BASIC OOP CONCEPTS:

Concepts of object-oriented programming include:

1. Classes
2. Objects
3. Data Abstraction
4. Encapsulation
5. Inheritance
6. Polymorphism
7. Data Hiding

CLASSES:

1. Classes are user-defined data-type, which contains collection of data members and member functions on which objects works.
2. Once a class has been declared, we can create any number of objects of that class.
3. Objects are variables of the type class.
4. A class is a collection of objects of similar types.
5. For example, mango, apple, and orange are members of the class “fruit”.
6. A class is defined in C++ using keyword “class” followed by the name of class.
7. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

```
class className
```

```
{
 // some data
// some functions
};
```

Example: Class in C++

```
class Test
{
   private:
       int data1;
       float data2;

   public:
       void function1()
       {   data1 = 2;  }

       float function2()
       {
           data2 = 3.5;
           return data2;
       }
  };
```

Here, we defined a class named "Test".

This class has two data members: data1 and data2 and two member functions: function1() and function2().

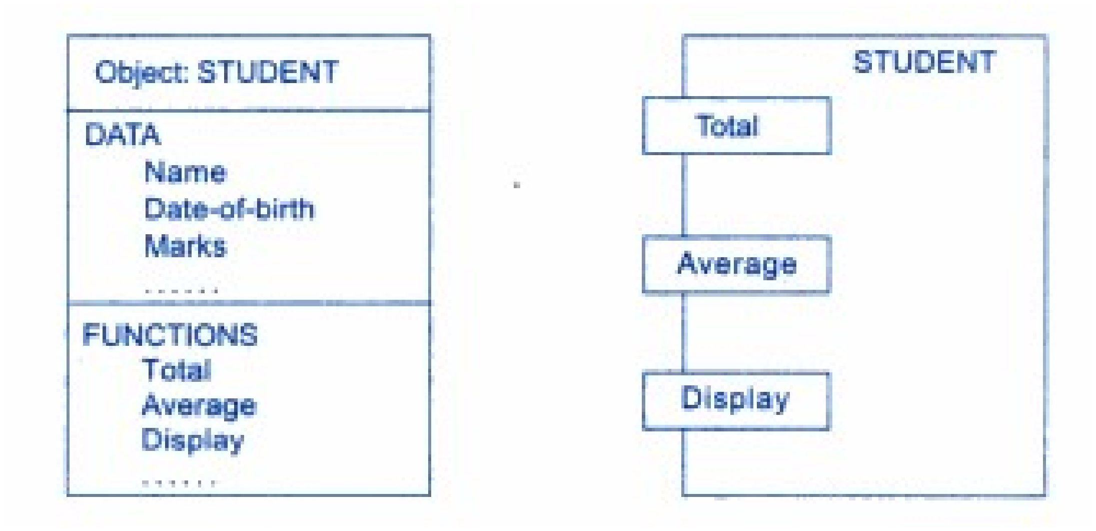
8. Class members can be private, public, or protected.
9. If fruit has been defined as a class, then the statement

   Fruit mango;

   Will create an object "mango" of the class Fruit.

OBJECT

1. Objects are the basic run-time entities in an object-oriented system.
2. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.
3. No storage is assigned when we define a class.
4. Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.
5. Each object has different data variables. Objects are initialized using special class functions called **Constructors**.
6. Whenever the object is not in use , another special class member function called **Destructor** is called, to release the memory reserved by the object.
7. Two ways of representing objects of class are:



Here we have considered object “student”.

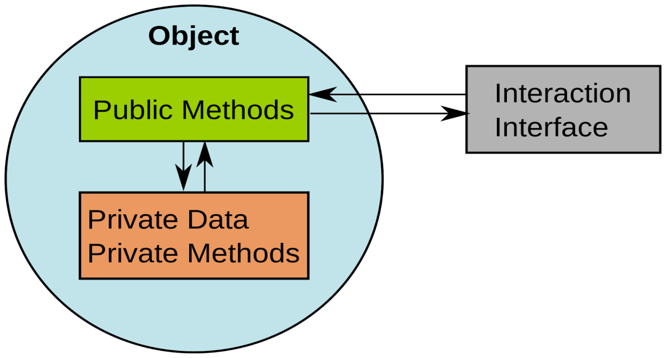
DATA ABSTRACTION:

1. Abstraction refers to the act of representing essential features without including the background details or explanations.
2. Classes use the concept of abstraction, so they are known as abstract data types (ADT).
3. Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.
4. Let's take one real life **example of a TV**, which you can
   - Turn on and off,
   - Change the channel,
   - Adjust the volume, and
   - Add external components such as speakers, VCRs, and DVD players.

BUT you do not know its internal details, that is,

- How it receives signals through a cable,
- How it translates them, and
- Finally displays them on the screen.

5. Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.



Abstraction using Classes: We can implement Abstraction in C++ using classes. Class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to outside world and which is not.

Abstraction using access Specifies:

Access specifiers help in implementing abstraction in C++. For example:

1. Members declared as **public** in a class, can be accessed from anywhere in the program.
2. Members declared as **private** in a class, can be accessed only from within the class.

Benefits of Data Abstraction:

1. Class internals are protected from user-level errors.
2. The class definition can be change without requiring change in user-level code.

3. Helps the user to avoid writing the low level code.
4. Avoids code duplication and increases reusability.
5. Helps to increase security of an application or program as only important details are provided to the user.

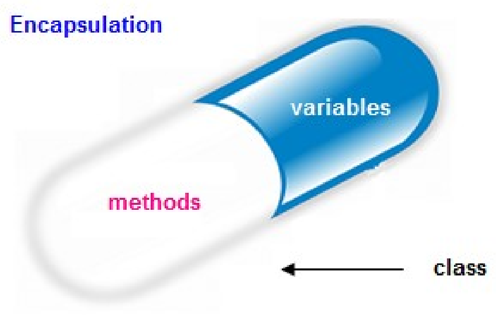
ENCAPSULATION:

- The wrapping up of data and functions into a single unit (called class) is known as encapsulation.
- All C++ programs are composed of the following two fundamental elements:

1) Program statements (code): This is the part of a program that performs actions and they are called functions.

2) **Program data:** The data is the information of the program, which is used by the program functions.

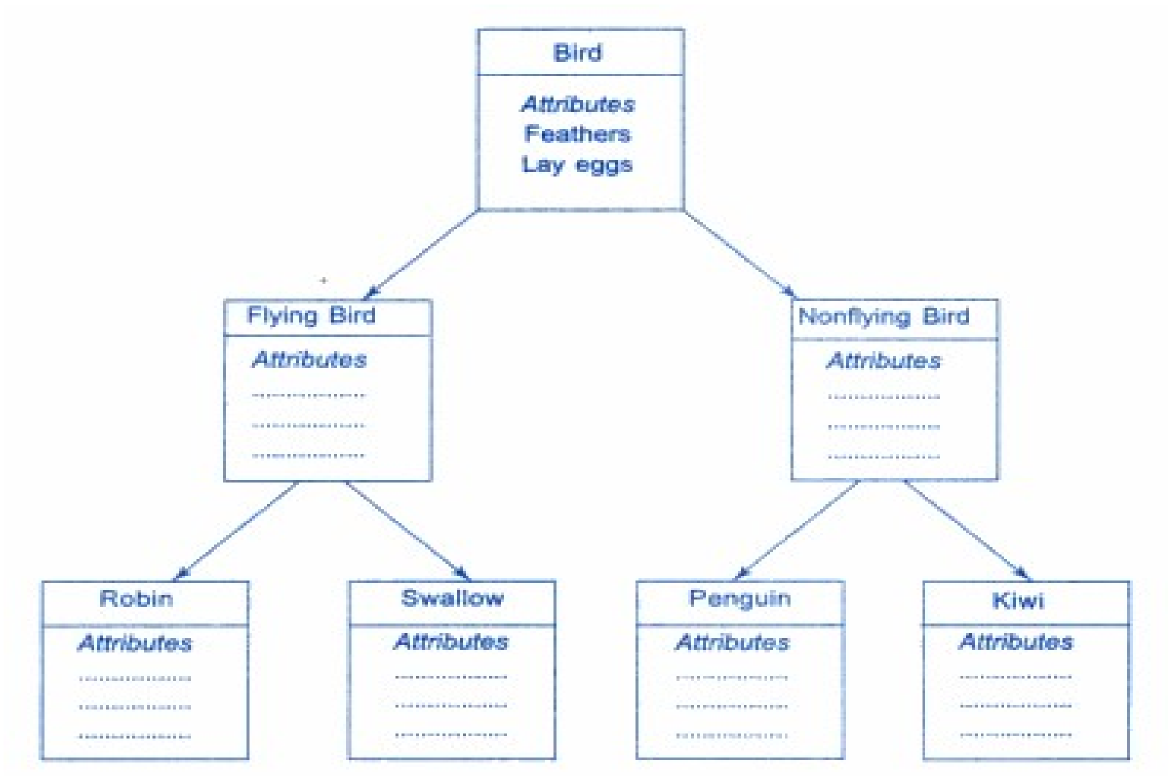
- OOP treats data, as an important element in program development, and does' not allow it to flow freely in the system. It ties data more closely to the functions that operate on it in a data structure called class.
- Encapsulation is an Object Oriented Programming concept that binds together the data and functions, and that keeps both safe from outside misuse.
- This feature is called "Data Encapsulation", where data members and member functions are packed in a class as small tablets are packed inside a capsule (medicine).



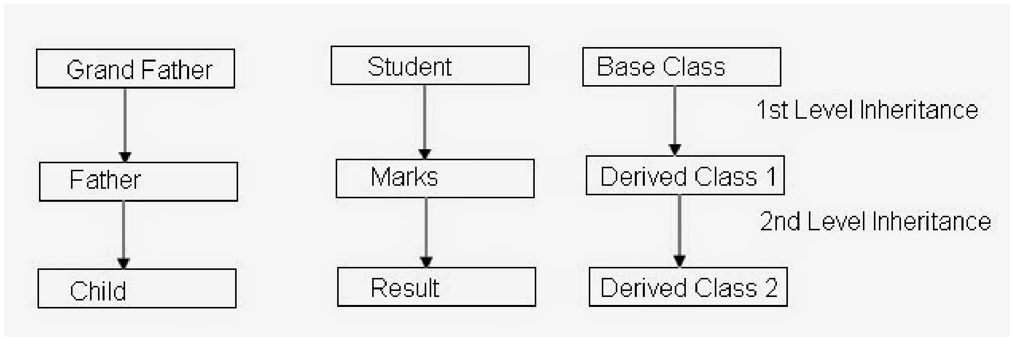
INHERITANCE:

⇔ *Inheritance* is the process by which objects of one class acquire properties of objects of another class.

1. Inheritance gives hierarchical structure of class and subclass in the form of "parent-child" relationship.
2. For example: the bird "robin" is a part of the class 'flying bird' which is again a part of the class 'bird'. Here class 'bird' is called super-class and class 'flying bird' is called sub-class(derived class).

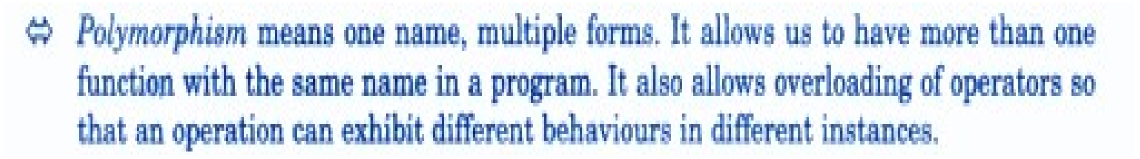


3. In inheritance each derived class shares common characteristics of its super class.
4. Each subclass shares all the attributes of super class and defines only that features which are unique to it.

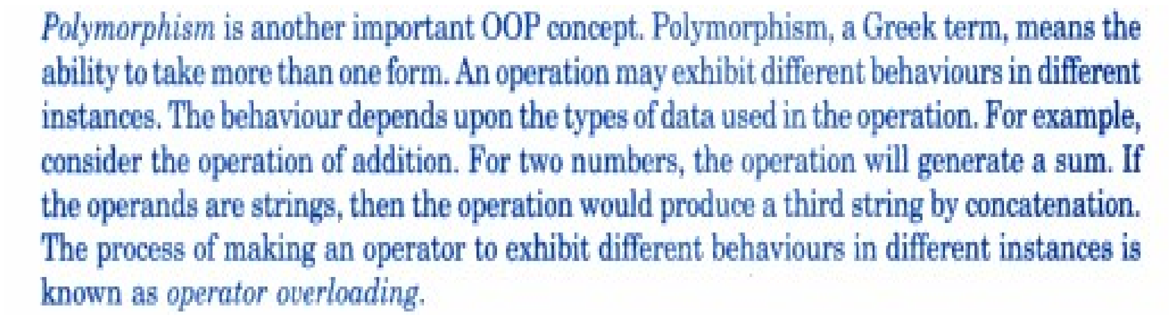


5. It provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from existing one.

POLYMORPHISM:

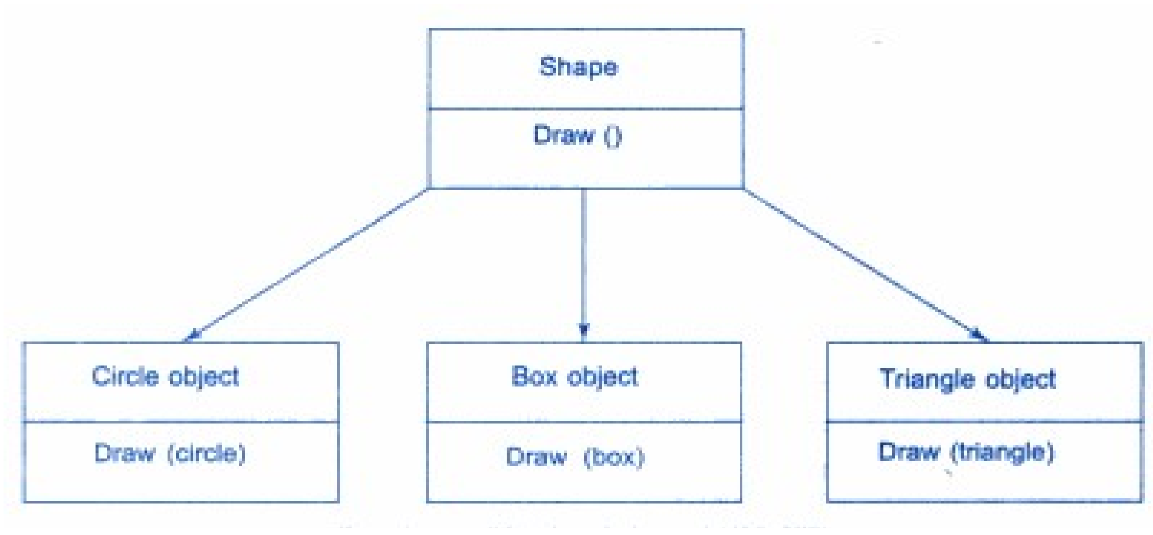


⇔ *Polymorphism* means one name, multiple forms. It allows us to have more than one function with the same name in a program. It also allows overloading of operators so that an operation can exhibit different behaviours in different instances.



Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviours in different instances is known as *operator overloading*.

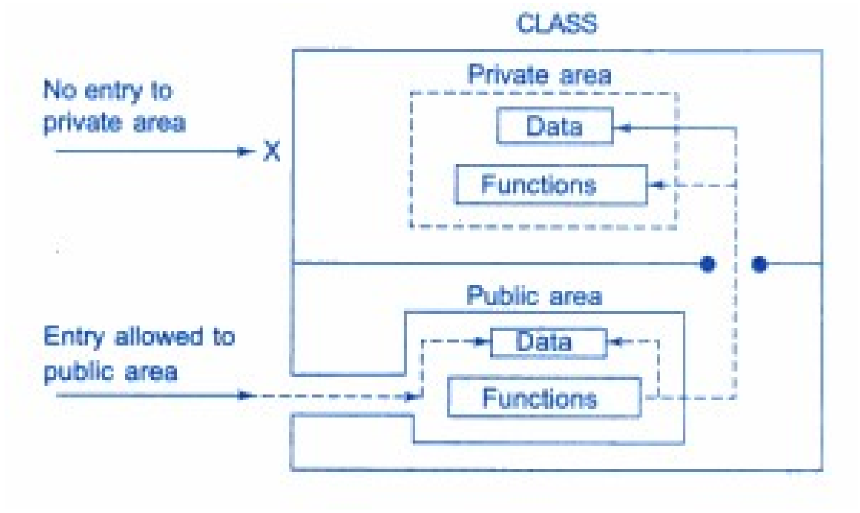
A single function name can be used to handle different number and different types of arguments. Single function name can be use to perform different types of tasks is known as function overloading



DATA-HIDING:

⇔ Insulation of data from direct access by the program is called *data hiding*.

➢ In OOP class data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. This feature is called data hiding or Information hiding. By default class data is private. But we can also make it public or protected.



Ques: What is OOP? How it is differ from POP?

Object oriented programming refers to a type of computer programming, which was invented to overcome the drawbacks of the POP. It follows "bottom-up" programming approach.

OOP gives more importance to data and does not allow it to flow freely around the system. It ties data to the functions that operates on it in the form of Class.

In POP approach, the problem is viewed as a sequence of things to be done, such as, input taking, calculating and displaying. **The primary focus stays on functions** which will be used to accomplish each task.

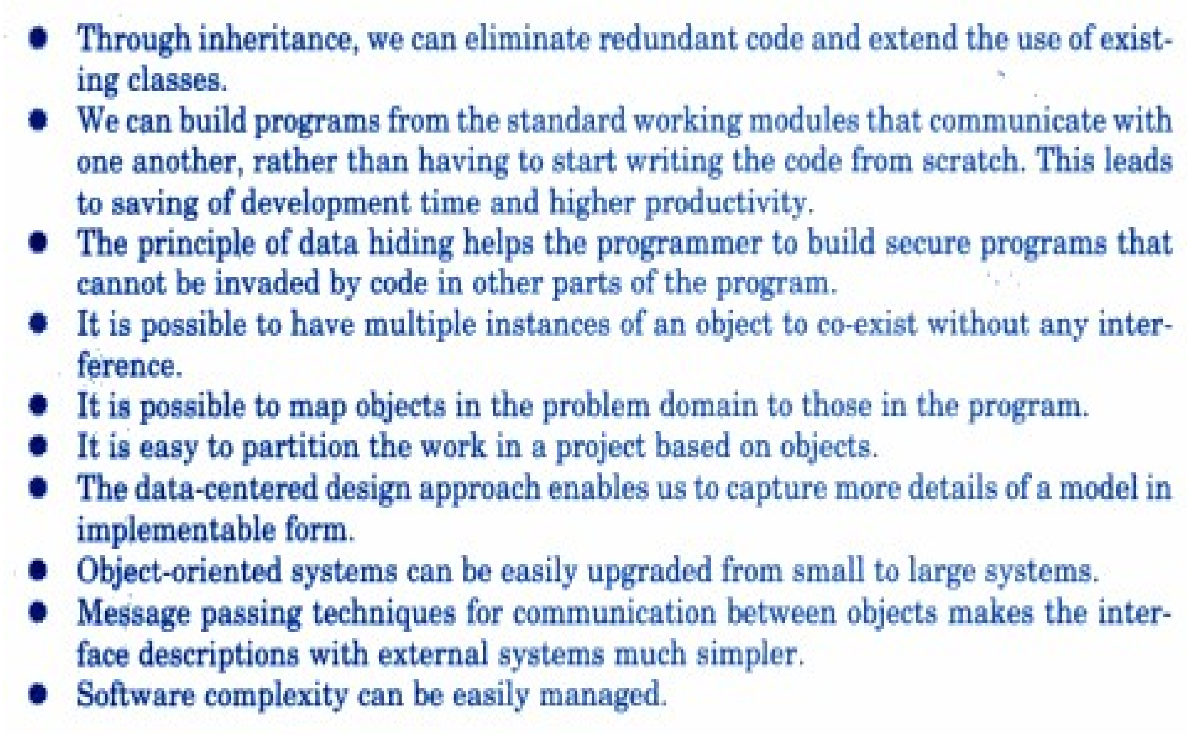
Difference between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP)

Factor	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data because it works as a **real world**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.

Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Programming Approach	It follows top down approach.	It follows bottom up approach.
Examples	Examples of POP are: C, VB, FORTRAN, Pascal.	Examples of OOP are: C++, JAVA, VB.NET, C#.NET.

Ques: State the advantages of OOP compare to POP .

Ans: Advantages of OOP are:



- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

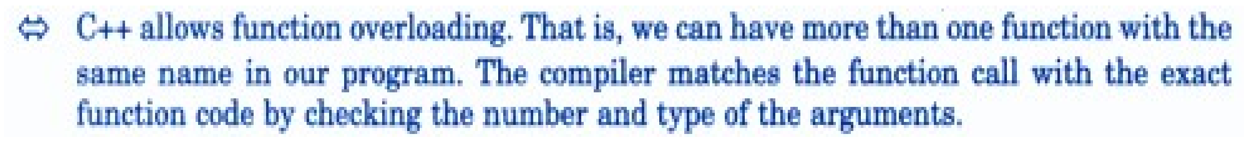
QUES: What is function overloading? When do we need to use it?

ANS: In C++, Single function name can be use to perform different types of tasks, this is known as function overloading.

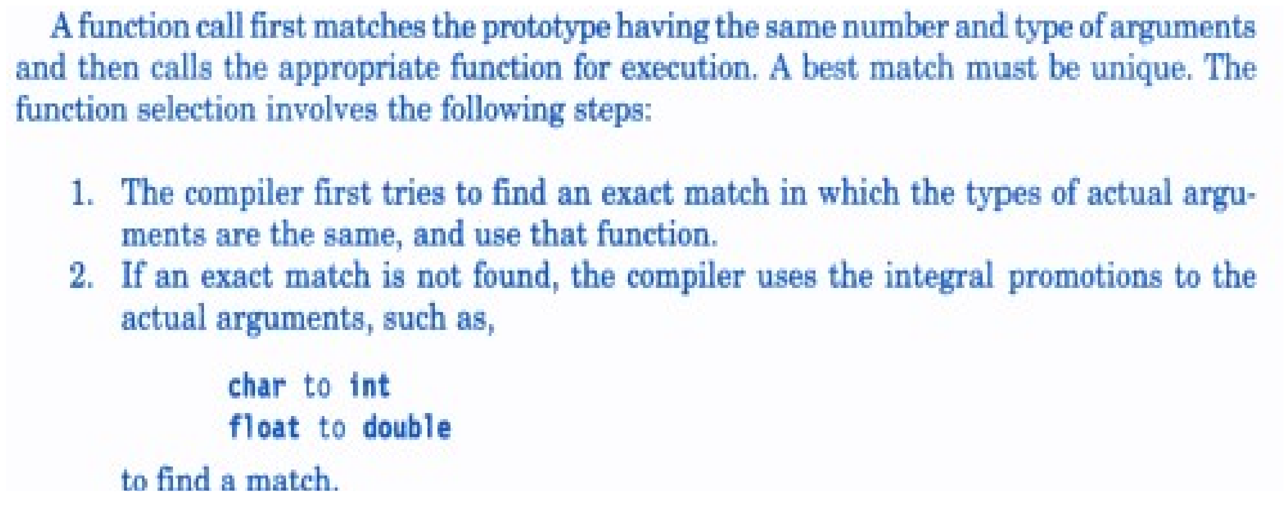
Format of overloaded function includes:

1. Number of arguments
2. Type of arguments
3. Sequence of arguments

When you call an overloaded function, the compiler determines the most appropriate function definition to use by comparing the format of calling statement with the function format specified in the definitions.



⇔ C++ allows function overloading. That is, we can have more than one function with the same name in our program. The compiler matches the function call with the exact function code by checking the number and type of the arguments.

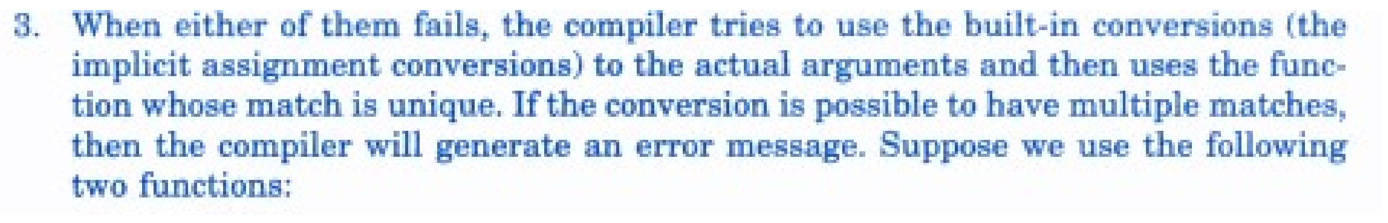


A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

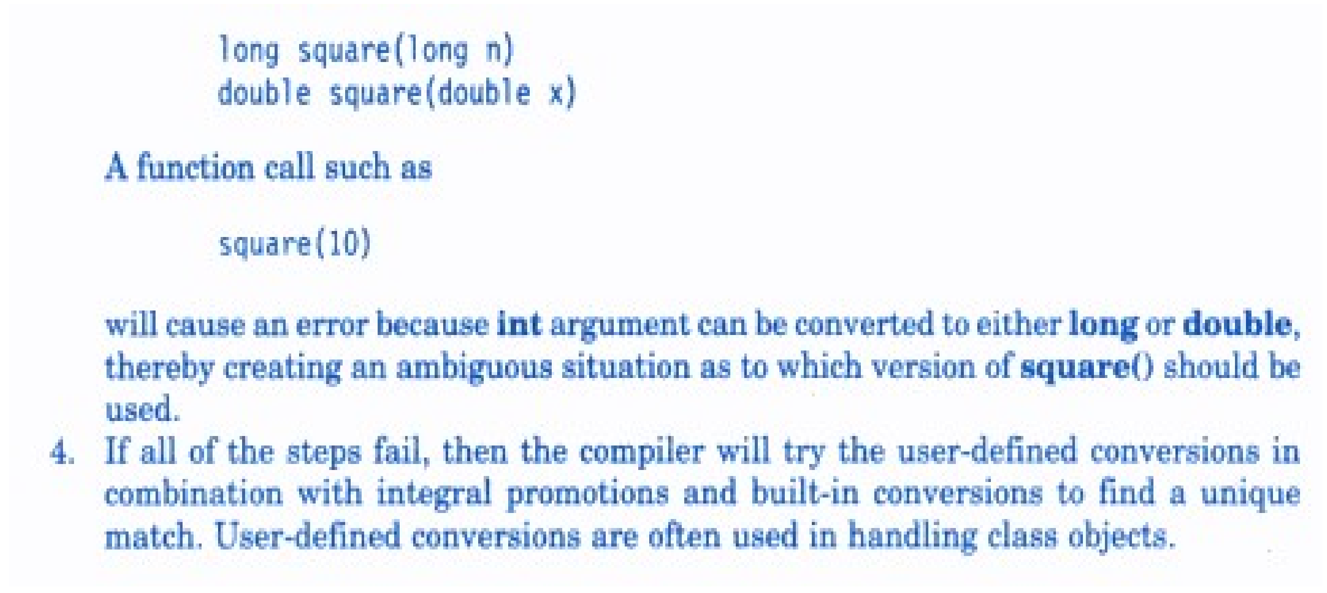
1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

```
        char to int
        float to double
```

   to find a match.



3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:



```
long square(long n)
double square(double x)
```

A function call such as

```
square(10)
```

will cause an error because **int** argument can be converted to either **long** or **double**, thereby creating an ambiguous situation as to which version of **square()** should be used.

4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Example of function overloading: this program prints the given character for a given number of times. Here function repchar() is overloaded.

```
#include<stdio.h>
#include<conio.h>
#include<iostream.h>
class r1
  {
  public:
 void repchar();
 void repchar(char ch);
 void repchar(char ch,int n);
  };
   void r1:: repchar()
  {
  int i,num;
  char ch;
  cout<<"enter the character"<<"\n";
  cin>>ch;
  cout<<"how many times"<<"\n";
  cin>>num;

for(i=0;i<=num;i++)
          {
          cout<<ch;
          }
}

void r1::repchar(char ch )
{
int i,num;
  cout<<"how many times"<<"\n";
  cin>>num;
 for(i=0;i<=num;i++)
          {
          cout<<ch;
          }
}
void r1::repchar(char ch,int n)
{  int i;

     for(i=0;i<=n;i++)
          {
          cout<<ch;
          }
}
  void main()
  {
  clrscr();
  char ch='S';
  r1 r;
r.repchar();
r.repchar(ch);
cout<<endl;
r.repchar(ch,10);
getch();
  }
```

Ques: What do you mean by overloading of an operator? Why is it necessary to overload an operator?

Ans: the process of giving **special meaning** or **additional task to an operator** is called operator overloading. it provides option of creating **new definitions** for most of the C++ operators.

Operator overloading is done with the help of a special function, called operator function, which describes the special task to an operator.

The general form of an **operator function** is:

```
return type classname :: operator op(arglist)
{
        Function body            // task defined
}
```

where *return type* is the type of value returned by the specified operation and *op* is the operator being overloaded. The *op* is preceded by the keyword **operator**. **operator** *op* is the function name.

The process of overloading involves the following steps:

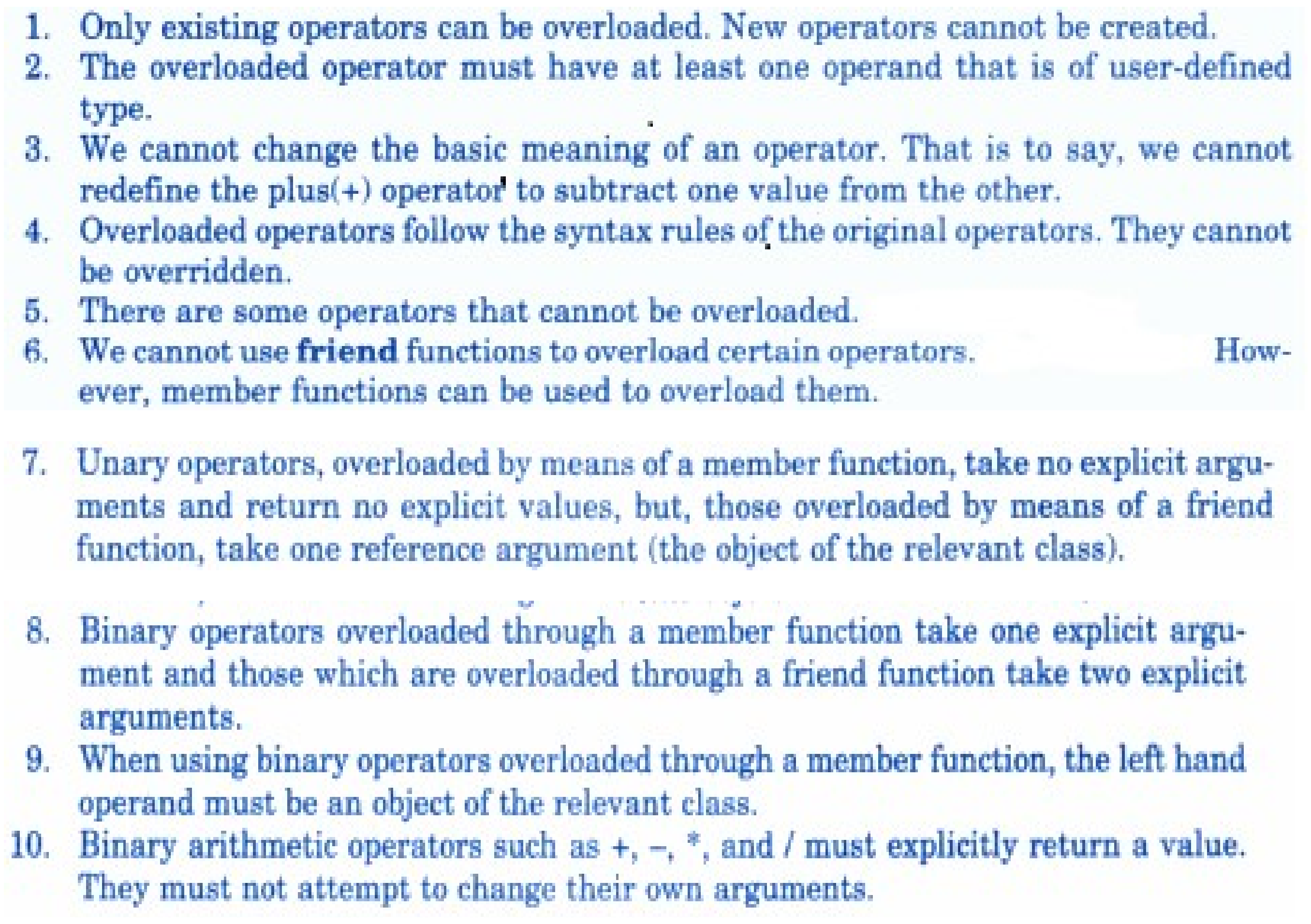
- Create a class that defines the data type that is to be used in the overloading operation.
- Declare the operator function in the public part of the class.
- Define the operator function to implement the required operation.

Overloaded operator function can be invoked by expression such as

Op X or X op # for unary operator and

X op Y # for binary operator

Rules for overloading operators are:



1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded.
6. We cannot use **friend** functions to overload certain operators. However, member functions can be used to overload them.
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +, –, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Ques: Why is operator overloading used?

You can write any C++ program without the knowledge of operator overloading. However, programmers to make program spontaneous use operator overloading. For example,

You can replace the code like:

```
calculation = add(multiply(a, b),divide(a, b));
```

To

```
calculation = (a*b)+(a/b);
```

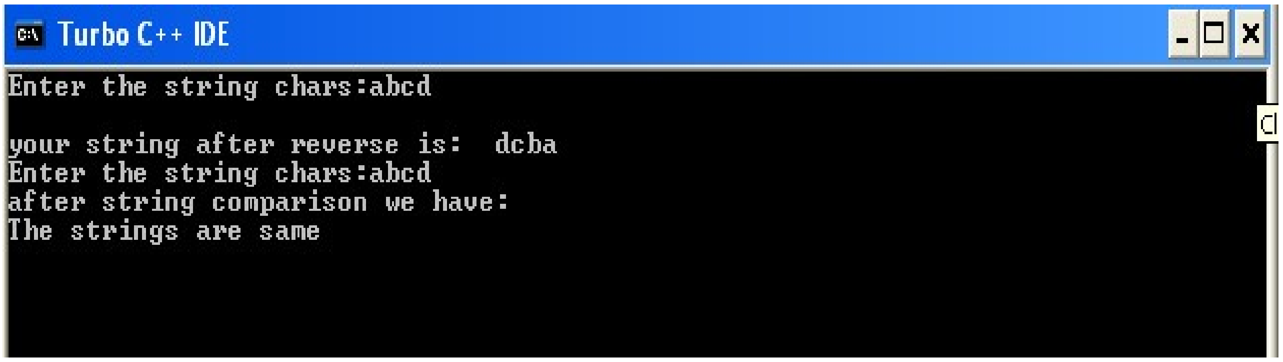
Example of operator overloading is:

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class string1
{
        char a[20];
    public:
        void input(void);
        void output(void);
        friend void operator -(string1);
        friend void operator ==(string1,string1);
};
void string1 :: input(void)
{
        cout<<"Enter the string chars:";
        cin>>a;
}
void string1 :: output(void)
{
	cout<<"your string is : "<<a;
}
void operator -(string1 A)
{
strrev(A.a);
cout<<"your string after reverse is:  "<<A.a;


}
void operator ==(string1 x,string1 y)
{
	if(strcmp(x.a,y.a)==0)
	{
		cout<<endl<<"The strings are same";
	}
	else
	{
		cout<<endl<<"The strings are different";
	}
}
void main()
{
	string1 abc,abc1;
```

```
        clrscr();

        abc.input();

        cout<<"\n";

        -abc;

        cout<<endl;

        abc1.input();

        cout<<"after string comparison we have: ";

        abc==abc1;

        getch();

}
```

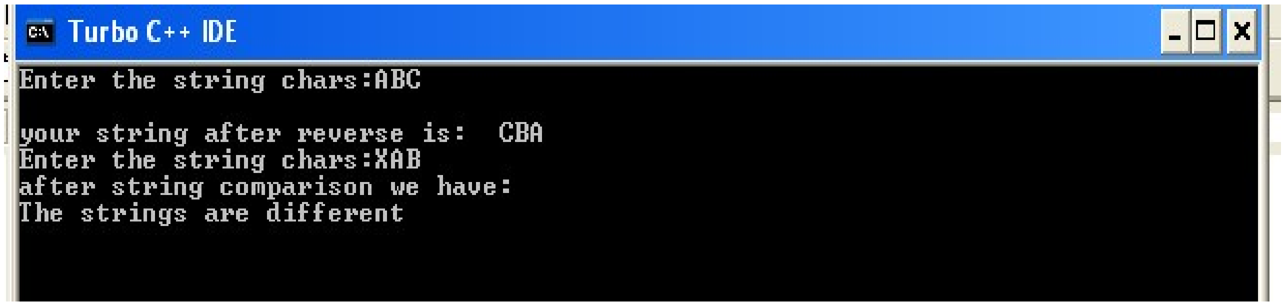
OUTPUT:

FOR DATA 1:



```
Enter the string chars:abcd

your string after reverse is:  dcba
Enter the string chars:abcd
after string comparison we have:
The strings are same
```

FOR DATA 2



```
Enter the string chars:ABC

your string after reverse is:  CBA
Enter the string chars:XAB
after string comparison we have:
The strings are different
```

CONSTRUCTORS:

Constructor is a ‘special’ member function used to create and initialize the objects of its class.

Characteristics of Constructor.

1. Constructor name and class name must be same.
2. Constructor doesn't return value.
3. Constructor is invoked automatically, when the object of class is created.
4. It is called constructor because it constructs the values of data members of the class.
5. They should be declared in the public section.
6. They cannot be inherited.
7. They can have default arguments.
8. Constructor cannot be virtual.
9. We cannot refer to their addresses.
10. Constructor may be overloaded.

TYPES OF CONSTRUCTOR

1. Default Constructor.
2. Copy Constructor.
3. Parameterize Constructor.

DEFAULT CONSTRUCTOR

- A constructor without any arguments is said to be ***default constructor***.
- The default constructor of class A is A::A().
- If no such constructor is defined, then the compiler supplies a default constructor.
- The statement: A a; invokes the default constructor of the compiler to create the object a.

COPY CONSTRUCTOR:

- Initialization of an object through another object is called **copy constructor**.
- In other words, copying the values of one object into another object is called **copy constructor.**
- An object can be initialized with another object of same type. This is same as copying the contents of a class to another class.
- In the below program, if you want to initialize an object A3 of class Area. So that it contains same values as A2, this can be performed as:

```
int main()

{

   Area A2;

// Copies the content of A2 to A3

   Area A3(A2);

     OR,

   Area A3 = A2;

}
```

EXAMPLE OF DEFAULT AND COPY CONSTRUCTOR:

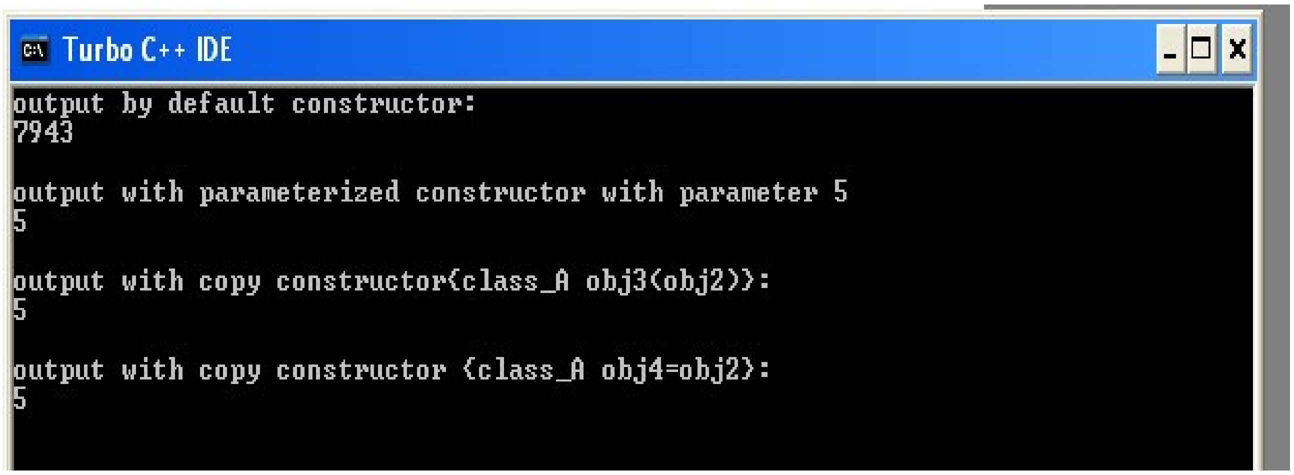
Following program demonstrates the use of default and copy constructor:

```
#include<iostream.h>
#include<conio.h>
class class_A
{
int id;
public:
```

```
class_A()
{
}
class_A(int x)
{
id=x;
}
class_A(class_A &y)
{
id=y.id;
}
void display()
{
cout<<id<<endl<<endl;
}
};
void main()
{
clrscr();
class_A obj1;
cout<<"output by default constructor:"<<endl;
obj1.display();
int p=5;
class_A obj2(p);
```

```
cout<<"output with parameterized constructor with parameter 5"<<endl;
obj2.display();
class_A obj3(obj2);
cout<<"output with copy constructor{class_A obj3(obj2)}:"<<endl;
obj3.display();
class_A obj4=obj2;
cout<<"output with copy constructor {class_A obj4=obj2}:"<<endl;
obj4.display();
getch();
}
```

OUTPUT:



```
output by default constructor:
7943

output with parameterized constructor with parameter 5
5

output with copy constructor{class_A obj3(obj2)}:
5

output with copy constructor {class_A obj4=obj2}:
5
```

Ques: Explain parameterized constructor. How constructor is called explicitly and Implicitly.

The constructors that can take arguments are called *parameterized constructors.*

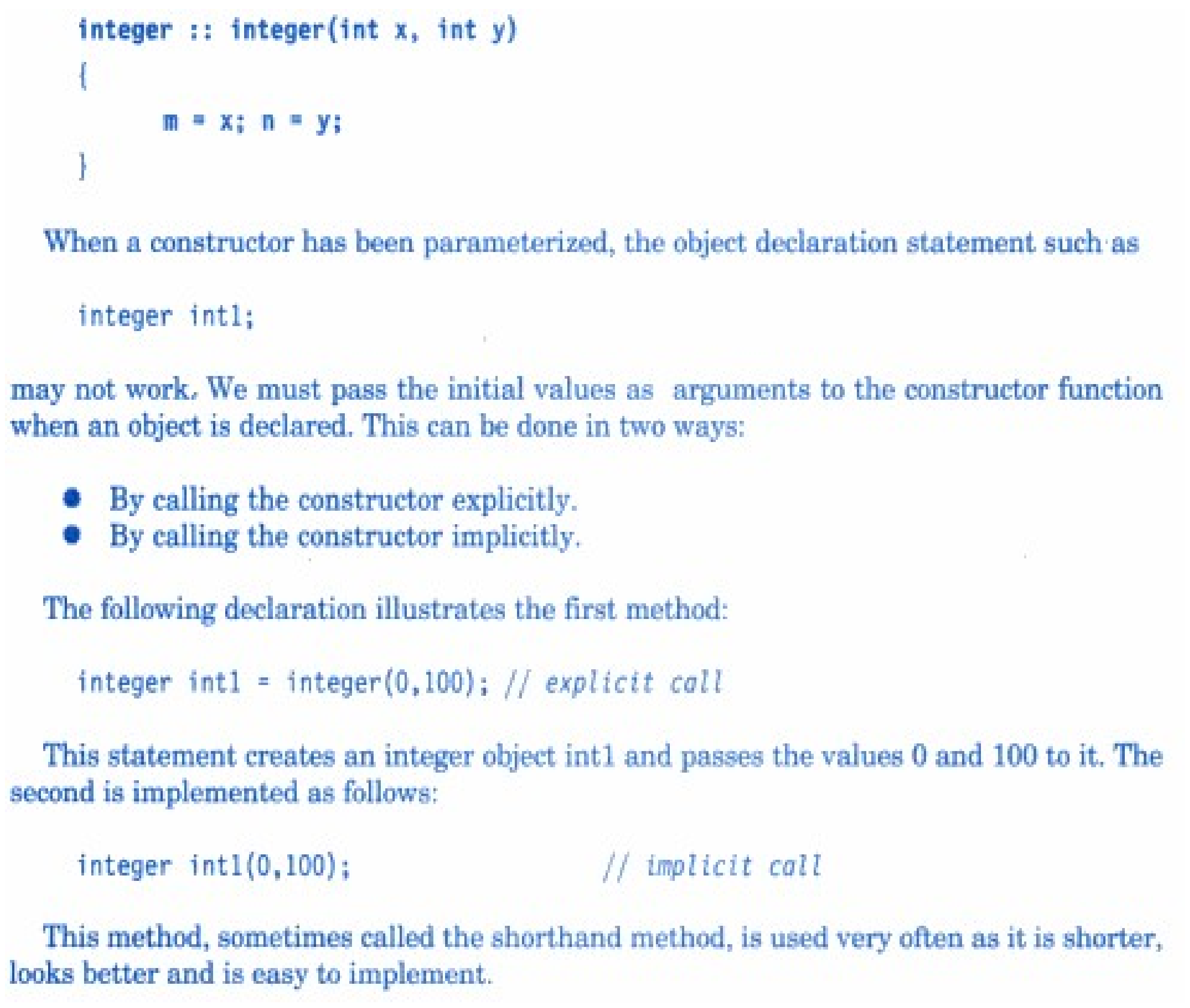
⇔ C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects.

⇔ A constructor has the same name as that of a class.

⇔ Constructors are normally used to initialize variables and to allocate memory.

Example:

```
class integer
{
        int m, n;
   public:
        integer(int x, int y);   // parameterized constructor
        .....
        .....
};
```



```
integer :: integer(int x, int y)
{
      m = x; n = y;
}
```

When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

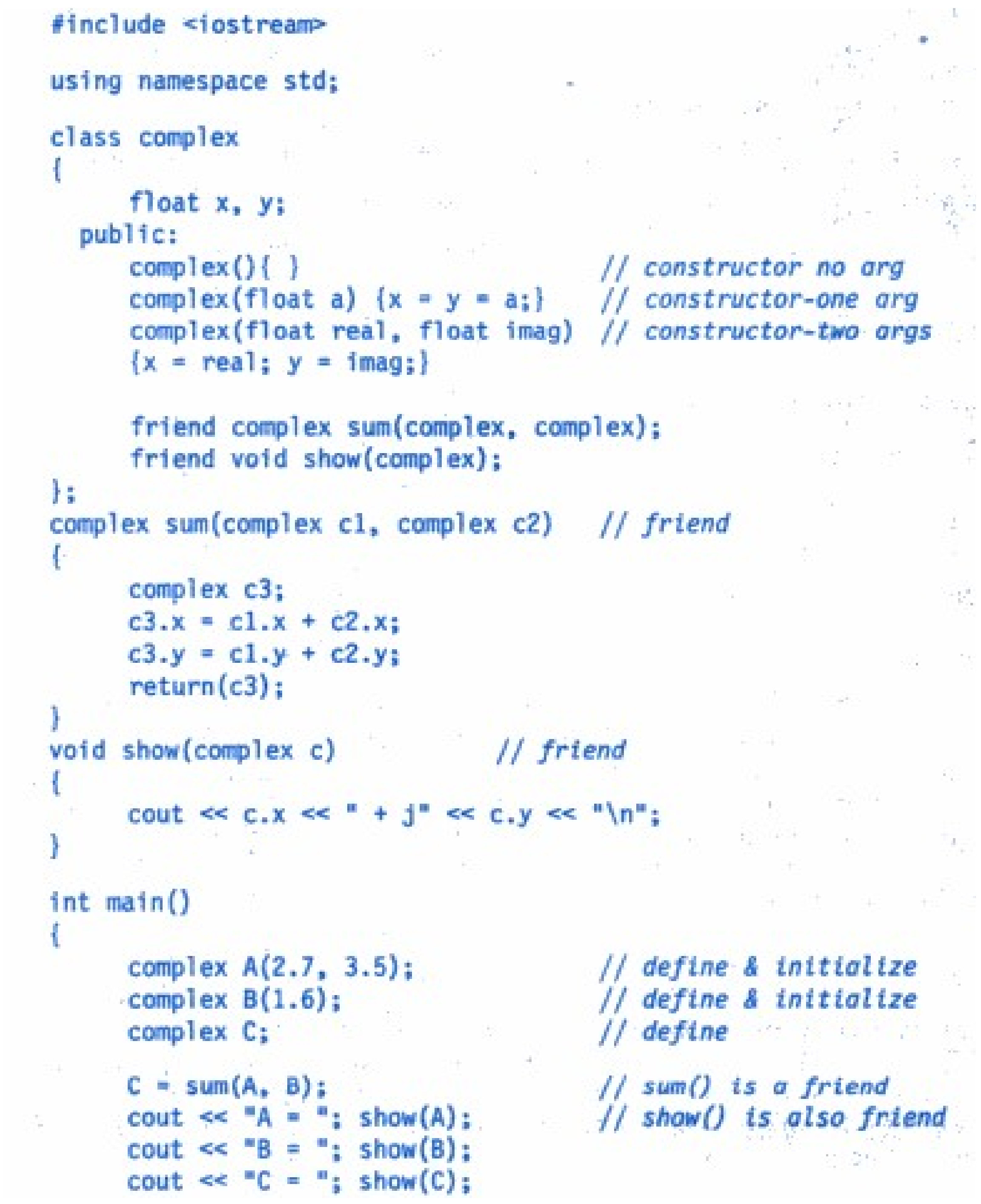
```
integer int1 = integer(0,100); // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);                  // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Example:

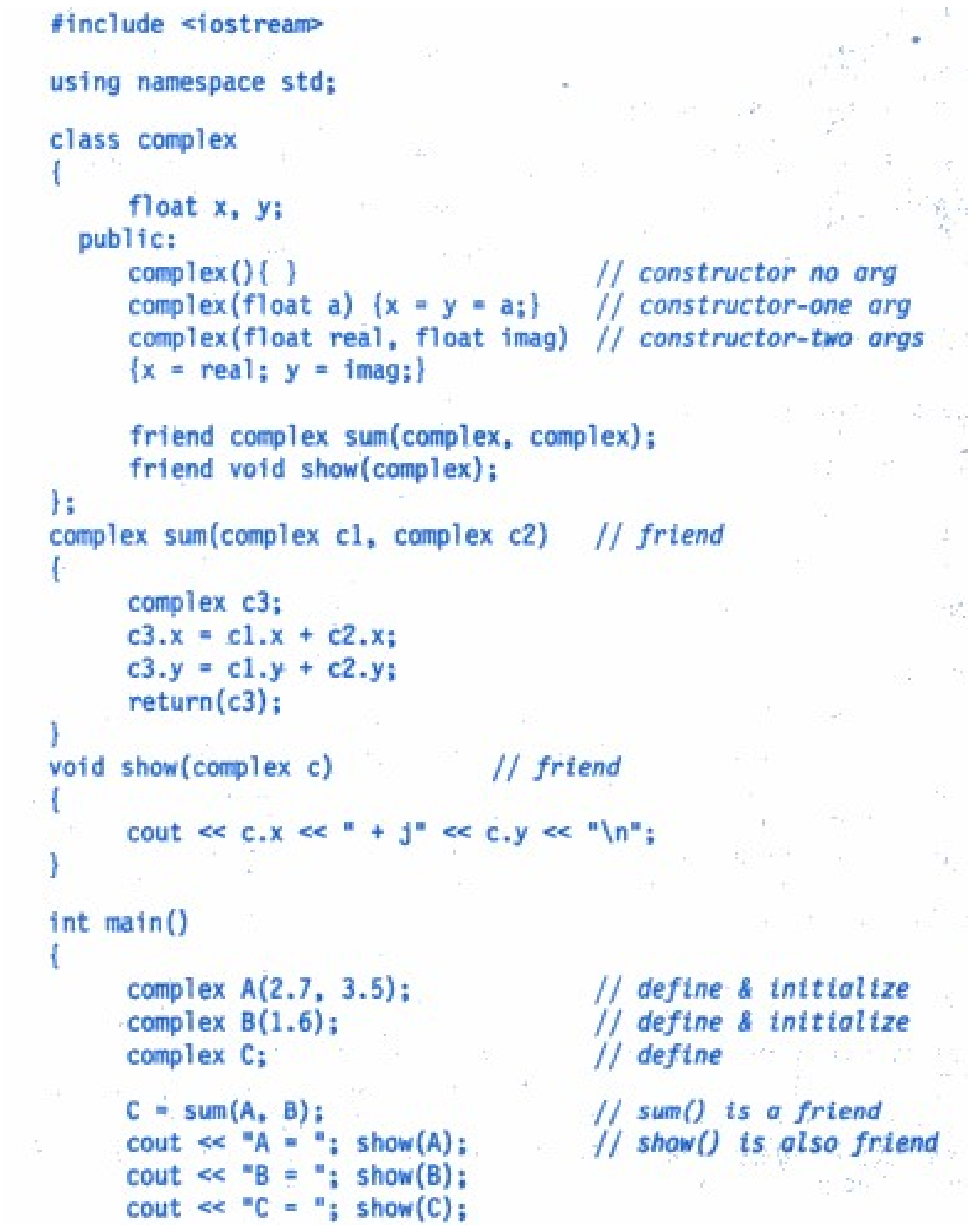


```
#include <iostream>

using namespace std;

class complex
{
        float x, y;
   public:
        complex(){ }                        // constructor no arg
        complex(float a) {x = y = a;}       // constructor-one arg
        complex(float real, float imag)     // constructor-two args
        {x = real; y = imag;}

        friend complex sum(complex, complex);
        friend void show(complex);
};
complex sum(complex c1, complex c2)    // friend
{
        complex c3;
        c3.x = c1.x + c2.x;
        c3.y = c1.y + c2.y;
        return(c3);
}
void show(complex c)              // friend
{
        cout << c.x << " + j" << c.y << "\n";
}

int main()
{
        complex A(2.7, 3.5);            // define & initialize
        complex B(1.6);                 // define & initialize
        complex C;                      // define

        C = sum(A, B);                  // sum() is a friend
        cout << "A = "; show(A);        // show() is also friend
        cout << "B = "; show(B);
        cout << "C = "; show(C);

    return 0;

}
```

Constructor Overloading:

- Constructor can be overloaded in a similar way as function overloading.
- Overloaded constructors have the same name (name of the class) but different number of arguments.
- Depending upon the number and type of arguments passed, specific constructor is called.
- Since, there are multiple constructors present, argument to the constructor should also be passed while creating an object.

Example 2: Constructor overloading



```
#include <iostream>

using namespace std;

class complex
{
      float x, y;
  public:
      complex(){ }                          // constructor no arg
      complex(float a) {x = y = a;}         // constructor-one arg
      complex(float real, float imag)       // constructor-two args
      {x = real; y = imag;}

      friend complex sum(complex, complex);
      friend void show(complex);
};
complex sum(complex c1, complex c2)    // friend
{
      complex c3;
      c3.x = c1.x + c2.x;
      c3.y = c1.y + c2.y;
      return(c3);
}
void show(complex c)                // friend
{
      cout << c.x << " + j" << c.y << "\n";
}

int main()
{
      complex A(2.7, 3.5);              // define & initialize
      complex B(1.6);                   // define & initialize
      complex C;                        // define

      C = sum(A, B);                    // sum() is a friend
      cout << "A = "; show(A);          // show() is also friend
      cout << "B = "; show(B);
      cout << "C = "; show(C);
```

Return 0;

}

DESTRUCTOR:

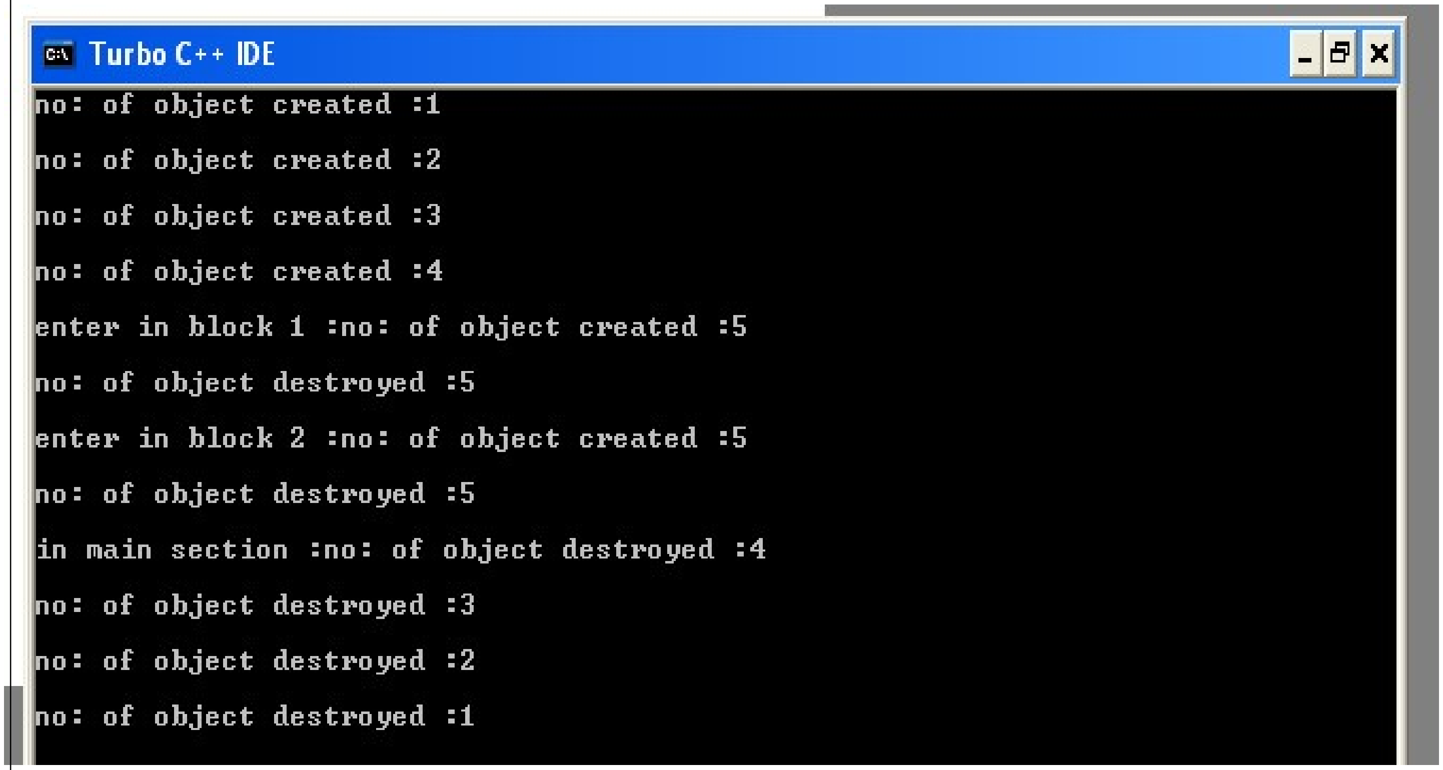
1. Constructor allocates the memory for an object.
2. Destructor deallocates the memory occupied by an object.
3. Like constructor, destructor name and class name must be same, preceded by a tilde (~) sign.
4. Destructors take no argument and have no return value.
5. Destructor is invoked when the object goes out of scope. In other words, Destructor is invoked, when compiler comes out form the function where an object is created.

EXAMPLE OF DESTRUCTOR IS:

```
 #include<iostream.h>
#include<conio.h>
int count=0;
class alpha
{
public:
alpha()
{
cout<<"no: of object created :";
cout<<++count;
cout<<"\n"<<endl;
}
~alpha()
{
cout<<"no: of object destroyed :";
cout<<count--;
cout<<"\n"<<endl;
}
};
void main()
{
{
clrscr();
alpha a1,a2,a3,a4;
{
cout<<"enter in block 1 :";
alpha a5;
}
{
cout<<"enter in block 2 :";
alpha a6;
}
cout<<"in main section :";
}
getch();
```

}

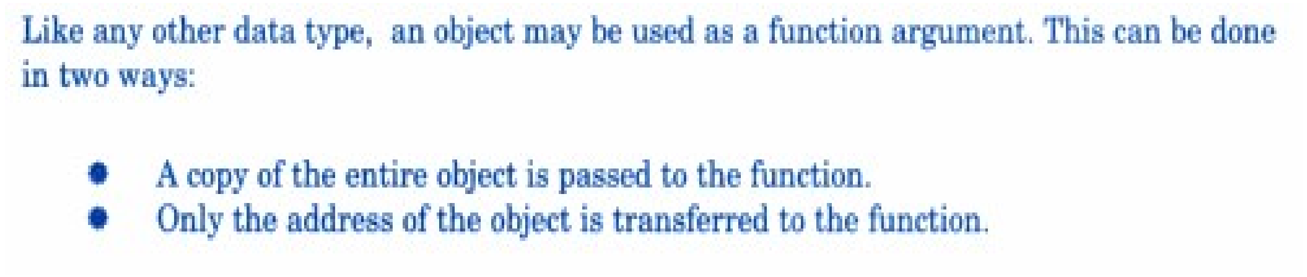
OUTPUT:



```
no: of object created :1
no: of object created :2
no: of object created :3
no: of object created :4
enter in block 1 :no: of object created :5
no: of object destroyed :5
enter in block 2 :no: of object created :5
no: of object destroyed :5
in main section :no: of object destroyed :4
no: of object destroyed :3
no: of object destroyed :2
no: of object destroyed :1
```

Ques: What do you mean by objects as function arguments? Explain pass-by-value and pass-by-reference with example?

Ans:



Like any other data type, an object may be used as a function argument. This can be done in two ways:

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

Pass By Value:

The first method is called *pass-by-value*. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function.

Pass By Reference:

The second method is called *pass-by-reference*. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

EXAMPLE:

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
class A
{
int a;
public:
A(int c)
{
a=c;
}
void friend funct1(A b)       //PASS OBJECT 'b' OF CLASS 'A' BY VALUE
{
 b.a=b.a+5;
 cout<<endl<<"     "<<b.a;
}
Void friend funct2(A &c)        // PASSING OBJECT 'c' OF CLASS 'A' BY REFERENCE
{
 c.a=c.a+10;
 cout<<endl<<"     "<<c.a;
}

void display()
{
cout<<"          a="<<a;
}
};
```

```
void main()
{
clrscr();
A S1(10);
A S2(20);
funct1(S1);                    //will display 15
funct2(S2);                    // will display 30
cout<<endl;
cout<<endl;
S1.display();                  //will display 10
S2.display();                  //will display 30
getch();
}
```

Ques: What is array of objects? Explain with example.

Ans: Object of class represents a single record in memory, if we want more than one record of class type; we have to create an array of object.

Syntax for Array of object

```
class class-name
{
    datatype var1;
    datatype var2;
    - - - - - - - - - -
    datatype varN;

    method1();
    method2();
    - - - - - - - - - -
    methodN();
};
```

```
class-name obj[ size ];
```

Example for Array of object

```
#include<iostream.h>
#include<conio.h>
class Employee
{
    int Id;
    char Name[25];
    int Age;
    long Salary;

    public:
    void GetData()          //Statement 1 : Defining GetData()
    {
        cout<<"\n\tEnter Employee Id : ";
        cin>>Id;
        cout<<"\n\tEnter Employee Name : ";
        cin>>Name;
        cout<<"\n\tEnter Employee Age : ";
        cin>>Age;
        cout<<"\n\tEnter Employee Salary : ";
        cin>>Salary;
    }
    void PutData()          //Statement 2 : Defining PutData()
```

```
        {
            cout<<"\n"<<Id<<"\t"<<Name<<"\t"<<Age<<"\t"<<Salary;
        }
};
void main()
{
        int i;
        Employee E[3];          //Statement 3 : Creating Array of 3 Employees
        for(i=0;i<3;i++)
        {
            cout<<"\nEnter details of "<<i+1<<" Employee";
            E[i].GetData();
        }
        cout<<"\nDetails of Employees";
        for(i=0;i<3;i++)
        E[i].PutData();

}
```

Output :

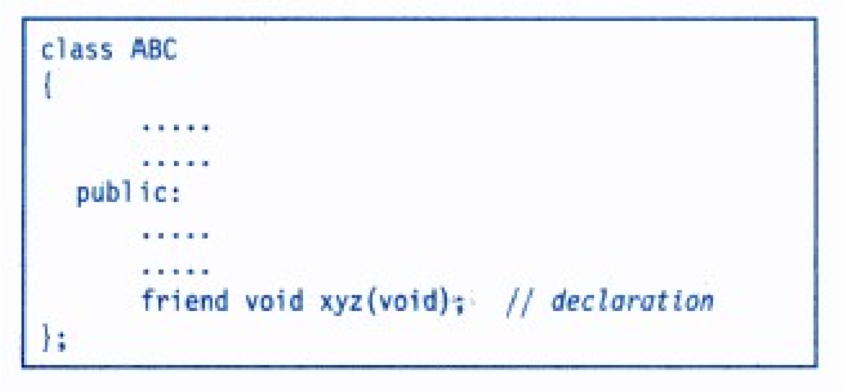
```
        Enter details of 1 Employee
                Enter Employee Id : 101
                Enter Employee Name : Suresh
                Enter Employee Age : 29
                Enter Employee Salary : 45000

        Enter details of 2 Employee
                Enter Employee Id : 102
                Enter Employee Name : Mukesh
```

```
Enter Employee Age : 31
                Enter Employee Salary : 51000

         Enter details of 3 Employee
                Enter Employee Id : 103
                Enter Employee Name : Ramesh
                Enter Employee Age : 28
                Enter Employee Salary : 47000

         Details of Employees
                101     Suresh    29    45000
                102     Mukesh     31    51000
                103     Ramesh     28    47000
```

In the above example, we are getting and displaying the data of 3 employee using array of object. Statement 1 is creating an array of Employee Emp to store the records of 3 employees.

Ques: When do we need friend function? Write a program to add two values defined in different classes using friend function.

1. Private members are accessed only within the class they are declared. Friend function is used to access the private and protected members of different classes.
2. It works as bridge between classes.
3. Friend function must be declared with **friend** keyword.



```
class ABC
{
      .....
      .....
   public:
      .....
      .....
      friend void xyz(void);   // declaration
};
```

Here function xyz() is a friend function of class ‘ABC’.

4. Friend function allows us to share a particular function in two or more classes.

5. For example function income_tax() can be used for manager as well as scientist class. For such cases common friend function can access the private data of both the classes.

Characteristics of friend function:

1) Friend function must be declared in all the classes from which we need to access private or protected members.
2) Friend function will be defined outside the class without specifying the class name and scope resolution operator.
3) Friend function will be invoked like normal function, without any object.
4) It is not in the scope of the class to which it has been declared as friend.
5) It has to use an object name and dot membership operator to access each member name.
6) It can be declared either in public or private part of class.
7) Usually, it has the objects as arguments.

Example of friend function

Class DB stores distance in meters and centimeters and class DM stores distance in feet and inches. Friend function is used for the addition operation.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
class DM;
class DB
{
	float me;
```

```
        float cm;
        public:
        void input(void)
        {
                cout<<"\t:::: DB ::::"<<endl;
                cout<<"Enter the Metes::";
                cin>>me;
                cout<<"Enter the Centimeters::";
                cin>>cm;
        }
        friend void add(DB,DM);
};
class DM
{
        float fe;
        float in;
        public:
        void input(void)
        {
                cout<<"\t:::: DM ::::"<<endl;
                cout<<"Enter the Feet::";
                cin>>fe;
                cout<<"Enter the Inches::";
                cin>>in;
        }
        friend void add(DB,DM);
};
void add(DB a,DM b)
{
        DB t;
        t.cm=(a.me*100)+a.cm+(b.fe*30.48)+(b.in*2.54);
        cout<<endl<<"Display Total of Metres      ::"<<t.cm/100;
        cout<<endl<<"Display Total of Centimeters ::"<<t.cm;
        cout<<endl<<"Display Total of Feet        ::"<<t.cm/30.48;
        cout<<endl<<"Display Total of Inches      ::"<<t.cm/2.54;
}

void main()
{
  DB x;
  DM y;
  clrscr();
  cout<<"Input the value"<<endl;
  x.input();
  y.input();
  cout<<"Display";
  add(x,y);
  getch();
}
```