CHAPTER 2 (THEORY SOLUTION)

1. **What sorts of shortcomings of structure are addressed by classes? Explain giving appropriate example.[PU:2014 fall]**

   The shortcoming of structure that are addressed by classes are as follows:

   1) Class can create a subclass that will inherit parent's properties and methods, whereas Structure does not support the inheritance.
   2) Structure members can be easily accessed by the structure variables in their scope but class members doesn't due to feature of data hiding.
   3) Classes support polymorphism (late binding), whereas structure doesn't.
   4) Also we can use "this" pointers in classes due to which you don't have to explicitly pass an object to a member function.
   5) The structure data type cannot be treated like built in types while performing arithmetic operations. But it is valid in class using the concept of operator overloading.

      For example:

      ```
      struct complex
      {
      int real;
      int imag;
      }
      struct complex c1,c2,c3;
      c3=c1+c2; Is illegal .
      ```

2. **Differentiate between structure and class. Why Class is preferred over structure? Support your answer with suitable examples.*[PU:2016 fall]***

Structure	Class
1. A structure is a collection of variables of different data types under a single unit.	1. A class is a user-defined blueprint or prototype from which objects are created.
2. To define structure we will use “struct” keyword.	2. To define class we will use “class” keyword.
3. A structure has all members public by default.	3. A class has all members private by default.

4. A Structure does not support inheritance.	4. A Class can inherit from another class. Which means class supports inheritance.
5. Structures are good for small and isolated model objects.	5. Classes are suitable for larger or complex objects.
6. Structure is a value type and its object is created on the stack memory.	6. Class is a reference type and its object is created on the heap memory.
7. Function member of the struct cannot be virtual or abstract	7. Function member of the class can be virtual or abstract.

Due to the following reasons class is preferred over structure.

- Structures in C++ doesn't provide data hiding where as a class provides data hiding
- A Structure is not secure and cannot hide its implementation details from the end user while a class is secure and can hide its programming and designing details.
- Classes support polymorphism (late binding), whereas structure doesn't.
- Class support inheritance but structure doesn't.
- "this" pointer will works only with class.
- Class lets us to use constructors and destructors.

3. Explain the various access specifier used in C++ with example.

Access Modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.
There are 3 types of access modifiers available in C++.They are:

1) Public:
All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

2) Private:

The class members declared as private can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

```
#include<iostream>
using namespace std;
class Rectangle
{
private:
float length,breadth,area;
public:
void setdata(float l,float b)
{
length=l;
breadth=b;
}
void disp_area()
{
area=length*breadth;
cout<<"Area="<<area<<endl;
}
};
int main()
{
Rectangle r;
r.setdata(10.5,6);
r.disp_area();
return 0;
}
```

Here, member function disp_area() is public so it is accessible outside class .But data member length ,breadth and area being private they are not accessible outside class. Only member function of that class disp_area() can access it.

3) Protected
class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

Example:
```
#include <iostream>
using namespace std;
class A
{
protected:
int a, b;
public:
void setdata(int x, int y)
{
a = x;
b = y;
}
};
class B : public A
{
public:
void display()
{
cout << "value of a=" << a << endl;
cout << "value of b=" << b << endl;
cout << "Sum of two protected variables a and b = "<< a + b << endl;
}
};
int main()
{
B obj;
obj.setdata(4, 5);
obj.display();
return 0;
}
```
In above example , protected variable a and b is accessed by the derived class B.
From above discussion we can conclude that:

Access Specifier	**Accessible from own class**	**Accessible from derived class**	**Accessible from objects outside class**
public	yes	yes	yes
private	yes	no	no
protected	yes	yes	no

4. What is data hiding/Information hiding? How do you achieve data hiding in C++? Explain with suitable program.*[PU:2019 fall]*

Data Hiding means protecting the data members of a class from an illegal or unauthorized access from outside class. It ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes.

By declaring the data member private in a class, the data members can be hidden from outside the class. Those private data members cannot be accessed by the object directly.

```
#include<iostream>
using namespace std;
class Square
{
private:
int num;
public:
void getdata()
{
cout << "Enter the number"<<endl;;
cin>>num;
}
void display()
{
cout << "Square of a given number= "<< num*num<<endl;
}
};
int main()
{
Square obj;
obj.getdata();
obj.display();
return 0;
}
```

In the above example, the variable "num" is private. Hence this variable can be accessed only by the member function of the same class and is not accessible from anywhere else. Hence outside the classes will be unable to access this variable which is called data hiding. In this way data hiding can be achieved.

5. What is encapsulation? How can encapsulation enforced in C++? Explain with suitable example code. ***[PU:2017 spring]***

Encapsulation is a process of combining data members and functions in a single unit called class.
In encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.
Encapsulation can be enforced by declaring all the variables in the class as private and writing public methods in the class to set and get the values of variable.
Example:

```
#include<iostream>
using namespace std;
class Square
{
private:
int num;
public:
void setnum(int x)
{
num=x;
}
int getnum()
{
return (num*num);
}
};
int main()
{
Square obj;
obj.setnum(5);
cout<<"Square of a number="<<obj.getnum()<<endl;
return 0;
}
```

In the above program the Class square is encapsulated as the variables are declared as private. The get methods getnum() is set as public, this method is used to access these variables. The setter method like setnum() is also declared as public and is used to set the values of the variables.

6. Where do you use friend function? What are the merits and di-merits of friend function?

In some situation non-member function need to access the private data of class or one class wants to access private data of second class and second wants to access private data of first class. This can achieve by using friend functions.
The non-member function that is "friendly" to a class, has full access rights to the private members of the class.

For example when we want to compare two private data members of two different classes in that case you need a common function which can make use of both the private variables of different class. In that case we create a normal function and make friend in both the classes, as to provide access of theirs private variables.

Merits of friend function:

- It can access the private data member of class from outside the class
- Allows sharing private class information by a non-member function.
- It acts as the bridge between two classes by operating on their private data's.
- It is able to access members without need of inheriting the class.
- It provides functions that need data which isn't normally used by the class.

Demerits of friend function:
- It violates the law of data hiding by allowing access to private members of the class from outside the class.
- Breach of data integrity
- Conceptually messy
- Runtime polymorphism in the member cannot be done.
- Size of memory occupied by objects will be maximum

7. Does friend function violate the data hiding? Explain Briefly.[PU:2017 fall]
The concept of data hiding indicates that nonmember functions should not be able to access the private data of class. But, it is possible with the help of a "friend function". That means, a function has to be declared as friend function to give it the authority to access to the private data.
Hence, friend function is not a member function of the class but can access private members of that class. So that's why friend function violate data hiding.

Let’s illustrate the given concept with the following example.

```
#include<iostream>
using namespace std;
class sample
{
private:
int a,b;
public:
void setdata(int x,int y)
{
a=x;
b=y;
}
friend void sum(sample s);
};
void sum(sample s)
{
cout<<"sum="<<(s.a+s.b)<<endl;
}
int main()
{
sample s;
s.setdata(5,10);
sum(s);
return 0;
}
```

Here in class named sample there are two private data members a and b. The function sum() is not a member function but when it is declared as friend in class sample then sum() can access the private data a and b. which shows that friend function violate the concept of data hiding.

8. What are the limitations of static member functions

Limitations of static member functions:

- It doesn’t have a “this” pointer.
- A static member function cannot directly refer to static members.
- Same function can’t have both static and non static types.
- It can’t be virtual.
- A static member function can’t be declared as const or volatile.

9. Data hiding Vs Encapsulation

Data hiding	**Encapsulation**
1) Data Hiding means protecting the members of a class from an illegal or unauthorized access.	1) Encapsulation means wrapping the implementation of data member and methods inside a class.
2) Data hiding focus more on data security.	2) Encapsulation focuses more on hiding the complexity of the system.
3) The data under data hiding is always private and inaccessible.	3) The data under encapsulation may be private or public.
4) When data member of class are private only member function of that class can access it.	4) When implementation of all the data member and methods inside a class are encapsulated, the method name can only describe what action it can perform on an object of that class.
5) Data hiding is a process as well as technique.	5) Encapsulation is a sub-process in data hiding.

10. Abstraction Vs Data hiding

Abstraction	**Data hiding**
1) Abstraction is a mechanism of expressing the necessary properties by hiding the details.	1) Data Hiding means protecting the members of a class from an illegal or unauthorized access
2) It's purpose is to hide complexity.	2) It's purpose is to achieve encapsulation.
3) Class uses the abstraction to derive a new user-defined datatype	3) Data hiding is used in a class to make its data private.
4) It focuses on observable behavior of data.	4) It focuses on data security.

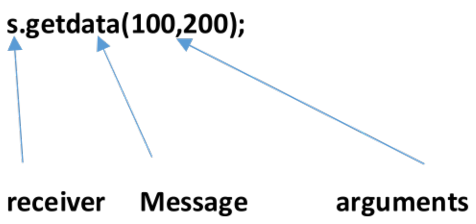
Tutorial solution (Chapter-3) Theory only

1. **What is the difference between message passing and function call? Explain the basic**
   message formalization. [PU:2006 spring]

Message Passing	**Function Call**
1) In a message passing there is a designated receiver for that message; the receiver is some object to which message is sent.	**1)** In function call, there is no designated receiver.
2) Interpretation of the message (that is method is used to respond the message) is determined by the receiver and can vary with different receivers.	**2)** Determination of which method to invoke is very early binding of a name to fragment in procedure calls
3) Message passing must involve name of the object, function name and information to be sent.	3) Simply function name and its arguments is used to made function call.
4) A message is always given to some object, called the receiver.	4) Name will be matched to a message to determine when the method should be executed Signature the combination of return type and argument types.
5) Example: st.getdata(2,5)	5) **Example:** getdata(2,5);

Explanation:
Message passing formalization

Message passing means the dynamic process of asking an object to perform a specific action.
•A message is always given to some object, called the receiver.
•The action performed in response to the message is not fixed but may be differ, depending on the class of the receiver .That is different objects may accept the same message the and yet perform different actions**.**
There are three identifiable parts to any message-passing expression. These are
1) Receiver: the object to which the message is being sent
2) Message selector: the text that indicates the particular message is being sent.
3) Arguments used in responding the message.



st.getdata(100,200);
receiver Message arguments

Example of message passing

```
#include<iostream>
#include<conio.h>
using namespace std;
class student
{
int roll;
public:
void getdata(int x)
{
roll=x;
}
void display()
{
cout<<"Roll number="<<roll;
}
};

int main()
{
student s;
s.getdata(325); //objects passing message
s.display(); //objects passing message
getch();
return 0;
}
```

2. **What are the possible memory errors in programming.[PU:2014 spring]**

Memory errors occur very commonly in programming, and they can affect application stability and correctness. These errors are due to programming bugs. They can be hard to reproduce, hard to debug, and potentially expensive to correct as well. Applications that have memory errors can experience major problems.

Memory errors can be broadly classified into Heap Memory Errors and Stack Memory Errors. Some of the challenging memory errors are:

1. **Invalid Memory Access in stack and heap:** This error occurs when a read or write instruction references unallocated or deallocated memory.
2. **Memory leaks**
   Memory leaks occur when memory is allocated but not released.
3. **Mismatched Allocation/Deallocation**
   This error occurs when a deallocation is attempted with a function that is not the logical counterpart of the allocation function used.
   To avoid mismatched allocation/deallocation, ensure that the right de-allocator is called. In C++, new[] is used for memory allocation and delete[] for freeing up.
4. **Missing allocation**
   This error occurs when freeing memory which has already been freed. This is also called "repeated free" or "double free".
5. **Uninitialized Memory Access**
   This type of memory error will occur when an uninitialized variable is read in your application. To avoid this type of memory error, always initialize variables before using them.

3. **Is it mandatory to use constructor in class. Explain?**

   Constructor is a ‘special’ member function whose task is to initialize the object of its class.
   It is not mandatory to use constructor in a class. As we know, Constructor are invoked automatically when the objects are created.
   So that object are initialized at the time of declaration. There is no need to separate function call to initialize the object. Which reduces coding redundancy and minimizes the programming error such as uninitialized memory access.

4. **Differentiate between constructor and destructor.**

Constructor	Destructor
1) Constructor is used to initialize the instance of the class.	1) Destructor destroys the objects when they are no longer needed.
2) Constructors is called when new instances of class is created.	2) Destructor is called when instances of class is deleted or released.
3) Constructor allocates the memory.	3) Destructor releases the memory.
4) Constructor can have arguments.	4) Destructor cannot have any arguments.
5) Overloading of constructor is possible.	5) Overloading of Destructor is not possible.
6) Constructor have same name as class name.	6) Destructor have same name as class name but with **tlide (~)** sign.

7) Syntax:
Class_name(arguments)
{
//Body of the constructor
}
	7) Syntax:
	~Class_name()
	{
	//Body of the destructor
	}

5. **Discuss the various situations when a copy constructor is automatically invoked. How a default constructor can be equivalent to constructor having default arguments.**

Various situations when the copy constructor is called or automatically invoked are:

- When compiler generates the temporary object.
- When an object is constructed based on the object of the same class.
- When an object of the class is passed to function by values as an argument.
- When an object of the class is returned by value.

Constructor with default arguments is a is parameterized constructor which has default values in arguments. Thus the arguments defined in such constructor are optional. We may or may not pass arguments while defining object of the class. When an object is created with no supplied values, the default values are used. By this way, constructor with default constructor is equivalent to default constructor.

Example:

```
#include<iostream>
using namespace std;
class Box
{
private:
float l,b,h;
public:
Box(float le=10,float br=5,float he=5)
{
        l=le;
        b=br;
        h=he;
}
void displaymember()
{
        cout<<"Length,breadth and height"<<l<<b<<h<<endl;
}
float getvolume()
{
        return (l*b*h);
}
};
int main()
{
        Box b;
        float vol;
        vol=b.getvolume();
        cout<<"Volume="<<vol<<endl;
        return 0;
}
```

6. **What is de-constructor? can you have two destructors in a class? Give example to support your reason.[PU:2014 spring]**

   De-constructor is a member function that destroys the object that have been created by a constructor.
   Destructor doesn't take any arguments, which means destructor cannot be overloaded. So, that there will be only one destructor in a class.
   Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.
   So, that there cannot be more than one destructor (two destructor) in a same class.

   (Write example of destructor yourself)

7. **Differentiate methods of argument passing in constructor and destructor**.

A constructor is allowed to accept the arguments as the arguments can be used to initialize the data members of the class.
A destructor does not accept any arguments as its only work is to deallocate the memory of the object.
In class, there can be multiple constructors which are identified by the number arguments passed.
In class, there is only one destructor.
Constructors can be overload to perform different action under the name of the same constructor whereas, destructors cannot be overloaded.

Example:

```
#include <iostream>
using namespace std;
class test
{
private:
int a;
public:

  test()
  {
           a=10;
   cout<<"Default constructor is called"<<endl;
  }

test(int x)
{
           a=x;
  cout<<"Parameterized constructor is called"<<endl;
}
  ~test()
  {
   cout<<"Destructor is called"<<endl;
   }
   void display()
   {
    cout<<"value of a="<<a<<endl;
   }
};
```

```
int main()
{
  test t1;
  test t2(5);
  t1.display();
  t2.display();
  return 0;
}
```

8. **What do you mean by stack vs Heap? Explain the memory recovery. Explain the use**
   of new and delete operator. [PU:2009 spring]

Stack

- It's a region of computer's memory that stores temporary variables created by each function (including the main() function).
- The stack is a "Last in First Out" data structure and limited in size. Every time a function declares a new variable, it is "pushed" (inserted) onto the stack. Every time a function exits, all of the variables pushed onto the stack by that function, are freed or popped (that is to say, they are deleted).
- Once a stack variable is freed, that region of memory becomes available for other stack variables.

Heap

- The heap is a region of computer's memory that is not managed automatically and is not as tightly managed by the CPU.
- We must allocate and de-allocate variable explicitly. (for allocating variable **new** and for freeing variables **delete** operator is used.
- It does not have a specific limit on memory size.

Memory recovery

Because in most languages objects are dynamically allocated, they must be recovered at run-time. There are two broad approches to this:

- Force the programmer to explicitly say when a value is no longer being used:
  delete aCard;   // C++ example
- Use a garbage collection system that will automatically determine when values are no longer being used, and recover the memory.

Use of new and delete operator

new is used to allocate memory blocks at run time (dynamically). While, delete is used to de-allocate the memory which has been allocated dynamically.

Example of new and delete in C+

```
#include <iostream>
using namespace std;
int main ()
{
int i,n;
int *ptr;
int sum=0;
cout << "How many numbers would you like to Enter? ";
cin >>n;
ptr= new int[n];

for (i=0; i<n; i++)
{
cout << "Enter number:"<<i+1<<endl;
cin >> ptr[i];
}
for (i=0; i<n; i++)
{
sum=sum+ptr[i];
}
cout<<"sum of numbers="<<sum<<endl;
delete[] ptr;
return 0;
}
```

9. **What are the advantages of dynamic memory allocation? Explain with suitable example. [PU:2016 spring]**

The main advantage of using dynamic memory allocation is preventing the wastage of memory or efficient utilization of memory. Let us consider an example:

In static memory allocation the memory is allocated before the execution of program begins (During compilation).In this type of allocation the memory cannot be resized after initial allocation. So it has some limitations. Like

- Wastage of memory
- Overflow of memory

eg. int num[100];
Here, the size of an array has been fixed to 100.If we just enter to 10 elements only, then their will be wastage of 90 memory location and if we need to store more than 100
elements there will be memory overflow.

But, in dynamic memory allocation memory is allocated during runtime.so it helps us to allocate and de-allocate memory during the period of program execution as per requirement. So there will be proper utilization of memory.

In c++ dynamic memory allocation can be achieved by using new and delete operator. **new** is used to allocate memory blocks at run time (dynamically). While, **delete** is used to de-allocate the memory which has been allocated dynamically.

Let us consider an example:

```
#include <iostream>
using namespace std;
int main ()
{
int i,n;
int *ptr;
int sum=0;
cout << "How many numbers would you like to Enter? ";
cin >>n;
ptr= new int[n];
for (i=0; i<n; i++)
{
cout << "Enter number:"<<i+1<<endl;
cin >> ptr[i];
}
for (i=0; i<n; i++)
{
sum=sum+ptr[i];
}
cout<<"sum of numbers="<<sum<<endl;
delete[] ptr;
return 0;
}
```

Here, in above example performs the sum of n numbers .But the value of n will be provided during runtime.so that memory allocation to store n numbers is done during program execution which results proper utilization of memory and preventing from wastage of memory or overflow of memory.

Inheritance Old question solution(Theory)

1. **When base class and derived class have the same function name what happens when derived class object calls the function?[PU 2017 fall]**

When the base class and derived class have the same function name, the derived class function overrides the function that is inherited from base class, which is known as function overriding. Now, when derived class object calls that overridden function, the derived class member function is accessed.

For example:

```
#include<iostream>
using namespace std;
class Base
{
public:
void display()
{
	cout<<"This is base class"<<endl;
}
};
class Derived:public Base
{
public:
void display()
{
	cout<<"This is derived class"<<endl;
}
};
int main()
{
Derived d;
d.display();
return 0;
}
```

Output:

```
This is derived class
```

In this example, the class derived inherits the public member function display() .When we redefine this function in derived class then the inherited members from base are overridden.so when derived class object calls the function display() it actually refers the function in derived class but not the inherited member function in base class.

2. **How inheritance support reusability features of OOP? Explain with example. [PU:2010 spring]**

   Using the concept of inheritance the data member and member function of classes can be reused. When once a class has been written and tested, it can be adapted by another programmer to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. This mechanism of deriving a new class from an old one is called inheritance.
   A derived class includes all features of the base class and then it can add additional features specific to derived class.

Example:

```
#include<iostream>
using namespace std;
class Sum
{
protected:
int a,b;
public:
void getdata()
{
        cout<<"Enter two numbers"<<endl;
        cin>>a>>b;
}
void display()
{
cout<<"a="<<a<<endl;
cout<<"b="<<b<<endl;
}
};
class Result:public Sum
{
        private:
        int total;
        public:
                void disp_result()
                {
                        total=a+b;
                        cout<<"sum="<<total<<endl;
                }
};
```

```
int main()
{
Result r;
r.getdata();
r.display();
r.disp_result();
return 0;
}
```

Here, In above example **Sum** is base class and **Result** is derived class. The data member **named a and b** and member function display () of base class are inherited and they are used by using the object of derived class named **Result.** Hence, we can see that inheritance provides reusability.

3. **How are arguments are sent to base constructors in multiple inheritance ?Who is responsibility of it.[PU:2013 spring]**

   In Multiple Inheritance arguments are sent to base class in the order in which they appear in the declaration of the derived class**.**
   For example:

   ```
   Class gamma: public beta, public alpha
   {
   }
   ```
   Order of execution
   ```
   beta(); base constructor (first)
   alpha(); base constructor(second)
   gamma();derived (last)
   ```

   The constructor of derived class is responsible to supply values to the base class constructor.

Program:

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

```
class alpha
{
int x;
public:
alpha(int a)
{
x=a;
cout<<"Alpha is initialized"<<endl;
}
void showa()
{
cout<<"x="<<x<<endl;
}
};
class beta
{
int y;
public:
beta(int b)
{
y=b;
cout<<"Beta is initialized"<<endl;
}
void showb()
{
cout<<"y="<<y<<endl;
}
};
class gamma:public beta,public alpha
{
int z;
public:
gamma(int a,int b,int c):alpha(a),beta(b)
{
z=c;
cout<<"Gamma is initialized"<<endl;
}
void showg()
{
cout<<"z="<<z<<endl;
}
};
```

```
int main()
{
gamma g(5,10,15);
g.showa();
g.showb();
g.showg();
return 0;
}
```

Output:

```
Beta is initialized
Alpha is initialized
Gamma is initialized
x=5
y=10
z=15
```

Here*,* **beta** is initialized first although it appears second in the derived constructor *as it has been* declared first in the derived class header line

```
class gamma: public beta, public alpha
{ }
```

If we change the order *to*

```
class gamma: public alpha, public beta
{ }
```

then alpha will be initialized first

4. **Class ‘Y’ has been derived from class ‘X’ .The class ‘Y’ does not contain any data members of its own. Does the class ‘Y’ require constructors? If yes why.[PU:2013 spring]**

When Class ‘Y’ has been derived from class ‘X’ and class ‘Y’ does not contain any data members of its own then,

- It is mandatory have constructor in derived class ‘Y’, whether it has data members to be initialized or not, if there is constructor with arguments(parameterized constructor) in base class ‘X’.
- It not necessary to have constructor in derived class ‘Y’ if base class ‘X’ does not contain a constructor with arguments (parameterized constructor).
  Derived class constructor is used to provide the values to the base class constructor.

Example

```
#include<iostream>
using namespace std;
class X
{
	private:
	int a;
	public:
	X (int m)
	{
		a=m;
	}
	void display()
	{
		cout<<"a="<<a<<endl;
	}
};
class Y:public X
{
public:
Y(int b):X(b)
{

}
};
int main()
{
Y obj(5);
obj.display();
return 0;
}
```

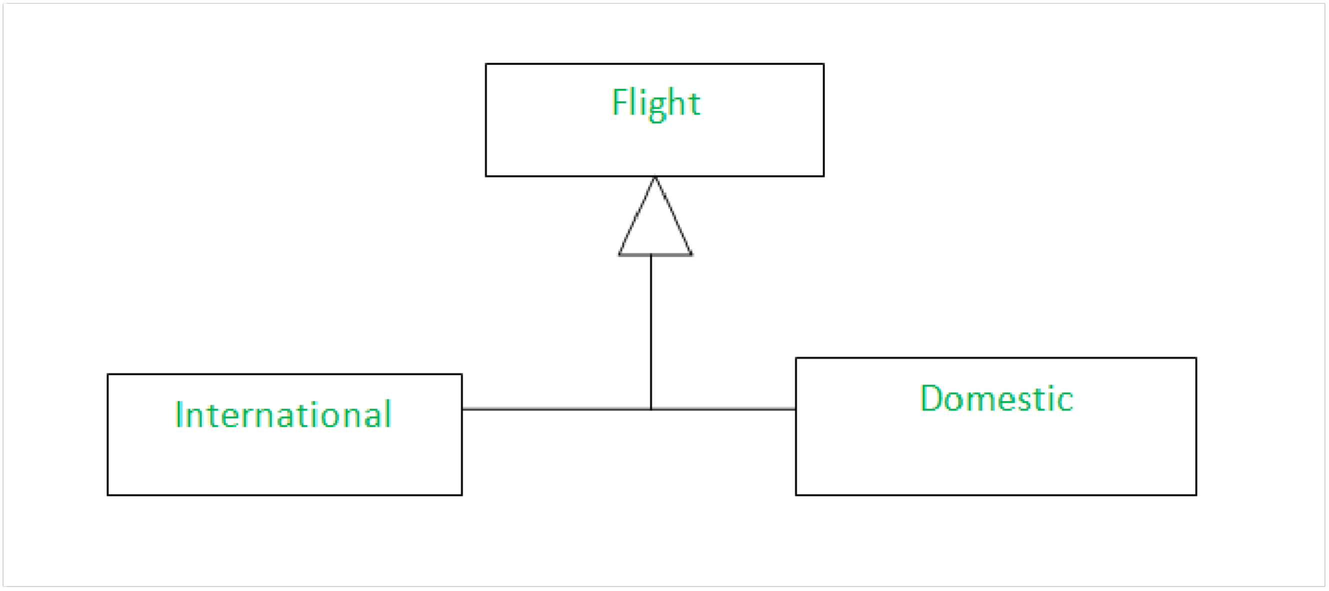
Here, in above example, class 'Y' does not have any data members but there is a parameterized constructor in class 'X' so, there is necessary to have constructor in class 'Y'.

5. **Write a short notes on:**

Generalization[PU:2013 spring]

Generalization is the process of taking out common properties and functionalities from two or more classes and combining them together into another class which acts as the parent class of those classes or what we may say the generalized class of those specialized classes. All the subclasses are a type of superclass. So we can say that subclass “is-A” superclass. Therefore Generalization is termed as “is-A relationship”.

Here is an example of generalization:



In this figure, we see that there are two types of flights so we made a flight class which will contain common properties and then we has an international and domestic class which are an extension of flight class and will have properties of flight as well as their own. Here flight is the parent/superclass and the other two are child/subclass. International Flight “is-a” flight as well as the domestic flight.

6. **Explain how composition provide reusability.**

In composition one class contains object of another class as its member data, which means members of one class is available in another class. Composition exhibits “has-a” relationship. For example, Company **has an** employee.

So, properties of one class can be used by another class independently.

Let us consider an example

```
#include<iostream>
using namespace std;
class Employee
{
int eid;
float salary;
public:
void getdata()
{
cout<<"Enter id and salary of employee"<<endl;
cin>>eid>>salary;
}
void display()
{
cout<<"Emp ID:"<<eid<<endl;
cout<<"Salary:"<<salary<<endl;
}
};
class Company
{
char cname[20];
char department[20];
Employee e;
public:
void getdata()
{
e.getdata();
cout<<"Enter company name and Deparment:"<<endl;
cin>>cname>>department;
}
void display()
{
e.display();
cout<<"Company name:"<<cname<<endl;
cout<<"Department:"<<department<<endl;
}
};
```

```
int main()
{
Company c;
c.getdata();
c.display();
return 0;
}
```

In above example class company contains the object of another class employee. As we know "company **has a** employee" sounds logical .so there exits has a relationship between company and employee. Class company contains the object of class employee and members of class employee are used in class company which provides reusability.

POLYMORPHISM (THEORY SOLUTION)

1) How can polymorphism be achieved during compile time and during runtime? Explain with examples in C++.

Compile time polymorphism can be achieved in two ways:

- Function overloading
- Operator Overloading

Runtime polymorphism can be achieved by using virtual function.

Now, let us discuss them briefly.

Compile time polymorphism

Compile time polymorphism means that an object is bound to its function call at the compile time. That means, there is no ambiguity at the compile time about which a function is linked to a particular function call. This mechanism is called early binding or static binding or static linking.

Function overloading

- The method of using same function name but with different parameter list along with different data type to perform the variety of different tasks is known as function overloading.
- The correct function is to be invoked is determined by checking the number and type of arguments but not function return type.

```
#include<iostream>
using namespace std;
class calcarea
{
public:
int area(int s)
{
return (s*s);
}
int area(int l,int b)
{
return(l*b);
}
float area(float r)
{
return(3.14*r*r);
}
};
int main()
{
calcarea c1,c2,c3;
cout<<"Area of square="<<c1.area(5)<<endl;
cout<<"Area of Rectangle="<<c2.area(5,10)<<endl;
cout<<"Area of Circle="<<c3.area(2.5f)<<endl;
return 0;
}
```

Operator overloading

```
#include<iostream>
using namespace std;
class space
{
private:
int x,y,z;
public:
space(int a,int b,int c)
{
x=a;
y=b;
z=c;
}
void display()
{
cout<<"x="<<x<<endl;
cout<<"y="<<y<<endl;
cout<<"z="<<z<<endl;
}
void operator -()
{
x=-x;
y=-y;
z=-z;
}
}
```

```
int main()
{
space s(5,10,15);
cout<<"s:"<<endl;
s.display();
-s;
cout<<"-s:"<<endl;
s.display();
return 0;
}
```

Runtime polymorphism

We should use virtual functions and pointers to objects to achieve run time polymorphism. For this, we use functions having same name, same number of parameters, and similar type of parameters in both base and derived classes. The function in the base class is declared as virtual using the keyword virtual.
A virtual function uses a single pointer to base class pointer to refer to all the derived objects. When a function in the base class is made virtual, C++ determines which function to use at run time based on the type of object pointed by the base class pointer, rather than the type of the pointer.

```
#include<iostream>
using namespace std;
class base
{
public:
virtual void show()
{
cout<<"Base Class Show function"<<endl;
}
};
class derived:public base
{
public:
void show()
{
cout<<"Derived Class Show function "<<endl;
}
};
```

```
int main()
{
base b1,*bptr;
derived d1;
bptr=&b1;
bptr->show();
bptr=&d1;
bptr->show();
return 0;
}
```

Output:

```
Base Class Show function
Derived Class Show function
```

2) Explain importance of polymorphism with a suitable example.

The word polymorphism means having many forms.
Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

Polymorphism (function overloading) allows one to write more than one method with the same name but with different number of arguments. Hence same method name can be used to perform operations on different data types.

Suppose we want to calculate the area of square, rectangle and circle. We can use the same function named area for all of these by just changing the number and type of arguments.

```
#include<iostream>
using namespace std;
class calcarea
{
public:
int area(int s)
{
return (s*s);
}
int area(int l,int b)
{
return(l*b);
}
float area(float r)
{
return(3.14*r*r);
}
};
int main()
{
calcarea c1,c2,c3;
cout<<"Area of square="<<c1.area(5)<<endl;
cout<<"Area of Rectangle="<<c2.area(5,10)<<endl;
cout<<"Area of Circle="<<c3.area(2.5f)<<endl;
return 0;
}
```

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function

```
#include<iostream>
using namespace std;
class shape
{
public:
virtual void draw()=0;
};
class square:public shape
{
public:
void draw()
{
cout<<"Implementing method to draw square"<<endl;
}
};
class circle:public shape
{
public:
void draw()
{
cout<<"Implementing method to draw circle"<<endl;
}
};
```

```
int main()
{
shape *bptr;
square s;
circle c;
bptr=&s;
bptr->draw();
bptr=&c;
bptr->draw();
return 0;
}
```

Similarly, let us consider an example, operator symbol ‘+’ is used for arithmetic operation between two numbers, however by overloading (means given additional job) it can be used over Complex Object like currency that has Rs and Paisa as its attributes, complex number that has real part and imaginary part as attributes. By overloading same operator ‘+’ can be used for different purpose like concatenation of strings. When same function name is used in defining different function to operate on different data (type or number of data) then this feature of polymorphism is function overloading.

```
#include<iostream>
using namespace std;
class complex
{
private:
int real,imag;
public:
complex()
{ }
complex(int r,int i)
{
real=r;
imag=i;
}
void display()
{
cout<<real<<"+i"<<imag<<endl;
}
complex operator +(complex);
};
```

```
complex complex::operator +(complex c2)
{
complex temp;
temp.real=real+c2.real;
temp.imag=imag+c2.imag;
return temp;
}

int main()
{
complex c1(2,3);
complex c2(2,4);
complex c3;
c3=c1+c2;
cout<<"c1=";
c1.display();
cout<<"c2=";
c2.display();
cout<<"c3=";
c3.display();
return 0;
}
```

3) **How do we make use of a virtual destructor when we need to make sure that the different destructors in inheritance chain are called in order? Explain with an example in C++.**

Let's first see what happens when we do not have a virtual Base class destructor.

```
class Base
{
public:
~Base()
{
cout << "Base class destructor"<<endl;
}
};
```

```
class Derived: public Base
{
public:
~Derived()
{
cout<< "Derived class destructor"<<endl;
}
};
int main()
{
Base* ptr = new Derived; //Base class pointer points to derived class object
delete ptr;
}
```

Output:

```
Base class Destructor
```

In the above example, delete ptr will only call the Base class destructor, which is undesirable because, then the object of Derived class remains un-destructed, because its destructor is never called. Which leads to memory leak situation.

To make sure that the derived class destructor is mandatory called, we make base class destructor as virtual

```
class Base
{
public:
virtual ~Base()
{
cout << "Base class destructor"<<endl;
}
};
```

Output:

```
Derived class Destructor
Base class Destructor
```

When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behavior.

Virtual function vs Pure virtual function [pu:2016 spring]

Virtual function	Pure virtual function
1. A virtual function is a member function of base class which can be redefined by derived class.	1. A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.
2. Classes having virtual functions are not abstract.	2. Base class containing pure virtual function becomes abstract.

3. Virtual return_type function_name(arglist)
{
// code
}
	3. Virtual return_type function_name(arglist)=0;

4. Base class having virtual function can be instantiated i.e. its object can be made.	4. Base class having pure virtual function becomes abstract i.e. it cannot be instantiated
5. All derived class may or may not redefine virtual function of base class.	5. All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.

Function Overloading Vs Function overriding

Function Overloading	Function Overriding
1. It allows us to declare two or more function having same name with different number of parameters or different datatypes of argument.	1. It allow us to declare a function in parent class and child class with same name and signature.
2. Overloading is utilized to have the same number of various functions which act distinctively relying	2. Overriding is required when a determined class function needs to perform some additional or unexpected job in comparison to the base class function.
3. It can occur without inheritance concept.	3. Overriding can only be done when one class is inherited by another classes.

4. It is an example of compile time polymorphism.	4. It is an example of runtime polymorphism.
5. The overloaded functions are always in the same scope.	5. Functions are always in the different scope.
6. we can have any number of overloaded functions	6. We can have only one overriding function in the child class.

Friend function vs Virtual function

Friend function	Virtual Function
1. It is non-member functions that usually have private access to class representation.	1. It is a base class function that can be overridden by a derived class.
2. It is used to access private and protected classes.	2. It is used to ensure that the correct function is called for an object no matter what expression is used to make a function class.
3. It is declared outside the class scope. It is declared using the 'friend' keyword.	3. It is declared within the base class and is usually redefined by a derived class. It is declared using a 'virtual' keyword.
4. It is generally used to give non-member function access to hidden members of a class.	4. It is generally required to tell the compiler to execute dynamic linkage of late binding on function.
5. They support sharing information of class that was previously hidden, provides method of escaping data hiding restrictions of C++, can access members without inheriting class, etc.	5. They support object-oriented programming, ensures that function is overridden, can be friend of other function, etc.
6. It can access private members of the class even while not being a member of that class.	6. It is used so that polymorphism can work.

Operator overloading [PU: 2016 fall]

The mechanism of adding special meaning to an operator is called operator overloading. It provides a flexibility for the creation of new definitions for most C++ operators. Using operator overloading we can give additional meaning to normal C++ operations such as (+,-,=,<=,+= etc.) when they are applied to user defined data types.

let us consider an example, operator symbol '+' is used for arithmetic operation between two numbers, however by overloading (means given additional job) it can be used over Complex Object like currency that has Rs and Paisa as its attributes, complex number that has real part and imaginary part as attributes.

By overloading same operator '+' can be used for different purpose like concatenation of strings, Addition of complex numbers, Addition of two vectors , Addition of two times etc.

General form:

```
return_type operator op(arglist)
{
function body //task defined
}
```

Where,

- return_type is the type of value returned by the specified operation.
- op is the operator being overloaded.
- Operator op is function name, where operator is keyword.