};

//Derived class
class manager: public employee{
int exp;
public:
manager():employee(){
exp = 0;
}
void readdata(){
employee::readdata();
cout<<"Enter Experience in Month : "<<endl;
cin>>exp;
}
void showdata(){
employee::showdata();
cout<<"Exprerience = "<<exp;
}
};
int main(){
clrscr();
manager e;
e.readdata();
cout<<endl;
e.showdata();
getch();
}
```

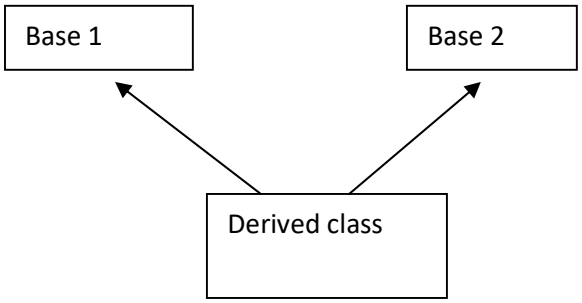
Output

```
Enter name and age : mohit
21
```

```
Enter eno and salary : 127
60000
Enter Experience in Month : 12

Name = mohit
Age = 21
Eno = 127
Salary = 60000
Experience = 12
```

Multiple Inheritance : when any single class inherits the features of more than one base class then such type of inheritance is called multiple inheritance.




```
#include<iostream.h>
#include<conio.h>
#include<string.h>
//Base Class
class person{
protected:
char name[20];
int age;

public:
person(){
strcpy(name, "");
age=0;
}
void readdata(){
cout<<"Enter name and age : ";
cin>>name>>age;
```

```
}
void showdata(){
cout<<"Name = "<<name<<endl;
cout<<"Age = "<<age<<endl;
}
};

//Base class
class employee{
private:
int eno;
float salary;

public:
employee(){
eno = 0;
salary = 0;
}
void readdata(){
cout<<"Enter eno and salary : ";
cin>>eno>>salary;
}
void showdata(){
cout<<"eno = "<<eno<<endl;
cout<<"salary = "<<salary<<endl;
}
};

//derived class
class manager: public employee, public person{
int exp;
public:
manager():employee(), person(){
exp = 0;
}
void readdata(){
person::readdata();
employee::readdata();
cout<<"Enter Experience in Month : "<<endl;
cin>>exp;
```

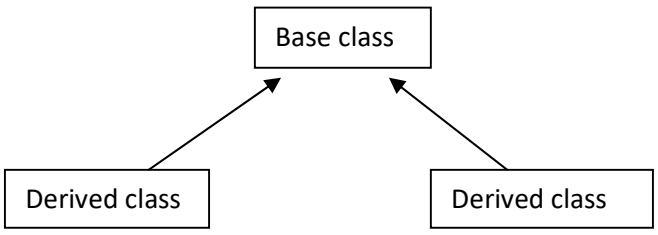
```
}
void showdata(){
person::showdata();
employee::showdata();
cout<<"Exprerience : "<<exp;
}
};
int main(){
clrscr();
manager e;
e.readdata();
e.showdata();
getch();
}
```

Output

```
Enter name and age : mohit
21
Enter eno and salary : 127
60000
Enter Experience in Month : 12

Name = mohit
Age = 21
Eno = 127
Salary = 60000
Experience = 12
```

Hierarchical Inheritance : In hierarchical two or more classes are derived from one base class




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

```
#include<conio.h>
#include<string.h>
//Base Class
class person{
protected:
char name[20];
int age;

public:
person(){
strcpy(name, "");
age=0;
}
void readdata(){
cout<<"Enter name and age : ";
cin>>name>>age;
}
void showdata(){
cout<<"Name = "<<name<<endl;
cout<<"Age = "<<age<<endl;
}
};

//Derived class
class employee:public person{
private:
int eno;
float salary;

public:
employee():person(){
eno = 0;
salary = 0;
}
void readdata(){
person::readdata();
cout<<"Enter eno and salary : ";
cin>>eno>>salary;
}
void showdata(){
```

```
person::showdata();
cout<<"eno = "<<eno<<endl;
cout<<"salary = "<<salary<<endl;
}
};

//Derived class
class manager:public person{
int exp;
public:
manager():person(){
exp = 0;
}
void readdata(){
person::readdata();
cout<<"Enter Experience in Month : "<<endl;
cin>>exp;
}
void showdata(){
person::showdata();
cout<<"Exprerience : "<<exp;
}
};
int main(){
clrscr();
employee e;
e.readdata();
e.showdata();
manager m;
m.readdata();
m.showdata();
getch();
}
```

Output

```
Enter name and age : mohit
22
Enter eno and salary : 127
70000
Name = mohit
```

```
Age = 22
Eno = 127
Salary = 70000
Enter name and age : mohit
23
Enter Experience in Month : 6
Name = mohit
Age = 22
Experience = 6
```

6. **Abstract Base Class**
   Any class for which we will not create any object is called abstract class. Its only purpose is to act as a base class for other classes.
   Eg. single inheritance

   class person (base class)

   class student (derived class)

7. **Ambiguity / confusion in Multiple Inheritance**
   In case of multiple inheritance sometimes compiler will be an ambiguous situation as shown below.

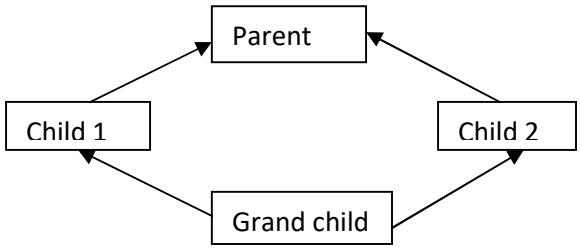
```
class base1{
public:
void show(){
cout<<"base1";
}
};
class base2{
public:
void show(){
cout<<"base2";
}
};
class derived:public base1, public base2{
};
```

```
void main(){
base1 b1;  //base 1
b1.show();
base2 b2;  //base 2
b2.show();
derived d1;
d1.show();  //  ]X Ambiguous statement will not be executed
d1.base1::show();  //base1
d1.base2::show();  //base2
}
```

8. Virtual Base class

When a class is made virtual base class, c++ takes necessary case to see that only one copy to that base class is inherited, regardless of how many inheritance paths exists between the virtual base class & a derived class.

Example




```
#include <iostream.h>
#include<conio.h>
#include<string.h>
//parent class
class person{
protected:
char* name;
int age;
public:
person(){
name  = "";
age = 0;
}
void readdata(){
```

```
cout<<"Enter name and age : ";
cin>>name>>age;
}
void showdata(){
cout<<"Name  = "<<name<<endl;
cout<<"Age = "<<age<<endl;
}
};

//child 1 class
class student: public person{
protected:
int sem;
int rollno;
public:
student():person(){
sem=0;
rollno=0;
}
void readdata(){
person::readdata();
cout<<"Enter sem  and rollno : ";
cin>>sem>>rollno;
}
void showdata(){
person::showdata();
cout<<"Semester  = "<<sem<<endl;
cout<<"Rollno = "<<rollno<<endl;
}
};

//child 2 class
class player: public person{
protected:
float height;
public:
player():person(){
height=0;
}
void readdata(){
```

```
        cout<<"Enter height  : ";
        cin>>height;
        }
        void showdata(){
        cout<<"height  = "<<height<<endl;
        }
        };


        //grandchild class
        class record: public student, public player{
        private:
        int marks;
        public:
        record():student(), player(){
        marks = 0;
        }
        void readdata(){
        student::readdata();
        player::readdata();
        cout<<"Enter marks : ";
        cin>>marks;
        }
        void showdata(){
        student::showdata();
        player::showdata();
        cout<<"Marks = "<<marks<<endl;
        }
        };
        void main(){
        record r1, r2;
        clrscr();
        r1.showdata();
        r2.readdata();
        r2.showdata();
        getch();
}
```

Output

```
Name =
```

```
Age = 0
Semester = 0
Rollno = 127
Height = 0
Marks = 0
Enter name and age : mohit
21
Enter semester and rollno : 6
127
Enter height : 6
Enter marks : 900
Name = mohit
Age = 21
Semester = 6
Rollno = 127
Height = 6
Marks = 900
```

9. Friend function

We can declare any function as friend of a class. Any function declared as friend of a class can access private data member of that class.

Example

```
//friend function
#include<iostream.h>
#include<conio.h>

class alpha{
private:
int data1;
public:
void read(){
cout<<"enter data : ";
cin>>data1;
}
friend void showsum();
};
```

```
class beta{
int data2;
void read(){
cout<<"enter data : ";
cin>>data2;
}
friend void showsum();
};

void showsum(){
alpha a;
beta b;
a.read();
b.read();
cout<<"sum is "<<a.data1+b.data2;
}
int main(){
clrscr();
showsum();
getch();
}
```

Output

```
Enter data 34
Enter data 12
Sum is 46
```

10. Friend class

If we want to access private data members of a class (x) into another class (y) as friend of class (x) using the friend keyword & we can all data member of (x) into class (y).

Example

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

```
class alpha{
private:
int data1;
friend beta;  //beta access members of alpha
};

class beta{
private:
int data2;
public:
alpha a1; //object of class alpha
void get(){
cout<<"Enter values\n";
cin>>a1.data1;
cin>>data2;
}
void sum(){
int s=a1.data1+data2;
cout<<"sum = "<<s;
}
};

int main(){
clrscr();
beta b1;
b1.get();
b1.sum();
getch();
}
```

Output

```
Enter values
23
54
Sum = 77
```

11. Polymorphism

Polymorphism is refers to ability of one thing to take many forms.

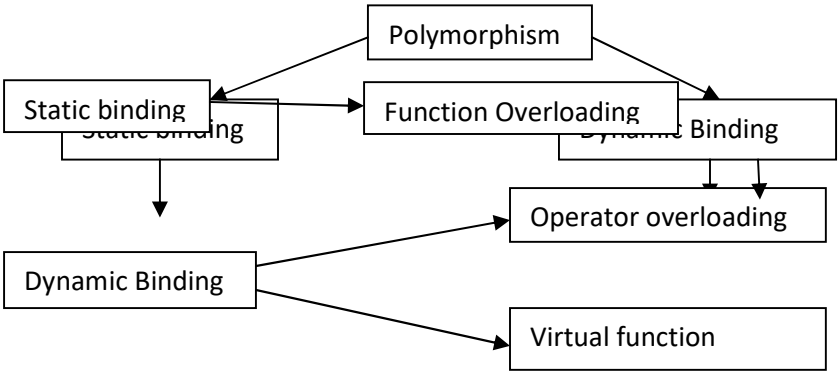
Polymorphism has two types of binding :

(i) Static Binding \ Compile time Binding \ Early Binding

(ii)Dynamic Binding \ Run time Binding \ Late Binding.

Binding means connecting the function call to the code to be executed in response to the call

To call any member function with the pointer to object, we arrow operator (->) between pointer & function



Example

```
#include<iostream.h>
#include<conio.h>
class base{
public:
void show(){
cout<<endl<<"base";
}
};
class derived : public base{
public:
void show(){
cout<<endl<<"Derived";
}
};
```

```
void main(){
base *ptr;   //pointer to class base
base b1;   //Base class object
derived d1;  //Derived class object

ptr = &b1;   /*pointer points to object b1 (Dynamic Binding / upcasting)   */

//Base show function
ptr ->show();    //function call

ptr = &d1;
//Derived show Function
ptr->show();
getch();
}
```

Output

```
Base
Base
```

12. pointer to Derived classes

c++ allows a pointer in a base class to point to either a base class object or to any derived class object.

For eg. if base is the name of a base class and derived is the name of a derived class, then a pointer declared to derived. Example is shown above.

13. Static binding or compile time

Static binding means that the code associated with the function call is linked at compile time. Static binding is also called as early binding.

Function overloading & operator overloading falls in this category.

14. Dynamic binding

Dynamic Binding means that the code associated with the function call is linked as runtime Dynamic binding as also known as late binding.

15. Virtual function

A member of a class that can be redefined in its derived class is known as virtual member. Virtual function allows derived classes to replace the implementation provided by the base class. The derived class can fully or partially replace the base class member function. Virtual must be defined in the base class

" In above example(polymorphism) the compiler of c++ will check the type of pointer (ptr) & not address on it & hence every time the function show() from the base class will be executed.

To remove this problem we declare function show() in the base class as virtual function "

Example

```
#include<iostream.h>
#include<conio.h>
class base{
public:
virtual void show(){ //virtual function
cout<<endl<<"base";
}
};
class derived : public base{
public:
void show(){
cout<<endl<<"Derived";
}
};
```

```
void main(){
base *ptr;   //pointer to class base
base b1;   //Base class object
derived d1;  //Derived class object

ptr = &b1;   /*pointer points to object b1 (Dynamic Binding / upcasting)  */
//Base show function
ptr ->show();    //function call

ptr = &d1;
//Derived show Function
ptr->show();
getch();
}
```

Output

```
Base
Derived
```

16. Pure virtual function

A pure virtual function is a function declared in a base class & has no definition relative to the base class

Example

```
#include<iostream.h>
#include<conio.h>
class base{
private:
int a;
public:
base()
{
a=10;
}
virtual void show()=0;
};

class derived: public base{
```

```
int b;
public:
derived(){
b=20;
}
void show(){
cout<<b;
}
};

int main(){
clrscr();
base *ptr; //pointer to class base
derived d1; //object of class derived
ptr = &d1; //pointer to objects do
ptr->show();
getch();
}
```

17. Virtual destructor

Destructor is invoked to free the memory storage by the c++ compiler automatically. But the destructor member function of the derived class is not invoked to free the memory storage which was allocated by the constructor. It is become the destructor member functions are non virtual & the message will not reach the destructor member functions under late binding. So it is better to have a member function as virtual.

Example

```
#include<iostream.h>
#include<conio.h>
class base{
int a;
public:
base(){
a = 10;
}
```

```
virtual void show()=0;
virtual ~ base(){
cout<<"Base class Destructed";
}
};

class derived : public base{
int b;
public:
derived(){
b =20;
}
void show(){
cout<<b;
}

virtual ~ derived (){
cout<<"derived class destructed";
}
};

void main(){
clrscr();
base *ptr;
derived d1;
ptr=&d1;
ptr->show();
getch();
}
```

Output

```
20
Derived class destructed
Base class destructed
```

18. Constructor overloading

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

```
class distence{
	int feet;
	float inches;
	public:
	distence(){   //single constructor
		fect = 0;
		inches = 0.0;
	}
	distence(int ft, float in){  //constructor with argument
		feet=ft;
		inches=in;
	}
	void readdata(){
		cout<<"Enter feet & inches";
		cin>>feet>>inches;
	}
	void showdata(){
		cout<<"feet"<<feet<<endl;
		cout<<"Inches"<<inches<<endl;
	}
};

int main(){
	distence d1, d2(3,4.5), d3;
	d1.showdata();  //0, 0.0
	d2.showdata();  //3, 4.5
	d3.readdata();  //read
	d3.showdata();  //write
}
```

19. Function overloading

```
#include<iostream.h>
#include<conio.h>
class printData{
public:
void print(int i){
cout<<"Displaying the int : "<<i<<endl;
}
```

```
void print(char* c){
cout<<"Displaying the character : "<<c<<endl;
}
void print(double f){
cout<<"Displaying the float : "<<f<<endl;
}
};
int main(){
clrscr();
printData pd;
pd.print(5); //call to print integer
pd.print(500.263); //call to print float
pd.print("Hello c++"); //call to print character
getch();
}
```

20. Unary operator overloading

```
#include<iostream.h>
#include<conio.h>
class abc{
int count;
public:
abc(){
count=0;
}
void show(){
cout<<count<<endl;
}

void operator ++(){   //operator overloading
count++;
}
};

int main(){
clrscr();
abc a, b;
a++;
```

```
++a;
a.show();
b++;
b++;
++b;
b.show();
getch();
}
```

Output

```
2
3
```

21. Binary operator overloading

```
#include<iostream.h>
#include<conio.h>
class time{
int hour, minute, second;
public:
time(){
hour=0;
minute=0;
second=0;
}

time(int h, int m, int s){
hour=h;
minute=m;
second=s;
}

time operator +(time t){  //Operator overloading
time temp;
temp.hour=hour+t.hour;
temp.minute=minute+t.minute;
temp.second=second+t.second;
if(temp.second>=60){
temp.minute++;
```

```
temp.second-=60;
}
if(temp.minute>=60){
temp.hour++;
temp.minute-=60;
}
return temp;
}

void show(){
cout<<"hour : "<<hour<<endl;
cout<<"minute : "<<minute<<endl;
cout<<"second : "<<second<<endl;
}
};
int main(){
clrscr();
time t1, t2(2,30,35), t3(3,20,35);
t1=t2+t3;
t1.show();
getch();
}
```

Output

```
hour : 5
minute : 51
second : 10
```

22. Inline function

C++ inline function is power-full concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Example

```
#include<iostream.h>
#include<conio.h>
inline float mul(float x, float y){
return (x*y);
```

```
        }

        inline float div(float p, float q){
        return (p/q);
        }

        int main(){
        clrscr();
        float a=12.345;
        float b=2.82;
        cout<<"Multiplication : "<<mul(a,b)<<endl;
        cout<<"Division : "<<div(a,b);
        getch();
}
```

23. Static data member

```
        #include<iostream.h>
        #include<conio.h>
        class item{
        static int count;
        int number;

        public:
        item(){
        number=0;
        count=0;
        }

        void getdata(int a){
        number=a;
        count++;
        }

        void getcount(){
        cout<<"number : "<<number<<endl;
        cout<<"count : "<<count<<endl;
        }
```

```
        ~item(){
        cout<<"Destroying the object"<<endl;
        }
        };

        int item::count;
        int main(){
        item a,b,c;
        clrscr();
        cout<<"Object crated calling default constructor : "<<endl;
        a.getcount();  // count variable have value zero
        b.getcount();
        c.getcount();
        cout<<"Object created with the member function : "<<endl;
        a.getdata(100);  /*all the object's count variable have total of
        objects(a+b+c = 3) */
        b.getdata(200);
        c.getdata(300);
        a.getcount();
        b.getcount();
        c.getcount();
        getch();
}
```

Output

```
Object crated calling default constructor :
number : 0
count : 0
number : 0
count : 0
number : 0
count : 0
Object created with the member function :
number : 100
count : 3
number : 200
count : 3
number : 300
count : 3
```

24. Static member function

We can define class member static using static keyword when we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

Example

```
#include<iostream.h>
#include<conio.h>
class Cube{
public:
        static int objectCount;
        Cube(double l=5.0, double b=5.0, double h=5.0){
                cout<<"Constructor called."<<endl;
                length=l;
                breadth=b;
                height=h;
                //Increase every time object is created
                objectCount++;
        }
        double volume(){
                return (length*breadth*height);
        }
        static int getCount(){
                return objectCount;
        }
private:
        double length; //length of a Cube
        double breadth; //Breadth of a Cube
        double height; //Height of a Cube
};

//Initialization static memeber of class Cube
int Cube::objectCount=0;
int main(void){
        clrscr();
        //Print total number of object after creating object.
        cout<<"Initially Object Count : "<<Cube::getCount()<<endl;
```

```
        Cube c1(2.0, 1.2, 5.6), c2(23.0, 3.6, 56.8), c3(23.7, 56.6, 12.5);
        //Print total number of object after creating object.
        cout<<"Finally Object Count : "<<Cube::getCount()<<endl;
    getch();
}
```

Output

```
Initially Object Count : 0
Constructor Called.
Constructor Called.
Constructor Called.
Finally Object Count : 3
```