UNIT-4 (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;
cout<<"Area of square="<<c1.area(5)<<endl;
cout<<"Area of Rectangle="<<c1.area(5,10)<<endl;
cout<<"Area of Circle="<<c1.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;
cout<<"Area of square="<<c1.area(5)<<endl;
cout<<"Area of Rectangle="<<c1.area(5,10)<<endl;
cout<<"Area of Circle="<<c1.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.

4. 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.

5.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.

6.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.

7.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.

8.Why does this pointer is widely used than object pointer? Write a program to implement pure polymorphism.

In C++, the this pointer is a special pointer that points to the object that the member function belongs to. It is used to access the member variables and methods of the current object.

The this pointer is widely used in C++ for several reasons:

i)This pointer can be used to refer current class instance variable

When a member variable or method has the same name as a local variable or parameter, the this pointer can be used to disambiguate between them. For example, consider the following code:

```
class MyClass {
   private:
      int x;

   public:
      void setX(int x) {
            this->x = x;
      }
};
```

In this code, the setX method takes a parameter x that has the same name as the member variable x. The this pointer is used to refer to the member variable x, so that it can be assigned the value of the parameter.

ii)this pointer return the objects it points to.

For example, the statement
return *this;
inside a member function will return the object that invoked the function.

Example:

```
#include<iostream>
#include<string.h>
using namespace std;

class person
{
char name[20];
float age;
public:
person()
{ }
person (char n[],float a)
{
strcpy(name,n);
age=a;
}
person& greater(person &x)
{
if(x.age>=age)
{
return x;
}
else
{
return *this;
}
}
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Age:"<<age<<endl;
}
};
```

```
int main()
{
person p1("Ram",52);
person p2("Hari",24);
person p3;
p3=p1.greater(p2);
cout<<"Elder person is:"<<endl;
p3.display();
return 0;
}
```